# Moonmath manual

TechnoBob and the Least Scruples crew

November 11, 2021

Lorem **ipsum** dolor sit amet, consectetur adipiscing elit. Pellentesque semper viverra dictum. Fusce interdum venenatis leo varius vehicula. Etiam ac massa dolor. Quisque vel massa faucibus, facilisis nulla nec, egestas lectus. Sed orci dui, egestas non felis vel, fringilla pretium odio. *Aliquam* vel consectetur felis. Suspendisse justo massa, maximus eget nisi a, maximus gravida mi.

Here is a citation for demonstration: **?**

# 1 Introduction

This is dump from other papers as inspiration for the intro:

Zero knowledge proofs are a class of cryptographic protocols in which one can prove honest computation without revealing the inputs to that computation. A simple high-level example of a zero-knowledge proof is the ability to prove one is of legal voting age without revealing the respective age. In a typical zero knowledge proof system, there are two participants: a prover and a verifier. A prover will present a mathematical proof of computation to a verifier to prove honest computation. The verifier will then confirm whether the prover has performed honest computation based on predefined methods. Zero knowledge proofs are of particular interest to public blockchain activities as the verifier can be codified in smart contracts as opposed to trusted parties or third-party intermediaries.

Zero-knowledge proofs (ZKPs) are an important privacy-enhancing tool from cryptography. Theyallow proving the veracity of a statement, related to confidential data, without revealing any in-formation beyond the validity of the statement. ZKPs were initially developed by the academiccommunity in the 1980s, and have seen tremendous improvements since then. They are now ofpractical feasibility in multiple domains of interest to the industry, and to a large community ofdevelopers and researchers. ZKPs can have a positive impact in industries, agencies, and for per-sonal use, by allowing privacy-preserving applications where designated private data can be madeuseful to third parties, despite not being disclosed to them.

ZKP systems involve at least two parties: a prover and a verifier. The goal of the prover is toconvince the verifier that a statement is true, without revealing any additional information. Forexample, suppose the prover holds a birth certificate digitally signed by an authority. In orderto access some service, the prover may have to prove being at least 18 years old, that is, thatthere exists a birth certificate, tied to the identify of the prover and digitally signed by a trustedcertification authority, stating a birthdate consistent with the age claim. A ZKP allows this, withoutthe prover having to reveal the birthdate.

## 1.1 Target audience

This book is accessible for both beginners and experienced developers alike. Concepts are gradually introduced in a logical and steady pace. Nonetheless, the chapters lend themselves rather well to being read in a different order. More experienced developers might get the most benefit by jumping to the chapters that interest them most. If you like to learn by example, then you should go straight to the chapter on Using Clarinet.

It is assumed that you have a basic understanding of programming and the underlying logical concepts. The first chapter covers the general syntax of Clarity but it does not delve into what programming itself is all about. If this is what you are looking for, then you might have a more difficult time working through this book unless you have an (undiscovered) natural affinity for such topics. Do not let that dissuade you though, find an introductory programming book and press on! The straightforward design of Clarity makes it a great first language to pick up.

# 2 The Zoo of Zero-Knowledge Proofs

First, a list of zero-knowledge proof systems:

1. Pinocchio (2013): Paper
   – Notes: trusted setup
2. BCGTV (2013): Paper
   – Notes: trusted setup, implementation
3. BCTV (2013): Paper
   – Notes: trusted setup, implementation
4. Groth16 (2016): Paper
   – Notes: trusted setup
   – Other resources: Talk in 2019 by Georgios Konstantopoulos
5. GM17 (207): Paper
   – Notes: trusted setup
   – Other resources: later Simulation extractability in ROM, 2018
6. Bulletproofs (2017): Paper
   – Notes: no trusted setup
   – Other resources: Polynomial Commitment Scheme on DL, 2016 and KZG10, Polynomial Commitment Scheme on Pairings, 2010
7. Ligero (2017): Paper
   – Notes: no trusted setup
   – Other resources:
8. Hyrax (2017): Paper
   – Notes: no trusted setup
   – Other resources:
9. STARKs (2018): Paper
   – Notes: no trusted setup
   – Other resources:
10. Aurora (2018): Paper
    – Notes: transparent SNARK
    – Other resources:

11. Sonic (2019): Paper
    - Notes: SNORK - SNARK with universal and updateable trusted setup, PCS-based
    - Other resources: Blog post by Mary Maller from 2019 and work on updateable and universal setup from 2018
12. Libra (2019): Paper
    - Notes: trusted setup
    - Other resources:
13. Spartan (2019): Paper
    - Notes: transparent SNARK
    - Other resources:
14. PLONK (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources: Discussion on Plonk systems and Awesome Plonk list
15. Halo (2019): Paper
    - Notes: no trusted setup, PCS-based, recursive
    - Other resources:
16. Marlin (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources: Rust Github
17. Fractal (2019): Paper
    - Notes: Recursive, transparent SNARK
    - Other resources:
18. SuperSonic (2019): Paper
    - Notes: transparent SNARK, PCS-based
    - Other resources: Attack on DARK compiler in 2021
19. Redshift (2019): Paper
    - Notes: SNORK, PCS-based
    - Other resources:

**Other resources on the zoo:** Awesome ZKP list on Github, ZKP community with the reference document

## To Do List

- Make table for prover time, verifier time, and proof size
- Think of categories - *Achieved Goals*: Trusted setup or not, Post-quantum or not, ...
- Think of categories - *Mathematical background*: Polynomial commitment scheme, ...
- ... while we discuss the points above, we should also discuss a common notation/language for all these things. (E.g. transparent SNARK/no trusted setup/STARK)

## Points to cover while writing

- Make a historical overview over the "discovery" of the different ZKP systems
- Make reader understand what paper is build on what result etc. - the tree of publications!
- Make reader understand the different terminology, e.g. SNARK/SNORK/STARK, PCS, R1CS, updateable, universal, ...
- Make reader understand the mathematical assumptions - and what this means for the zoo.
- Where will the development/evolution go? What are bottlenecks?

#### Other topics I fell into while compiling this list

- Vector commitments: `https://eprint.iacr.org/2020/527.pdf`
- Snarkl: `http://ace.cs.ohio.edu/~gstewart/papers/snaarkl.pdf`
- Virgo?: `https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F19/projects/reports/project5_report_ver2.pdf`

# 3 Preliminaries

## 3.1 Preface and Acknowledgements

This book began as a set of lecture and notes accompanying the zk-Summit 0x and 0xx .... It arose from the desire to collect the scattered information of snarks [] and present them to an audience that does not have a strong backgroud in cryptography []

## 3.2 Purpose of the book

The first version of this book is written by security auditors at Least Authority where we audited quite a few snark based systems. Its included "what we have learned" destilate of the time we spend on various audits.

We intend to let illus- trative examples drive the discussion and present the key concepts of pairing computation with as little machinery as possible. For those that are fresh to pairing-based cryptography, it is our hope that this chapter might be particu- larly useful as a first read and prelude to more complete or advanced expositions (e.g. the related chapters in [Gal12]).

On the other hand, we also hope our beginner-friendly intentions do not leave any sophisticated readers dissatisfied by a lack of formality or generality, so in cases where our discussion does sacrifice completeness, we will at least endeavour to point to where a more thorough exposition can be found.

One advantage of writing a survey on pairing computation in 2012 is that, after more than a decade of intense and fast-paced research by mathematicians and cryptographers around the globe, the field is now racing towards full matu- rity. Therefore, an understanding of this text will equip the reader with most of what they need to know in order to tackle any of the vast literature in this remarkable field, at least for a while yet.

Since we are aiming the discussion at active readers, we have matched every example with a corresponding snippet of (hyperlinked) Magma [BCP97] code 1 , where we take inspiration from the helpful Magma pairing tutorial by Dominguez Perez et al. [DKS09].

Early in the book we will develop examples that we then later extend with most of the things we learn in each chapter. This way we incrementally build a few real world snarks but over full fledged cryptographic systems that are nevertheless simple enough to be computed by pen and paper to illustrate all steps in grwat detail.

## 3.3 How to read this book

Books and papers to read: XXXXXXXXXXX

Software to try: XXXXXXXXXXXXXXXXXXXX

Correctly prescribing the best reading route for a beginner naturally requires individual diagnosis that depends on their prior knowledge and technical preparation.

## 3.4 Cryptological Systems

The science of information security is referred to as *cryptology*. In the broadest sense, it deals with encryption and decryption processes, with digital signatures, identification protocols, cryptographic hash functions, secrets sharing, electronic voting procedures and electronic money. EXPAND

## 3.5 SNARKS

## 3.6 complexity theory

Before we deal with the mathematics behind zero knowledge proof systems, we must first clarify what is meant by the runtime of an algorithm or the time complexity of an entire mathematical problem. This is particularly important for us when we analyze the various snark systems...

For the reader who is interested in complexity theory, we recommend, or example or , as well as the references contained therein.

### 3.6.1 Runtime complexity

The runtime complexity of an algorithm describes, roughly speaking, 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 many more. However, such accuracy is seldom required and is mostly meaningful to consider only the asymptotic computational effort.

In computer science, the runtime of an algorithm is therefore not specified in individual calculation steps, but instead looks for an upper limit which approximates the runtime as soon as the input quantity becomes very large. This can be done using the so-called *Landau notation* (also called big -$\mathcal{O}$-notation) A precise definition would, however, go beyond the scope of this work and we therefore refer the reader to .

For us, only a rough understanding of transit times is important in order to be able to talk about the security of crypographic systems. For example, $\mathcal{O}(n)$ 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 and $\mathcal{O}(2^n)$ stands for an exponential running time (chapter 2.4).

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 ( chapter 3.1).

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 a very difficult and deep-seated question .

For us, the time complexity of the so-called discrete logarithm problem will be important. This is a problem for which we only know slow algorithms on classical computers at the moment, but for which at the same time we cannot rule out that faster algorithms also exist.

STUFF ON CRYPTOGRAPHIC HASH FUNCTIOND

## 3.7 Software Used in This Book

### 3.7.1 Sagemath

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 Sage programming language. Sage is a dialect of the learning-friendly programming language Python, which was extended and optimized for computing with, in and over 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 websit [1]. Note however 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.

We recommend the interested reader, who is not familiar with sagemath to read on the many tutorial before starting this book. For example

[1] `https://doc.sagemath.org/html/en/installation/index.html`

# 4 Arithmetics

How much mathematics is needed to understand zero knowledge proofs? The answer, of course, depends the level of detail the reader wants to understand them. It is possible to describe zero knowledge proofs not using mathematics at all, however, to read a foundational paper like [GROTH16], some understanding of mathematics is needed to at least understand the basic concepts.

Otherwise any student who is interested in learning the concepts, but who has never seen or played with, say, a finite field, or an elliptic curve, may quickly become overwhelmed. This is not so much due to the complexity of the mathematics needs but perhaps more because of the vast amount of technical jargon, of unknown terms, obscure symbols that quickly makes a text unreadable, despite the concepts being actually not that hard. As a result, the reader might either lose interest, or gain some dangerous smattering that in a worst case scenario materializes in immature code.

In this chapter we therefore derive the mathematical concepts needed to understand the basic concepts underlying snark development and we encourage the reader who is not familiar with basic number theory and elliptic curves to take the time and read this chapter until they are able to at least solve most of the simple exercises.

If on the other hand the reader is already skilled in elliptic curve cryptography they might skip this section and only come back for reference and comparison. Maybe the most interesting parts are XXX.

We start at a very basic level and only really requies fundamental concepts like integer arithmetics. At the same time we'll have a focus on teaching the reader how to think mathematically and to understand that there are numbers and methatical structures out there that appear to be very different from the stuff we learned in school and yet on a deeper level they are in deed very similar.

We want to stress however, that our introduction is informal, incomplete and optimized to enable the reader to understand zero knowledge concepts as efficient as possible. Our focus and design choices are so that we give as little theory as necessary but accompanied by a wealth of numerical examples. We found this on the believe, that such an informal, example- driven approach to learning mathematics may ease the beginner's digestion in the initial stages.

For instance, a beginner would be likely to find it more beneficial to first compute a simple toy snark in a pen and paper style all the way through all steps before they dig deeper and actually develop real world production ready systems. Also having already a few simple examples in you head, it is likely easier to only then read the actual academic papers.

However in order to be able to derive those toy example, some mathematical groundwork is needed. This chapter therefore will help the reader to focus on what is important, while at the same time serve as first exercises the reader is encouraged to recompute themself. Every section usually then ends with a list of additional exercises in increasing difficulty order, to help the reader memorising and applying the concepts given.

Overall the goal of this chapter is to provide a reader who is starting with nothing more than basic high school algebra, to be able to solve basic tasks in elliptic curve cryptography without the need of a computer.

We start with a brief recapitulation of basic integer arithmetics like long division, the greatest common divisior and Euclid's algorithm. After that we introduce modular arithmetics as **the most important** skill to compute our pen and paper examples. We then introduce polynomials, compute their analogs to integer arithmetics and introduce the important concept of Lagrange interpolation.

After this practical warm up, we have to introduce some basic algebraic terms like groups and fields, because those terms are all over the place when reading academic papers in the context of zero knowledge proof. The beginner is good advised to memorize those terms and think about them. We define these terms in the general abstract way of mathematics, hoping that the non mathematical trained reader will gradually learn to become comfortable with this style. We then give basic examples and do basic computations with these examples to get familiar with the concepts.

## 4.1 Integer Arithmetics

In a sense, integer arithmetics is at the hart of the foundation of large parts of modern cryptography as it provides the most basic tools to do computations in those systems. Fortunately, most readers will probably remember integer arithmetics from school. It is however important for the rest of the book to be able to apply those concepts to understand and execute computations in the various pen and paper examples that are the main contribution of the moon math manual. We will therefore recapitulate those concepts filling up some knowledge gaps.

In what follows we apply standard mathematical notations and use the symbol $\mathbb{Z}$ for the set of all integers, that is we write

$$\mathbb{Z} := \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \tag{4.1}$$

So whenever you see the symbol $\mathbb{Z}$, think of the set of all integers. If $a \in \mathbb{Z}$ is an integer, we write $|a|$ for the *absolute value* of $a$, that is the the non-negative value of $a$ without regard to its sign. In addition we will use the symbol $\mathbb{N}$ for the set of all counting numbers, that is we write

$$\mathbb{N} := \{0,1,2,3,\ldots\} \tag{4.2}$$

including the number 0. So whenever you see the symbol $\mathbb{N}$, think of the set of all non negative integers.

To make it easier to memorize new concepts and symbols, we might frequently link to definitions (See 4.2 for a definition of $\mathbb{Z}$) in the beginning, but as to many links render a text unreadable, we will assume the reader will become familiar with definitions as the text proceeds at which point we will not link them anymore.

Both sets $\mathbb{N}$ and $\mathbb{Z}$ have a notion of addition as well as multiplication defined on them and also most of us are probably able to do many integer computations in their head, we will frequently invoke the SageMath system (3.7.1) for more complicated computations. One way to invoke the integer-type in sage is:

```
sage: ZZ # A sage notation for the integer type
Integer Ring
sage: NN # A sage notation for the counting number type
Non negative integer semiring
sage: ZZ(5) # Get an element from the Ring of integers
5
```

```
sage: ZZ(5) + ZZ(3)
8
sage: ZZ(5) * NN(3)
15
sage: ZZ.random_element(10**50)
70550236113498624702091184075467131816309441524857
sage: ZZ(27713).str(2) # Binary string representation
110110001000001
sage: NN(27713).str(2) # Binary string representation
110110001000001
sage: ZZ(27713).str(16) # Hexadecimal string representation
6c41
```

Of particular interest for us are the so called *prime numbers*, which are counting numbers $p \in \mathbb{N}$ with $p \geq 2$, which are divisible by themself and by 1 only. Prime numbers are called *odd* if they are not the number 2. We write $\mathbb{P}$ for the set of all prime numbers and $\mathbb{P}_{\geq 3}$ for the set of all odd prime numbers. $\mathbb{P}$ is infinite and can be ordered according to size, so that we can write them as

$$2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,\ldots \tag{4.3}$$

which is sequence $A000040$ in OEIS. In particular, we can talk about small and large prime numbers.

As the *fundamental theorem of arithmetics* tells us, prime numbers are in a certain sense the basic building blocks from which all other natural numbers are composed. To see that, let $n \in \mathbb{N}_{\geq 2}$ be any natural number. Then there are always prime numbers $p_1, p_2, \ldots, p_k \in \mathbb{P}$, such that

$$n = p_1 \cdot p_2 \cdot \ldots \cdot p_k\,. \tag{4.4}$$

This representation is unique, except for permutations in the factors and is called the **prime factorization** of $n$.

**Example 1** (Prime Factorization)**.** *To see what we mean by prime factorization of a number, lets look at the number* $19214758032624000 \in \mathbb{N}$*. To get its prime factors, we can successively divide it by all prime numbers in ascending order starting with* 2*. We get*

$$19214758032624000 = 2\cdot 2\cdot 2\cdot 2\cdot 2\cdot 2\cdot 2\cdot 3\cdot 3\cdot 3\cdot 5\cdot 5\cdot 5\cdot 7\cdot 11\cdot 17\cdot 17\cdot 23\cdot 43\cdot 43\cdot 47$$

*We can double check our findings invoking sage, which provides an algorithm to factor counting numbers:*

```
sage: n = NN(19214758032624000)
sage: factor(n)
2^7 * 3^3 * 5^3 * 7 * 11 * 17^2 * 23 * 43^2 * 47
```

Having done the computation from the previous example, reveals an important observation: Computing the factorization was computationally expensive, while on the other hand, giving a string of prime numbers, computing their product is fast.

From this an important question arises: How fast we can compute the prime factorization of a natural number? This is the famous *factorization problem* and as far as we know, there is no method on a classical Turing machine that is able to compute this representation in polynomial time. The fastest algorithms known today run sub-exponentially, with $\mathcal{O}((1+\varepsilon)^n)$ and some $\varepsilon > 0$.

It follows that number factorization $\Leftrightarrow$ prime number multiplication is an example of, what is called a one-way function. Something that is easy to compute in one direction, but hard to compute in the other direction. Existence of one way functions like this are basic cryptographic assumptions, that the security of many crypto systems is based on.

It should be pointed out however hat the American mathematician Peter Williston Shor developed an algorithm in 1994 which can calculate the prime factor representation of a natural number in polynomial time on a quantum computer. The consequence of this is, of course, that crptyosystems, which are based on the time complexity of the prime factor problem, are unsafe as soon as practically usable quantum computers are available.

**Exercise 1.** *What is the absolute value of the integers* $-123$, $27$ *and* $0$*?*

**Exercise 2.** *Compute the factorization of* $6469693230$ *and double check your results using sage.*

**Exercise 3.** *Consider the following equation* $4 \cdot x + 21 = 5$*. Compute the set of all solutions x under the following assumptions: 1. The equation is defined over the type of natural numbers. 2. The equation is defined over the type of integers.*

**Exercise 4.** *Consider the following equation* $2x^3 - x^2 - 2x = -1$*. Compute the set of all solutions x under the following assumptions: 1. The equation is defined over the type of natural numbers. 2. The equation is defined over the type of integers. 2. The equation is defined over the type* $\mathbb{Q}$ *of fractions.*

**Euclidean Division** In general there is no division defined in the usual sense for integers, as for example 7 divided by 3 will not be an integer again. However it is possible to divide any two integers with a remainder. So for example 7 divided by 3 is equal to 2 with a remainder of 1, since $7 = 2 \cdot 3 + 1$.

Doing integer division like this is probably something many of us remember from school. It is usually called *Euclidean division*, or division with remainder and it is an important technique, that every reader must become familiar with to understand many concepts in this book. The precise definition is as follows:

Let $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ be two integers with $b \neq 0$. Then there is always another integer $m \in \mathbb{Z}$ and a counting number $r \in \mathbb{N}$, with $0 \leq r < |b|$ such that

$$a = m \cdot b + r \tag{4.5}$$

This decomposition of $a$ given $b$ is called *Euklidean division*, where $a$ is called the *divident*, $b$ is called the *divisor*, $m$ is called the *quotient* and $r$ is called the *remainder*.

**Notation and Symbols 1.** *Suppose that the numbers* $a, b, m$ *and* $r$ *satisfy equation (4.5). Then we often write*

$$a \text{ div } b := m, \qquad a \text{ mod } b := r \tag{4.6}$$

*to describe the quotient and the remainder of the Euclidean division. We also say, that an integer a is divisible by another integer b if* $a \text{ mod } b = 0$ *holds. In this case we also write* $b|a$*.*

So in a Nutshell Euclidean division is a process of dividing one integer by another, in a way that produces a quotient and a non negative remainder the latter of which is smaller than the absolute value of the divisor. It can be shown, that both the quotient and the remainder always exist and are unique, as long as the divident is different from 0.

A special situation occurs, is the remainder is zero, because in this special case the divident *is* divisible by the divisor. Our notation $b|a$ reflects that.

**Example 2.** *Applying Euclidean division and our previously defined notation 4.25 to the divisor* $-17$ *and the divident* $4$, *we get*

$$-17 \; div \; 4 = -5, \qquad -17 \; mod \; 4 = 3$$

*because* $-17 = -5 \cdot 4 + 3$ *is the Euclidean division of* $-17$ *and* $4$ *(Since the remainder is by definition a non-negative number). In this case* $4$ *does not divide* $-17$ *as the reminder is not zero. Writing* $4|-17$ *therefore has no meaning. On the other hand we can write* $4|12$, *since* $4$ *divides* $12$, *as* $12 \; mod \; 4 = 0$. *We can invoke sagemath to do the computation for us. We get*

```
sage: ZZ(-17) // ZZ(4) # Integer quotient
-5
sage: ZZ(-17) % ZZ(4) # remainder
3
sage: ZZ(4).divides(ZZ(-17)) # self divides other
False
sage: ZZ(4).divides(ZZ(12))
True
```

Methods to compute Euclidean division for integers are called *integer division algorithms*. Probably the best known algorithm is the so called *long division*, that most of us might have learned in school. It should be noted however that there are faster methods like *Newton–Raphson division*.

As long division is the standard method used for pen-&-paper division of multi-digit numbers expressed in decimal notation, the reader should become familiar with it as we use it all over this book when we do simple pen-and-paper computations. However instead of defining the algorithm formally, we rather give some examples, that hopelly will make the process clear

**Example 3** (Integer Long Division)**.** *To give an example of integer long division algorithm, lets divide the integer* $a = 143785$ *by the number* $b = 17$. *Our goal is therefore to find solutions to equation 4.5, that is we need to find the quotient* $m \in \mathbb{Z}$ *and the reminder* $r \in \mathbb{N}$ *such that* $143785 = m \cdot 17 + r$. *Using a notation that is mostly used in Commonwealth countries, we compute*

$$\begin{array}{r} 8457 \\ 17\overline{)143785} \\ \underline{136\phantom{000}} \\ 77\phantom{00} \\ \underline{68\phantom{00}} \\ 98\phantom{0} \\ \underline{85\phantom{0}} \\ 135 \\ \underline{119} \\ 16 \end{array} \tag{4.7}$$

*We therefore get* $m = 8457$ *as well as* $r = 16$ *and indeed we have* $143785 = 8457 \cdot 17 + 16$, *which we can double check invoking sage:*

```
sage: ZZ(143785).quo_rem(ZZ(17)) # Euclidean Division
(8457, 16)
sage: ZZ(143785) == ZZ(8457)*ZZ(17) + ZZ(16) # check
```

```
True
```

*In a nutshell, the algorithm loops through the digits of the dividend from the left to right, subtracting the largest possible multiple of the divisor (at the digit level) at each stage; the multiples then become the digits of the quotient, and the remainder is the first digit of the dividend.*

**Exercise 5** (Integer Long Division)**.** *Find an* $m \in \mathbb{Z}$ *as well as an* $r \in \mathbb{N}$ *such that* $a = m \cdot b + r$ *holds for the following pairs* $(a,b) = (27,5)$, $(a,b) = (27,-5)$, $(a,b) = (127,0)$, $(a,b) = (-1687,11)$ *and . In which cases are your solutions unique?*

$$(a,b) = (0,7)$$

**Exercise 6** (Long Division Algorithm)**.** *Write an algorithm in pseudocode that computes integer long division, handling all edge cases properly.*

**The Extended Euclidean Algorithm** One of the most critical parts in this book is modular arithmetics XXX and its application in the computations in so called finite fields, as we explain in XXX. In modular arithmetics it is sometimes possible to define actual division and multiplicative inverses of numbers, that is very different from inverses as we know them from other systems like factional numbers.

However, to actually compute those inverses we have to get familiar with the so-called *extended Euclidean algorithm*. To recapitulate jargon first, the *greatest common divisor* (GCD) of two nonzero integers $a$ and $b$ is the greatest non-zero counting number $d$ such that $d$ divides both $a$ and $b$; that is $d|a$ as well as $d|b$. We write $gcd(a,b) := d$ for this number. In addition two counting numbers are called **relative prime** or **coprime**, if their greatest common divisor is 1.

The extended Euclidean algorithm is then a method to calculate the greatest common divisor of two counting numbers $a$ and $b \in \mathbb{N}$, as well as two additional integers $s,t \in \mathbb{Z}$, such that the equation

$$gcd(a,b) = s \cdot a + t \cdot b \tag{4.8}$$

holds. The following pseudocode shows in detail how to calculate these numbers with the extended Euclidean algorithm:

The algorithm is simple enough to be done effectively in pen-&-paper examples, where it is common to write it as a table where the rows represent the while-loop and the columns represent the values of the the array $r$, $s$ and $t$ with index $k$. The following example provides a simple execution:

**Example 4.** *To illustrate the algorithm, lets apply it to the numbers* $a = 12$ *and* $b = 5$*. Since* $12, 5 \in \mathbb{N}$ *as well as* $12 \geq 5$ *all requirements are meat and we compute*

| $k$ | $r_k$ | $s_k$ | $t_k = (r_k - s_k \cdot a)$ $div$ $b$ |
|---|---|---|---|
| *0* | *12* | *1* | *0* |
| *1* | *5* | *0* | *1* |
| *2* | *2* | *1* | *-2* |
| *3* | *1* | *-2* | *5* |

*From this we can see that* 12 *and* 5 *are relatively prime (coprime), since their greatest common divisor is* $gcd(12,5) = 1$ *and that the equation* $1 = (-2) \cdot 12 + 5 \cdot 5$ *holds. We can also invoke sage to double check our findings:*

```
sage: ZZ(12).xgcd(ZZ(5)) # (gcd(a,b),s,t)
(1, -2, 5)
```

**Algorithm 1** Extended Euclidean Algorithm

**Require:** $a, b \in \mathbb{N}$ with $a \geq b$

**procedure** EXT-EUCLID$(a, b)$

$r_0 \leftarrow a$

$r_1 \leftarrow b$

$s_0 \leftarrow 1$

$s_1 \leftarrow 0$

$k \leftarrow 1$

**while** $r_k \neq 0$ **do**

$q_k \leftarrow r_{k-1}$ div $r_k$

$r_{k+1} \leftarrow r_{k-1} - q_k \cdot r_k$

$s_{k+1} \leftarrow s_{k-1} - q_k \cdot s_k$

$k \leftarrow k+1$

**end while**

**return** $gcd(a,b) \leftarrow r_{k-1}$, $s \leftarrow s_{k-1}$ and $t := (r_{k-1} - s_{k-1} \cdot a)$ div $b$

**end procedure**

**Ensure:** $gcd(a,b) = s \cdot a + t \cdot b$

**Exercise 7** (Extended Euclidean Algorithm)**.** *Find integers $s, t \in \mathbb{Z}$ such that $gcd(a,b) = s \cdot a + t \cdot b$ holds for the following pairs $(a,b) = (45,10)$, $(a,b) = (13,11)$, $(a,b) = (13,12)$. What pairs $(a,b)$ are coprime?*

**Exercise 8** (Towards Prime fields)**.** *Let $n \in \mathbb{N}$ be a counting number and $p$ a prime number, such that $n < p$. What is the greatest common divisor $gcd(p,n)$?*

**Exercise 9.** *Find all numbers $k \in \mathbb{N}$ with $0 \leq k \leq 100$ such that $gcd(100,k) = 5$.*

**Exercise 10.** *Show that $gcd(n,m) = gcd(n+m,m)$ for all $n, m \in \mathbb{N}$.*

## 4.2 Modular arithmetic

In mathematics, so called *modular arithmetic* is a system of arithmetic for integers, where numbers "wrap around" when reaching a certain value, much like calculations on a clock wrap around whenever the value exceeds the number 12, 24 or 60, depending on your clock. For example if the clock shows that it is 11 o'clock, then 20 hours later it will be 7 o'clock, not 31 o'clock. The letter of which has no meaning on a normal clock that shows hours.

The number at which the wrap occurs is called the *modulus*. Modular arithmetics generalizes the clock example to arbitrary moduli and studies equations and phenomena that arizes in this new kind of arithmetics. It is of central importance for understanding most modern crypto systems, in large parts because the exponentiation function has an inverse with respect to certain moduli, that is hard to compute. In addition we will see that it provides the foundation of what is called finite fields (See XXX)

Also it will turn out that modular arithmetic appears very different from ordinary integer arithmetic that we are all familiar with, we encouurage the interested reader to work through the example and to discover that, once they accept that this is a new kind of calculations, its actually not that hard.

**Congruency** In what follows, let $n \in \mathbb{N}$ with $n \geq 2$ be a fixed counting number, that we will call the *modulus* of our modular arithmetics system. With such an $n$ given, we can then group integers into classes, by saying that two integers are in the same class, whenever their Euklidean division **??** by $n$ will give the same remainder. We then say that two numbers are *congruent* whenever they are in the same class.

**Example 5.** *If we choose* $n = 12$ *as in our clock example, then the integers* $-7$*,* $5$*,* $17$ *and* $29$ *are all congruent with respect to* $12$*, since all of them have the remainder* $5$ *if we Euklidean divide them by* $12$*. In the picture of an analog* $12$*-hour clock, starting at* $5$ *o'clock, when we add* $12$ *hours we are again at* $5$ *o'clock, representing the number* $17$*. On the other hand when we subtract* $12$ *hours, we are at* $5$ *o'clock again, representing the number* $-7$*.*

We can formulize this intuition of what congruency should be into a proper definition utilizing Euklidean division as explained previously 4.1: Let $a, b \in \mathbb{Z}$ be two integers and $n \in \mathbb{N}$ a natural number. Then $a$ and $b$ are said to be **congruent with respect to the modulus** $n$, if and only if the equation

$$a \bmod n = b \bmod n \tag{4.9}$$

holds. If on the other hand two numbers are not congruent with respect to a given modulus $n$, we call them *incongruent* w.r.t. $n$.

A *congruency* is then nothing but an equation "up to congruency", which means that the equation only needs to hold if we take the modulus on both sides. In which case we write

$$a \equiv b \pmod{n} \tag{4.10}$$

**Exercise 11.** *Which of the following pairs of numbers are congruent with respect to the modulus* $13$*:* $(5,19)$*,* $(13,0)$*,* $(-4,9)$*,* $(0,0)$*.*

**Exercise 12.** *Find all integers x, such that the congruency* $x \equiv 4 \pmod{6}$ *is satisfied.*

**Modular Arithmetics** On particulary nice thing about congruencies is, that we can do calculations (arithmetics), much like we can with integer equations. That is we can add or multiply numbers on both sides. The main difference is probably that the congruency $a \equiv b \pmod{n}$ is only equivalent to the congruency $k \cdot a \equiv k \cdot b \pmod{n}$ for some non zero integer $k \in \mathbb{Z}$, whenever $k$ and the modulus $n$ are coprime. The following list gives a set of useful rules:

Suppose that the congurencies $a_1 \equiv b_1 \pmod{n}$ as well as $a_2 \equiv b_2 \pmod{n}$ are satisfied for integers $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and that $k \in \mathbb{Z}$ is another integer. Then:

- $a_1 + k \equiv b_1 + k \pmod{n}$ (compatibility with translation)
- $k \cdot a_1 \equiv k \cdot b_1 \pmod{n}$ (compatibility with scaling)
- $a_1 + a_2 \equiv b_1 + b_2 \pmod{n}$ (compatibility with addition)
- $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{n}$ (compatibility with multiplication)

Other rules like compatibility with subtraction and exponentiation follow from this rules, as for example compatibility with subtraction is compatibility with scaling by $k = -1$ and compatibility with addition.

Note that the previous rules are implications not equivalences, which means that you can not necessarily reverse those rules. The following rules makes this precise:

- If $a_1 + k \equiv b_1 + k \pmod{n}$, then $a_1 \equiv b_1 \pmod{n}$

- If $k \cdot a_1 \equiv k \cdot b_1 \pmod{n}$ and $k$ is coprime with $n$, then $a_1 \equiv b_1 \pmod{n}$
- If $k \cdot a_1 \equiv k \cdot b_1 \pmod{k \cdot n}$, then $a_1 \equiv b_1 \pmod{n}$

Another property of congruencies, not known in the traditional arithmetics of integers is the so called *Fermat's Little Theorem*. In simple words, it says that in modular arithmetics every number raised to the power of a prime number modulus is congruent to the number itself. Or, to be more precise, if $p \in \mathbb{P}$ is a prime number and $k \in \mathbb{Z}$ is an integer, then:

$$k^p \equiv k \pmod{p}, \tag{4.11}$$

If $k$ is coprime to $p$, then we can divide both sides of this congruency by $k$ and rewrite the expression into the equivalent form

$$k^{p-1} \equiv 1 \pmod{p} \tag{4.12}$$

We can invoke sage, to compute examples for both $k$ being coprime and not coprime to $p$:

```
sage: ZZ(137).gcd(ZZ(64))
1
sage: ZZ(64)** ZZ(137) % ZZ(137) == ZZ(64) % ZZ(137)
True
sage: ZZ(64)** ZZ(137-1) % ZZ(137) == ZZ(1) % ZZ(137)
True
sage: ZZ(1918).gcd(ZZ(137))
137
sage: ZZ(1918)** ZZ(137) % ZZ(137) == ZZ(1918) % ZZ(137)
True
sage: ZZ(1918)** ZZ(137-1) % ZZ(137) == ZZ(1) % ZZ(137)
False
```

Now, since this was a lot to digest for a reader who has never encountered modular arithmetics before, lets compute an example that contains most of the stuff we just described:

**Example 6.** *Assume that we choose the modulus* 6 *and that our task is to solve the following congruency for* $x \in \mathbb{Z}$

$$7 \cdot (2x+21) + 11 \equiv x - 102 \pmod{6}$$

*As many rules for congruencies are more or less same as for integers, we can proceed in a way similar, as we would if we had an equation to solve. The first thing we notice, is that* $7 \cdot (2x+21) + 11 = 14x + 158$*, since both sides of a congruency contain ordinary integers. We can therefore rewrite the congruency into the equivalent form*

$$14x + 158 \equiv x - 102 \pmod{6}$$

*In a next step we want to shift all encounters of x to left and every other term to the right. So we applay the "compatibility with translation" rules two times. In a first step we choose* $k = -x$ *and in a second step we choose* $k = -158$*. Since "compatibility with translation" transforms a congruency into an equivalent form, the solution set will not change and we get*

$$\begin{aligned} 14x + 158 \equiv x - 102 \pmod{6} \Leftrightarrow \\ 14x - x + 158 - 158 \equiv x - x - 102 - 158 \pmod{6} \Leftrightarrow \\ 13x \equiv -260 \pmod{6} \end{aligned}$$

*If our congruency would just be a normal integer equation, we would divide both sides by* $13$ *to get* $x = -20$ *as our solution. However in case of a congruency we need to make sure that the modulus and the number we want to divide by are coprime first. Only then will we get an equivalent expression. So we need to the greatest common divisor* $gcd(13,6)$ *and since* $13$ *is prime and* $6$ *is not a multiple of* $13$*, we know* $gcd(13,6) = 1$*, so both numbers are indeed coprime. We therefore compute*

$$13x \equiv -260 \quad (\ mod\ 6\ ) \Leftrightarrow x \equiv -20 \quad (\ mod\ 6\ )$$

*Our task is now to find all integers x, such that x is congruent to* $-20$ *with respect to the modulus* $6$*. So we have to find all x such*

$$x\ mod\ 6 = -20\ mod\ 6$$

*Since* $-4 \cdot 6 + 4 = -20$ *we know* $-20\ mod\ 6 = 4$ *and hence we know that* $x = 4$ *is a solution. However* $22$ *is another solution since* $22\ mod\ 6 = 4$ *as well and so is* $-20$*. In fact there are infinite many solutions given by the set*

$$\{\ldots, -8, -2, 4, 10, 16, \ldots\} = \{4 + k \cdot 6 \mid k \in \mathbb{Z}\}$$

*Putting all this together we have shown that the every x from the set* $\{x = 4 + k \cdot 6 \mid k \in \mathbb{Z}\}$ *is a solution to the congruency* $7 \cdot (2x + 21) + 11 \equiv x - 102 \quad (\ mod\ 6\ )$*. We double ckeck for, say,* $x = 4$ *as well as* $x = 14 + 12 \cdot 6 = 86$ *using sage:*

```
sage: (ZZ(7)* (ZZ(2)*ZZ(4) + ZZ(21)) + ZZ(11))  % ZZ(6) == (ZZ(4) - ZZ(102))  % ZZ(6)
True
sage: (ZZ(7)* (ZZ(2)*ZZ(76) + ZZ(21)) + ZZ(11))  % ZZ(6) == (ZZ(76) - ZZ(102))  % ZZ(6)
True
```

The discouraged reader, who at this point thinks that modular aithmetics is to complicated, might consider two thinks: First, computing congruencies in modular arithmetics is not really more complicated then computations in more familiar number systems like fractional numbers. Its just a matter of getting used to it. Second, the theory of prime fields (and more general residue class rings) takes a different view on modular rithmetics with the attempt to simplify thinks. In other words, once we understand prime field arithmetics, thinks become conceptually cleaner and more easy to compute.

**Exercise 13.** *Choose the modulus* $13$ *and find all solutions* $x \in \mathbb{Z}$ *to the following congruency* $5x + 4 \equiv 28 + 2x \quad (\ mod\ 13\ )$

**Exercise 14.** *Choose the modulus* $23$ *and find all solutions* $x \in \mathbb{Z}$ *to the following congruency* $69x \equiv 5 \quad (\ mod\ 23\ )$

**Exercise 15.** *Choose the modulus* $23$ *and find all solutions* $x \in \mathbb{Z}$ *to the following congruency* $69x \equiv 46 \quad (\ mod\ 23\ )$

**The Chinese Remainder Theorem** We have seen in the previous paragraph how to solve congruencies in modular arithmetic. However one question that remains is, how to solve systems of congruencies, whith different moduli? The answer is given by the so called *Chinese raimainder theorem*, which tells us, that for any $k \in \mathbb{N}$ and coprime natural numbers

$n_1,\dots n_k \in \mathbb{N}$ as well as integers $a_1,\dots a_k \in \mathbb{Z}$, the so-called *simultaneous congruency*

$$
\begin{aligned}
x &\equiv a_1 \quad (\bmod n_1) \\
x &\equiv a_2 \quad (\bmod n_2) \\
&\cdots \\
x &\equiv a_k \quad (\bmod n_k)
\end{aligned}
\tag{4.13}
$$

has a solution and all possible solutions of this congruence system are congruent modulo the product $N = n_1 \cdot \ldots \cdot n_k$. In fact, the following algorithm computes the solution set:

**Algorithm 2** Chinese Reminder Theorem

**Require:** $n_0,\dots,n_{k-1} \in \mathbb{N}$ coprime
  **procedure** CONGRUENCY-SYSTEMS-SOLVER($k, a_{0,\dots,k-1}, n_{0,\dots,k-1}$)
    $N \leftarrow n_0 \cdot \ldots \cdot n_{k-1}$
    **while** $j < k$ **do**
      $N_j \leftarrow N/n_j$
      $(\_, s_j, t_j) \leftarrow EXT-EUCLID(N_j, n_j)$    $\triangleright\ 1 = s_j \cdot N_j + t_j \cdot n_j$
    **end while**
    $x' \leftarrow \sum_{j=0}^{k-1} a_j \cdot s_j \cdot N_j$
    $x \leftarrow x' \bmod N$
    **return** $\{x + m \cdot N \mid m \in \mathbb{Z}\}$
  **end procedure**
**Ensure:** $\{x + m \cdot N \mid m \in \mathbb{Z}\}$ is the complete solution set to 4.13.

This is the classical Chinese remainder theorem as it was already known in ancient China. Under certain circumstances, the theorem can be extended to non-coprime moduli $n_1,\dots,n_k$ but we don't need that extension in the book.

**Example 7.** *To illustrate how to solve simultaneous congruences using the Chinese remainder theorem, let's look at the following system of congruencies:*

$$
\begin{aligned}
x &\equiv 4 \quad (\ mod\ 7\ ) \\
x &\equiv 1 \quad (\ mod\ 3\ ) \\
x &\equiv 3 \quad (\ mod\ 5\ ) \\
x &\equiv 0 \quad (\ mod\ 11\ )
\end{aligned}
$$

*Clearly all moduli are coprime and we have* $N = 7 \cdot 3 \cdot 5 \cdot 11 = 1155$, *as well as* $N_1 = 165$, $N_2 = 385$, $N_3 = 231$ *and* $N_4 = 105$. *From this we calculate with the extended Euclidean algorithm*

$$
\begin{aligned}
1 &= 2 \cdot 165 + -47 \cdot 7 \\
1 &= 1 \cdot 385 + -128 \cdot 3 \\
1 &= 1 \cdot 231 + -46 \cdot 5 \\
1 &= 2 \cdot 105 + -19 \cdot 11
\end{aligned}
$$

*so we have* $x = 4 \cdot 2 \cdot 165 + 1 \cdot 1 \cdot 385 + 3 \cdot 1 \cdot 231 + 0 \cdot 2 \cdot 105 = 2398$ *as one solution. Because* $2398\ mod\ 1155 = 88$ *the set of all solutions is* $\{\dots,-2222,-1067,88,1243,2398,\dots\}$. *In particular, there are infinitely many different solutions. We can invoke sage's computation of the Chinese Remainder Theorem (CRT) to double check our findings:*

```
sage: CRT_list([4,1,3,0], [7,3,5,11])
88
```

As we have seen in various examples before, computing congruencies can be cumersome and solution sets are huge in general. It is therefore advantagous to find some kind of simplification for modular arithmetic.

Fortunately this is possible and relatively straight forward once we consider all integers that have the same remainder with respect to a given modulus $n$ in Euclidean division to be equivalent. Then we can go a step further and identify each set of numbers with equal remainder with that remainder and call it a *remainder class* or *residue class* in modulo $n$ arithmetics.

It then follows from the properties of Euclidean division, that there are exactly $n$ different remainder classes for every modulus $n$ and that integer addition and multiplication can be projected to a new kind of addition and multiplication on thoses classes.

Roughly speaking the new rules for addition and multiplication are then computed by taking any element of the firsr equivalence class and some element of the second, then add or multiply them in the usual way and see in which equivalence class the result is contained. The following example makes the abstract idea more concrete

**Example 8** (Arithmetics modulo 6)**.** *Choosing the modulus $n = 6$ we have six equivalence classes of integers which are congruent modulo 6 (which have the same remainder when divided by 6) and when we identify those remainder classes, with the remainder, we get the following identification:*

$$\begin{array}{lll} 0 := \{\ldots,-6,0,6,12,\ldots\}, & 1 := \{\ldots,-5,1,7,13,\ldots\}, & 2 := \{\ldots,-4,2,8,14,\ldots\} \\ 3 := \{\ldots,-3,3,9,15,\ldots\}, & 4 := \{\ldots,-2,4,10,16,\ldots\}, & 5 := \{\ldots,-1,5,11,17,\ldots\} \end{array}$$

*Now to compute the addition of those equivalence classes, say $2+5$, one chooses arbitrary elements from both sets say 14 and $-1$, adds those numbers in the usual way and then looks in which equivalence class the result will be.*

*So we have $14+(-1) = 13$ and 13 is in the equivalence class (of) 1. Hence we find that $2+5 = 1$ in modular 6 aithmetics, which is a more readable way to write the congruency $2+5 \equiv 1 \quad (\ mod\ 6\ )$.*

*Applying the same reasoning to all equivalence classes, addition and multiplication can be transferred to the equivalence classes and the results are summarized in the following addition and multiplication tables for modulus 6 aithmetics:*

| + | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 1 | 2 | 3 | 4 | 5 | 0 |
| 2 | 2 | 3 | 4 | 5 | 0 | 1 |
| 3 | 3 | 4 | 5 | 0 | 1 | 2 |
| 4 | 4 | 5 | 0 | 1 | 2 | 3 |
| 5 | 5 | 0 | 1 | 2 | 3 | 4 |

| · | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 0 | 2 | 4 | 0 | 2 | 4 |
| 3 | 0 | 3 | 0 | 3 | 0 | 3 |
| 4 | 0 | 4 | 2 | 0 | 4 | 2 |
| 5 | 0 | 5 | 2 | 3 | 2 | 1 |

*This way we have define a new arithmetic system, that contains just 6 numbers and that comes with its own definition of addition and multiplication. Lets symbolize it by $\mathbb{Z}_6$ and call it modular 6 arithmetics.*

*To see why such an identification of a congruency class with its remainder is useful and actually simplifies congruency computations a lot, lets go back to the congruency*

$$7\cdot(2x+21)+11 \equiv x-102 \quad (\ mod\ 6\ ) \tag{4.14}$$

*from example 6 again. As shown in that example, arithmetics of congruencies can deviate from ordinary arithmetics as for example division needs to check for coprimeness of the modulus and the divident and solutions are in general not unique.*

*The point here is, that we can rewrite this congruency into an equation over our new arithmetic type $\mathbb{Z}_6$ by "projecting onto the remainder classes". In particular, since $7 \bmod 6 = 1$, $21 \bmod 6 = 3$, $11 \bmod 6 = 5$ and $102 \bmod 6 = 0$ we have*

$$7\cdot(2x+21)+11 \equiv x-102 \pmod{6} \text{ over } \mathbb{Z}$$
$$\Leftrightarrow\ 1\cdot(2x+3)+5 = x \text{ over } \mathbb{Z}_6$$

*and we can use the multiplication and addition tables to solves the equation on the right, like we would solve normal integer equations:*

$$\begin{aligned}
1\cdot(2x+3)+5 &= x \\
2x+3+5 &= x && \text{\# addition-table: } 3+5=2 \\
2x+2 &= x && \text{\# add 4 and } -x \text{ on both sides} \\
2x+2+4-x &= x+4-x && \text{\# addition-table: } 2+4=0 \\
x &= 4
\end{aligned}$$

*So we see that, despite the somewhat unfimilar rules of addition and multiplication, solving congruencies this way is very similar to solving normal equations. And indeed the solution set is identical to the solution set of the original congruency, since 4 is identified with the set $\{4+6\cdot k \mid k\in\mathbb{Z}\}$.*

*We can invoke sage to do computations in our modular 6 arithmetics type. This is particulary useful to double-check our computations:*

```
sage: Z6 = Integers(6)
sage: Z6(2) + Z6(5)
1
sage: Z6(7)*(Z6(2)*Z6(4)+Z6(21))+Z6(11) == Z6(4) - Z6(102)
True
```

**Jargon 1** (*k*-bit modulus)**.** *In cryptographic papers, we can sometimes read phrases like "[...] using a 4096-bit modulus". This means that the underlying modulus n of the modular arithmetic used in the system has a binary representation with a length of 4096 bits. For example, the number 6 has the binary representation 110 and henxe example describes a 3-bit modulus arithmetics system.*

**Exercise 16.** *Let $a,b,k$ be integers, such that $a\equiv b \pmod{n}$ holds. Show $a^k \equiv b^k \pmod{n}$.*

**Exercise 17.** *Let $a,n$ be integers, such that a and n are not coprime. For which $b\in\mathbb{Z}$ does the congruency $a\cdot x\equiv b \pmod{n}$ have a solution x and how does the solution set look in that case?*

**Modular Inverses** As we know integers can be added, subtracted and multiplied, but not divided in general, as for example $3/2$ is not an integer anymore. To see why this is, from a more theoretical perspective, lets consider the definition of a multiplicative inverse first. When we have a set that has some kind of multiplication defined on it and we have a distinguished element of that set, that behaves neutral with respect to that multiplication (doesn't change anything when multiplied with any other element), then we can define *multiplicative inverses* in the following way:

Let $S$ be our set that has some notion $a \cdot b$ of multiplication and a *neutral element* $1 \in S$, such that $1 \cdot a = a$ for all elements $a \in S$. Then a *multiplicative inverse* $a^{-1}$ of an element $a \in S$ is defined by

$$a \cdot a^{-1} = 1 \tag{4.15}$$

So roughly speaking a multiplicative inverse is defined in such a way, that it cancels the original element to give 1, whenever they are multiplied.

Numbers that have multiplicative inverses are of particular interest, because they immediately lead to the definition of division by those numbers. In fact if $a$ is number, such that the multiplicative inverse $a^{-1}$ exist, then we define *division* by $a$ simply as multiplication by the inverse, i.e

$$\frac{b}{a} := b \cdot a^{-1} \tag{4.16}$$

**Example 9.** *Consider the set of rational numbers $\mathbb{Q}$, that is the set of all fractions. Then the neural element of multiplication is* 1*, since $1 \cdot a = a$ for all rational numbers. For example $1 \cdot 4 = 4$, $1 \cdot \frac{1}{4} = \frac{1}{4}$, or $1 \cdot 0 = 0$ and so on.*

*Then every rational number $a \neq 0$ has a multiplicative inverse, given by $\frac{1}{a}$. For example the multiplicative inverse of* 3 *is $\frac{1}{3}$, since $3 \cdot \frac{1}{3} = 1$, the multiplicative inverse of $\frac{5}{7}$ is $\frac{7}{5}$, since $\frac{5}{7} \cdot \frac{7}{5} = 1$ and so on.*

**Example 10.** *Looking at the set $\mathbb{Z}$ of integers, we see that with respect to multiplication the neutral element is the number* 1 *and we notice, that no integer $a \neq 1$ has a multiplicative inverse, since the equation $a \cdot x = 1$ has no integer solutions for $a \neq 1$.*

*The definition of multiplicative inverse works verbtim for addition, too. In the case of integers, the neutral element with respect to addition is* 0*, since $a + 0 = 0$ for all integers $a \in \mathbb{Z}$. The additive inverse then always exist and is given by the negative number $-a$, since $a + (-1) = 0$.*

**Example 11.** *Looking at the set $\mathbb{Z}_6$ of residual classes modulo* 6 *from example XXX, we can use the multiplication table to find multiplicative inverses. To see that we look at the row of the element and then find the entry equal to* 1*. If such an entry exist, the element of that column is the multiplicative inverse. If on the other hand the row has no entry equal to* 1*, we know that the element has no multiplicative inverse.*

*For example in $\mathbb{Z}_6$ the multiplicative inverse of* 5 *is* 5 *itself, since $5 \cdot 5 = 1$. We can moreover see that* 5 *and* 1 *are the only elements that have multiplicative inverses in $\mathbb{Z}_6$.*

*Now since* 5 *has a multiplicative inverse modulo* 6*, it makes sense to "divide by* 5 *in $\mathbb{Z}_6$. For example*

$$\frac{4}{5} = 4 \cdot 5^{-1} = 4 \cdot 5 = 2$$

From the last example we can make the interesting observation, that while 5 has no multiplicative inverse as an integer, it has a multiplicative inverse in modular 6 arithmetics.

So the question remains, to understand, what elements have multiplicative inverses in modular arithmetics. The answer is, that in modular $n$ arithmetics, a residue class $r$ has a multiplicative inverse, if and only if $n$ and $r$ are coprime. Since $ggt(n, r) = 1$ in that case, we know from the extended Euclidean algorithm, that there are numbers $s$ and $t$, such that

$$1 = s \cdot n + t \cdot r \tag{4.17}$$

and if we take the modulus $n$ on both sides the term $s \cdot n$ vanishes, which tells us that $t \bmod n$ is the multiplicative inverse of $r$ in modular $n$ arithmetics.

**Example 12** (Multiplicative inverses in $\mathbb{Z}_6$)**.** *In the previous example we have looked up multiplicative inverses in $\mathbb{Z}_6$ from lookup-table XXX. In real world examples, it is of course usually impossible to write down those lookup tables as the modulus is way too large and the sets occasionally contain more elements, then there are atoms in the observable universe.*

*No to see that $2 \in \mathbb{Z}_6$ has no multiplicative inverse in $\mathbb{Z}_6$ without using the lookup table, we immediately observe that $2$ and $6$ are not coprime since their greatest common divisor is $2$. If follows that equation 4.17 has no solutions $s$ and $t$ and hence $2$ has no multiplicative inverse.*

*The same reasoning works for $3$ and $4$, too as both are not coprime with $6$ and the only case that is different is $5$, since $ggt(6,5) = 1$. To compute the multiplicative inverse of $5$ we use the extended Euclidean algorithm and compute*

| $k$ | $r_k$ | $s_k$ | $t_k = (r_k - s_k \cdot a)$ *div* $b$ |
|---|---|---|---|
| 0 | 6 | 1 | 0 |
| 1 | 5 | 0 | 1 |
| 2 | 1 | 1 | -1 |
| 3 | 0 | . | . |

*So we get $s = 1$ as well as $t = -1$ and have $1 = 1 \cdot 6 - 1 \cdot 5$. From this follows that $-1$ mod $6 = 5$ is the multiplicative inverse of $5$ in modular $6$ arithmetics. We can double check using sage:*

```
sage: ZZ(6).xgcd(ZZ(5))
(1, 1, -1)
```

At this point the attentive reader might notice, that the situation, where the modulus is a prime number is of particular interest, since we know from exercise XXX, that in this cases all remainder classes must have modular inverses, since $ggt(r,n) = 1$ for prime $n$ and $r < n$. In fact Fermat's little theorem then gives a way to compute multiplicative inverses in this situation, since in case of a prime modulus $p$ and $r < p$, we have

$$\begin{aligned} r^p &\equiv r \quad (\bmod\ p) \Leftrightarrow \\ r^{p-1} &\equiv 1 \quad (\bmod\ p) \Leftrightarrow \\ r \cdot r^{p-2} &\equiv 1 \quad (\bmod\ p) \end{aligned}$$

which tells us, that the multiplicative inverse of a residue class $r$ in modular $p$ arithmetic is precisely $r^{p-2}$.

**Example 13** (Modular 5 arithmetics)**.** *To see the unique properties of mudular arithmetics whenever the modulus is prime numbers, we will parallel our findings from example XXX, but this time for the prime modulus $5$. For $n = 5$ we have five equivalence classes of integers which are congruent modulo $5$. We write*

$$\begin{aligned} &0 := \{\ldots, -5, 0, 5, 10, \ldots\}, \quad 1 := \{\ldots, -4, 1, 6, 11, \ldots\}, \quad 2 := \{\ldots, -3, 2, 7, 12, \ldots\} \\ &3 := \{\ldots, -2, 3, 8, 13, \ldots\}, \quad 4 := \{\ldots, -1, 4, 9, 14, \ldots\} \end{aligned}$$

*Addition and multiplication can be transferred to the equivalence classes, in a way exactly similar to example XXX. This results in the following addition and multiplication tables:*

| + | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 | 0 |
| 2 | 2 | 3 | 4 | 0 | 1 |
| 3 | 3 | 4 | 0 | 1 | 2 |
| 4 | 4 | 0 | 1 | 2 | 3 |

| $\cdot$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 |
| 2 | 0 | 2 | 4 | 1 | 3 |
| 3 | 0 | 3 | 1 | 4 | 2 |
| 4 | 0 | 4 | 3 | 2 | 1 |

*Calling the set of reminder classes in modular 5 arithmetics with this addition and multiplication $\mathbb{F}_5$ (for reasons we explain in more detail in XXX), we see some subtle but important differences to the situation in $\mathbb{Z}_6$. In particular we see that in the multiplication table every remainer $r \neq 0$ has the entry 1 in its row and therefore has a multiplicative inverse. In addition there are no non zero elements, such that their product is zero.*

*To use Fermat's little theorem in $\mathbb{F}_5$ for computing multiplicative inverses (instead of using the multiplcation table), lets consider $3 \in \mathbb{F}_3$. We know that the multiplicative inverse is then given by the remainder class that contains $3^{5-2} = 3^3 = 3 \cdot 3 \cdot 3 = 4 \cdot 3 = 2$. And indeed $3^{-1} = 2$, since $3 \cdot 2 = 1$ in $\mathbb{F}_5$.*

*We can invoke sage to do computations in our modular 5 arithmetics type. This is particularly useful to double-check our computations:*

```
sage: Z5 = Integers(5)
sage: Z5(3)**(5-2)
2
sage: Z5(3)**(-1)
2
sage: Z5(3)**(5-2) == Z5(3)**(-1)
True
```

**Example 14.** *To understand one of the principle difference in prime number modular arithmetics vs. other number modular arithmetics, consider the linear equation $a \cdot x + b = 0$ defined over both types $\mathbb{F}_5$ and $\mathbb{Z}_6$. Since in $\mathbb{F}_5$ every non zero element has a multiplicative inverse, we can always solves equations like this, which is not true in $\mathbb{Z}_6$. To see that consider the equation $3x + 3 = 0$. In $\mathbb{F}_5$ we have*

$$
\begin{aligned}
3x+3 &= 0 && \text{\# add 2 and on both sides} \\
3x+3+2 &= 2 && \text{\# addition-table: } 2+3=0 \\
3x &= 2 && \text{\# divide by } 3 \\
2\cdot(3x) &= 2\cdot 2 && \text{\# multiplication-table: } 2+2=4 \\
x &= 4
\end{aligned}
$$

*So in the case of our prime number modular arithmetics, we get the unique solution $x = 4$. Now consider $\mathbb{Z}_6$. In this case*

$$
\begin{aligned}
3x+3 &= 0 && \text{\# add 3 and on both sides} \\
3x+3+3 &= 3 && \text{\# addition-table: } 3+3=0 \\
3x &= 3 && \text{\# no multiplicative inverse of 3 exists}
\end{aligned}
$$

*So in this case, we can not solve the equation for $x$, by dividing by 3. And indeed we use the multiplication table of $\mathbb{Z}_6$, we find that there are three solutions $x \in \{1,3,5\}$, such that $3x+3 = 0$ holds true for all of them.*

**Exercise 18.** *Consider the modulus $n = 24$. Which of the integers 7, 1, 0, 805, $-4255$ have multiplicative inverses in modular 24 arithmetics? Compute the inverses, in case they exist.*

**Exercise 19.** *Find the set of all solutions to the congruency $17(2x+5)-4 \equiv 2x+4 \pmod 5$. Then project the congruency into $\mathbb{F}_5$ and solve the resulting equation in $\mathbb{F}_5$. Compare the results.*

**Exercise 20.** *Find the set of all solutions to the congruency $17(2x+5)-4 \equiv 2x+4 \pmod 6$. Then project the congruency into $\mathbb{Z}_6$ and try to solve the resulting equation in $\mathbb{Z}_6$.*

## 4.3 Polynomial Arithmetics

A polynomial is an expression consisting of variables (also called indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponentiation of variables. All coefficients of a polynomial must have the same type, e.g. being integers or fractions etc. To be more precise a *univariate polynomial* is an expression

$$P(x) := \sum_{j=0}^{m} a_j t^j = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0 \ , \tag{4.18}$$

where $x$ is called the *indeterminate*, each $a_j$ is called a *coefficient*. If $R$ is the type of the coefficients then the set of all **univariate polynomials with coefficients in** $R$ is written as $R[x]$. We often simply *polynomial* instead of univariate polynomial, write $P(x) \in R[x]$ for a polynomial and denote the constant term as $P(0)$.

A polynomial is called the *zero polynomial* if all coefficients are zero and a polynomial is called the *one polynomial* if the constant term is 1 and all other coefficients are zero.

If an univariate polynomial $P(x) = \sum_{j=0}^{m} a_j x^j$ is given, that is not the zero polynomial, we call

$$deg(P) := m \tag{4.19}$$

the *degree* of $P$ and define the degree of the zero polynomial to be $-\infty$, where $-\infty$ (negative infinity) is a symbol with the property that $-\infty + m = -\infty$ for all counting numbers $m \in \mathbb{N}$. In addition we write

$$Lc(P) := a_m \tag{4.20}$$

and call it the *leading coefficient* of the polynomial $P$. We can restrict the set $R[x]$ of *all* polynomials with coefficients in $R$, to the set of all such polynomials that have a degree that does not exceed a certain value. If $m$ is the maximum degree allowed, we write $R_{\leq m}[x]$ for the set of all ploynomials with a degree less or equal to $m$.

**Example 15** (Integer Polynomials). *The coefficients of a polynomial must all have the same type. The set of polynomials with integer coefficients is written as $\mathbb{Z}[x]$. Examples of such polynomials are:*

$$
\begin{aligned}
P_1(x) &= 2x^2 - 4x + 17 && \#\ \textit{with } deg(P_1) = 2 \textit{ and } Lc(P_1) = 2 \\
P_2(x) &= x^{23} && \#\ \textit{with } deg(P_2) = 23 \textit{ and } Lc(P_2) = 1 \\
P_3(x) &= x && \#\ \textit{with } deg(P_3) = 1 \textit{ and } Lc(P_3) = 1 \\
P_4(x) &= 174 && \#\ \textit{with } deg(P_4) = 0 \textit{ and } Lc(P_4) = 174 \\
P_5(x) &= 1 && \#\ \textit{with } deg(P_5) = 0 \textit{ and } Lc(P_5) = 1 \\
P_6(x) &= 0 && \#\ \textit{with } deg(P_5) = -\infty \textit{ and } Lc(P_6) = 0 \\
P_7(x) &= (x-2)(x+3)(x-5)
\end{aligned}
$$

*In particular every integer can be seen as an integer polynomial of degree zero. $P_7$ is a polynomial, because we can expand its definition into $P_7(x) = x^3 - 4x^2 - 11x + 30$, which is polynomial of degree 3 and leading coefficient 1. The following expressions are not integer polynomial*

$$
\begin{aligned}
Q_1(x) &= 2x^2 + 4 + 3x^{-2} \\
Q_2(x) &= 0.5x^4 - 2x \\
Q_3(x) &= 1/x
\end{aligned}
$$

*We can invoke sage to do computations with polynomials. To do so we have to specify the symbol for the indertiminate and the type for the coefficients. Note however that sage defines the degree of the zero polynomial to be* $-1$.

```
sage: Zx = ZZ['x'] # integer polynomials with indeterminate x
sage: Zt.<t> = ZZ[] # integer polynomials with indeterminate t
sage: Zx
Univariate Polynomial Ring in x over Integer Ring
sage: Zt
Univariate Polynomial Ring in t over Integer Ring
sage: p1 = Zx([17,-4,2])
sage: p1
2*x^2 - 4*x + 17
sage: p1.degree()
2
sage: p1.leading_coefficient()
2
sage: p2 = Zt(t^23)
sage: p2
t^23
sage: p6 = Zx([0])
sage: p6.degree()
-1
```

**Example 16** (Polynomials over $\mathbb{Z}_6$)**.** *Recall our definition of the residue classes* $\mathbb{Z}_6$ *and their arithmetics as defined in* **??**. *The set of all polynomials with indeterminate* $x$ *and coefficients in* $\mathbb{Z}_6$ *is symbolized as* $\mathbb{Z}_6[x]$. *Example of polynomials from* $\mathbb{Z}_6$ *are:*

$$\begin{aligned}
P_1(x) &= 2x^2 - 4x + 5 && \#\, \text{with } deg(P_1) = 2 \text{ and } Lc(P_1) = 2 \\
P_2(x) &= x^{23} && \#\, \text{with } deg(P_2) = 23 \text{ and } Lc(P_2) = 1 \\
P_3(x) &= x && \#\, \text{with } deg(P_3) = 1 \text{ and } Lc(P_3) = 1 \\
P_4(x) &= 3 && \#\, \text{with } deg(P_4) = 0 \text{ and } Lc(P_4) = 3 \\
P_5(x) &= 1 && \#\, \text{with } deg(P_5) = 0 \text{ and } Lc(P_5) = 1 \\
P_6(x) &= 0 && \#\, \text{with } deg(P_5) = -\infty \text{ and } Lc(P_6) = 0 \\
P_7(x) &= (x-2)(x+3)(x-5)
\end{aligned}$$

*As in the previous example* $P_7$ *is a polynomial. However since we are working with coefficients from* $\mathbb{Z}_6$ *now the expension of* $P_7$ *is computed differently, as we have to invoke addition and multiplication in* $\mathbb{Z}_6$ *as defined in XXX. We get:*

$$\begin{aligned}
(x-2)(x+3)(x-5) &= (x+4)(x+3)(x+1) && \#\, \text{additive inverses in } \mathbb{Z}_6 \\
&= (x^2 + 4x + 3x + 3 \cdot 4)(x+1) && \#\, \text{bracket expansion} \\
&= (x^2 + 1x + 0)(x+1) && \#\, \text{compuation in } \mathbb{Z}_6 \\
&= (x^3 + x^2 + x^2 + x) && \#\, \text{bracket expansion} \\
&= (x^3 + 2x^2 + x)
\end{aligned}$$

*We can invoke sage to do computations with polynomials, that have their coefficients in* $\mathbb{Z}_6$. *To do so we have to specify the symbol for the indertiminate and the type for the coefficients:*

```
sage: Z6 = Integers(6)
sage: Z6x = Z6['x']
sage: Z6x
Univariate Polynomial Ring in x over Ring of integers modulo 6
sage: p1 = Z6x([5,-4,2])
sage: p1
2*x^2 + 2*x + 5
sage: p1 = Z6x([17,-4,2])
sage: p1
2*x^2 + 2*x + 5
sage: Z6x(x-2)*Z6x(x+3)*Z6x(x-5) == Z6x(x^3 + 2*x^2 + x)
True
```

Given some element from the same type as the coefficients of a polynomial, the polynomial can be evaluated at that element, which means that we insert the given element for every ocurrence of the indeterminate $x$ in the polynomial expression.

To be more precise let $P \in R[x]$, with $P(x) = \sum_{j=0}^{m} a_j x^j$ be a polynomial with coefficient of type $R$ and let $b \in R$ be an element of that type. Then the *evaluation* of $P$ at $b$ is given by

$$P(a) = \sum_{j=0}^{m} a_j b^j \tag{4.21}$$

**Example 17.** *Consider the integer polynomials from example XXX again. To evaluate them at given points, we have to insert the point for all occurences of x in the polynomial expression. Inserting arbitrary values from $\mathbb{Z}$, we get:*

$$\begin{aligned}
P_1(2) &= 2\cdot 2^2 - 4\cdot 2 + 17 = 17\\
P_2(3) &= 3^{23} = 94143178827\\
P_3(-4) &= -4 = -4\\
P_4(15) &= 174\\
P_5(0) &= 1\\
P_6(1274) &= 0\\
P_7(-6) &= (-6-2)(-6+3)(-6+5) = -264
\end{aligned}$$

*Note however that is not possible to evaluate any of those polynomial on values of different type. It is for example strictly speaking wrong to write $P_1(0.5)$, since $0.5$ is not an integer. We can verify our computations using sage:*

```
sage: Zx = ZZ['x']
sage: p1 = Zx([17,-4,2])
sage: p7 = Zx(x-2)*Zx(x+3)*Zx(x-5)
sage: p1(ZZ(2))
17
sage: p7(ZZ(-6)) == ZZ(-264)
True
```

**Example 18.** *Consider the polynomials with coefficients in $\mathbb{Z}_6$ from example XXX again. To evaluate them at given values from $\mathbb{Z}_6$, we have to insert the point for all occurences of x in the polynomial expression. We get:*

$$
\begin{aligned}
P_1(2) &= 2\cdot 2^2 - 4\cdot 2 + 5 = 2 - 2 + 5 = 5\\
P_2(3) &= 3^{23} = 3\\
P_3(-4) &= P_3(2) = 2\\
P_5(0) &= 1\\
P_6(4) &= 0
\end{aligned}
$$

```
sage: Z6 = Integers(6)
sage: Z6x = Z6['x']
sage: p1 = Z6x([5,-4,2])
sage: p1(Z6(2)) == Z6(5)
True
```

**Exercise 21.** *Compare both expansions of $P_7$ from $\mathbb{Z}[x]$ and from $\mathbb{Z}_6[x]$ in example XXX and example XXX and consider the definition of $\mathbb{Z}_6$ as given in example XXX. Can you see how the definition of $P_7$ over $\mathbb{Z}$ projects to the definition over $\mathbb{Z}_6$ if you consider the residue classes of $\mathbb{Z}_6$?*

**Polynomial Arithmetics** Polynomials behave like integers in many ways. In particular they can be added, subtracted and multiplied. In addition they have their own notion of Euclidean division. Roughly speaking two polynomials are added by simply adding the coefficients of the same index and they are multiplied by applying the distributive property, that is by multiplying every term of the left factor with every term of the right factor and add the results together.

To be more precise let $\sum_{n=0}^{m_1} a_n x^n$ and $\sum_{n=0}^{m_2} b_n x$ be two polynomials from $R[x]$. Then the *sum* and the *product* of these polynomials is defined as:

$$\sum_{n=0}^{m_1} a_n x^n + \sum_{n=0}^{m_2} b_n x^n = \sum_{n=0}^{max(\{m_1,m_2\})} (a_n + b_n) x^n \tag{4.22}$$

$$\left(\sum_{n=0}^{m_1} a_n x^n\right)\cdot\left(\sum_{n=0}^{m_2} b_n x^n\right) = \sum_{n=0}^{m_1+m_2} \sum_{i=0}^{n} a_i b_{n-i} x^n \tag{4.23}$$

A rule for polynomial subtraction can be deduced from these two rules by first multiplying the subtrahend with (the polynomial) $-1$ and then add the result to the minuend.

Regarding ower definition of the degree of a polynomial, we see that the degree of the sum is always the maximum of the degrees of both summands and the degree of the product is always the degree of the factors, since we defined $-\infty\cdot m = \infty$ for every integer $m \in \mathbb{Z}$. Using sage's definition of degree, this would not hold, as the zero polynomials degree is $-1$ is sage, which would violate this rule.

**Example 19.** *To given an example of how polynomial arithmetics work, consider the following two integer polynomials $P, Q \in \mathbb{Z}[x]$ with $P(x) = 5x^2 - 4x + 2$ and $Q(x) = x^3 - 2x^2 + 5$. The* *sum of these two polynomials is computed by adding the coefficients of each term with equal exponent in x. This gives*

$$\begin{aligned}(P+Q)(x) &= (0+1)x^3+(5-2)x^2+(-4+0)x+(2+5)\\ &= x^3+3x^2-4x+7\end{aligned}$$

*The product of thess two polynomials is computed by multiplication of each term in the first factor with each term in the second factor. We get*

$$\begin{aligned}(P\cdot Q)(x) &= (5x^2-4x+2)\cdot(x^3-2x^2+5)\\ &= (5x^5-10x^4+25x^2)+(-4x^4+8x^3-20x)+(2x^3-4x^2+10)\\ &= 5x^5-14x^4+10x^3+21x^2-20x+10\end{aligned}$$

```
sage: Zx = ZZ['x']
sage: P = Zx([2,-4,5])
sage: Q = Zx([5,0,-2,1])
sage: P+Q == Zx(x^3 +3*x^2 -4*x +7)
True
sage: P*Q == Zx(5*x^5 -14*x^4 +10*x^3+21*x^2-20*x +10)
True
```

**Example 20.** *Lets consider the polynomials of the previous example but interpreted in modular 6 aithmetics. So we consider $P,Q\in\mathbb{Z}_6[x]$ again with $P(x)=5x^2-4x+2$ and $Q(x)=x^3-2x^2+5$. This time we get*

$$\begin{aligned}(P+Q)(x) &= (0+1)x^3+(5-2)x^2+(-4+0)x+(2+5)\\ &= (0+1)x^3+(5+4)x^2+(2+0)x+(2+5)\\ &= x^3+3x^2+2x+1\end{aligned}$$

$$\begin{aligned}(P\cdot Q)(x) &= (5x^2-4x+2)\cdot(x^3-2x^2+5)\\ &= (5x^2+2x+2)\cdot(x^3+4x^2+5)\\ &= (5x^5+2x^4+1x^2)+(2x^4+2x^3+4x)+(2x^3+2x^2+4)\\ &= 5x^5+4x^4+4x^3+3x^2+4x+4\end{aligned}$$

```
sage: Z6x = Integers(6)['x']
sage: P = Z6x([2,-4,5])
sage: Q = Z6x([5,0,-2,1])
sage: P+Q == Z6x(x^3 +3*x^2 +2*x +1)
True
sage: P*Q == Z6x(5*x^5 +4*x^4 +4*x^3+3*x^2+4*x +4)
True
```

**Exercise 22.** *Compare the sum $P+Q$ and the product $P\cdot Q$ from the previous two example XXX and XXX and consider the definition of $\mathbb{Z}_6$ as given in example XXX. How can we derive the computations in $\mathbb{Z}_6[x]$ from the computations in $Z[x]$?*

**Euklidean Division** The ring of polynomials shares a lot of properties with the integers. In particular there is also the concept of Euclidean division and the algorithm of long division defined for polynomials. Recalling from Euclidean division of integers XXX, we know, that given two integers $a$ and $b \neq 0$ there is always another integer $m$ and a counting number $r$ with $r < |b|$, such that $a = m \cdot b + r$ holds.

We can generalize this to polynomials, whenever the leading coefficient of the dividend polynomial has a notion of multiplicative inverse. In fact given two polynomials $A$ and $B \neq 0$ from $R[x]$, such that $Lc(B)^{-1}$ exists in $R$, there exist two polynomials $M$ (the quotient) and $R$ (the remainder), such that

$$A = M \cdot B + R \tag{4.24}$$

and $deg(R) < deg(B)$. Similar to integer Euclidean division both $M$ and $R$ are uniquely defined by these relations.

**Notation and Symbols 2.** *Suppose that the polynomials $A, B, M$ and $R$ satisfy equation XX. Then we often write*

$$A\ div\ B := M, \qquad A\ mod\ B := R \tag{4.25}$$

*to describe the quotient and the remainder polynomials of the Euclidean division. We also say, that a polynomial $A$ is divisible by another polynomial $B$ if $A\ mod\ B = 0$ holds. In this case we also write $B|A$ and call $B$ a factor of $A$.*

Analog to integers, methods to compute Euclidean division for polynomials are called *polynomial division algorithms*. Probably the best known algorithm is the so called *polynomial long division*.

**Algorithm 3** Polynomial Euclidean Algorithm

**Require:** $A, B \in R[x]$ with $B \neq 0$, such that $Lc(B)^{-1}$ exists in $R$
 **procedure** POLY-LONG-DIVISION($A, B$)
  $M \leftarrow 0$
  $R \leftarrow A$
  $d \leftarrow deg(B)$
  $c \leftarrow Lc(B)$
  **while** $deg(R) \geq d$ **do**
   $S := Lc(R) \cdot c^{-1} \cdot x^{deg(R)-d}$
   $M \leftarrow M + S$
   $R \leftarrow R - S \cdot B$
  **end while**
  **return** $(Q, R)$
 **end procedure**
**Ensure:** $A = M \cdot B + R$

This algorithm works only when there is a notion of division by the leading coefficient of $B$. It can be generalized, but we will only need this somewhat simpler method in what follows.

**Example 21** (Polynomial Long Division)**.** *To give an example of how the previous algorithm works, lets divide the integer polynomial $A(x) = x^5 + 2x^3 - 9 \in \mathbb{Z}[x]$ by the integer polynomial $B(x) = x^2 + 4x - 1 \in \mathbb{Z}[x]$. Since $B$ is not the zero polynomial and the leading coefficient of $B$ is $1$, which is invertible as an integer, we can applay algorithm XXX. Our goal is to find solutions to equation XXX, that is we need to find the quotient polynomial $M \in \mathbb{Z}[x]$ and the reminder* *polynomial $R \in \mathbb{Z}[x]$ such that $x^5 + 2x^3 - 9 = M(x) \cdot (x^2 + 4x - 1) + R$. Using a notation that is mostly used in Commonwealth countries, we compute*

$$
\begin{array}{rrrrrrr}
 & & & X^3 & -4X^2 & +19X & -80 \\ \cline{2-7}
X^2+4X-1 \big) & X^5 & & +2X^3 & & & -9 \\
 & -X^5 & -4X^4 & +X^3 & & & \\ \cline{2-4}
 & & -4X^4 & +3X^3 & & & \\
 & & 4X^4 & +16X^3 & -4X^2 & & \\ \cline{3-5}
 & & & 19X^3 & -4X^2 & & \\
 & & & -19X^3 & -76X^2 & +19X & \\ \cline{4-6}
 & & & & -80X^2 & +19X & -9 \\
 & & & & 80X^2 & +320X & -80 \\ \cline{5-7}
 & & & & & 339X & -89
\end{array} \tag{4.26}
$$

*We therefore get $M(x) = x^3 - 4x^2 + 19x - 80$ as well as $R(x) = 339x - 89$ and indeed we have $x^5 + 2x^3 - 9 = (x^3 - 4x^2 + 19x - 80) \cdot (x^2 + 4x - 1) + (339x - 89)$, which we can double check invoking sage:*

```
sage: Zx = ZZ['x']
sage: A = Zx([-9,0,0,2,0,1])
sage: B = Zx([-1,4,1])
sage: M = Zx([-80,19,-4,1])
sage: R = Zx([-89,339])
sage: A == M*B + R
True
```

**Example 22.** *In the previous example polynomial division gave a non trivial (non vanishing, i.e non-zero) remainder. Of special interest are divisions that don't give a remainder. Such divisors are called factors of the dividend.*

*For example consider the integer polynomial $P_7$ from example XXX again. As we have shown, it can be written both as $x^3 - 4x^2 - 11x + 30$ as well as $(x-2)(x+3)(x-5)$. From this we can see that the polynomials $F_1(x) = (x-2)$, $F_2(x) = (x+3)$ and $F_3(x) = (x-5)$ are all factors of $x^3 - 4x^2 - 11x + 30$, since division of $P_7$ by any of these factors will result in a zero remainder.*

**Exercise 23.** *Consider the polynomial expressions $P(x) := -3x^4 + 4x^3 + 2x^2 + 4$ and $Q(x) = x^2 - 4x + 2$. Compute the Euklidean division of $P$ by $Q$ in the following types*

1. $P, Q \in \mathbb{Z}[x]$
2. $P, Q \in \mathbb{Z}_6[x]$
3. $P, Q \in \mathbb{Z}_5[x]$

*Then consider the result in $\mathbb{Z}[x]$ and in $\mathbb{Z}_6[x]$. How can compute the result in $\mathbb{Z}_6[x]$ from the result in $\mathbb{Z}[x]$?*

**Exercise 24.** *Show that the polynomial $P(x) = 2x^4 - 3x + 4 \in \mathbb{Z}_5[x]$ is a factor of the polynomial $Q(x) = x^7 + 4x^6 + 4x^5 + x^3 + 2x^2 + 2x + 3 \in \mathbb{Z}_5[x]$, thatvis show $P|Q$. What is $Q$ div $P$?*

**Prime Factors** Recall that the fundamental theorem of arithmetics XXX tells us, that every number is the product of prime numbers. Something similar holds for polynomials, too.

The polynomial analog to a prime number is a so called *irreducible polynomial*, which is defined as a polynomial that cannot be factored into the product of two non-constant polynomials using Euclidean division. Irreducible polynomials are for polynomials what prime numbers are for integers. They are the basic building blocks from which all other polynomials can be constructed. To be more precise, let $P \in R[x]$ be any polynomial. Then there are always irreducible polynomials $F_1, F_2, \ldots, F_k \in R[x]$, such that

$$P = F_1 \cdot F_2 \cdot \ldots \cdot F_k \,. \tag{4.27}$$

This representation is unique, except for permutations in the factors and is called the **prime factorization** of $P$.

**Example 23.** *Consider the polynomial expression $P = x^2 - 3$. When we interpret $P$ as an integer polynomial $P \in \mathbb{Z}[x]$, we find that this polynomial is irreducible, since any factorization other then $1 \cdot (x^2 - 3)$, must look like $(x-a)(x+a)$ for some integer $a$, but there is no integers $a$ with $a^2 = 3$.*

```
sage: Zx = ZZ['x']
sage: p = Zx(x^2-3)
sage: p.roots()
[]
sage: p.factor()
x^2 - 3
```

*On the other hand interpreting $P$ as a polynomial $P \in \mathbb{Z}_6[x]$ in modulo 6 arithmetics, we see that $P$ has two factors $F_1 = (x-3)$ and $F_2 = (x+3)$, since $(x-3)(x+3) = x^2 - 3x + 3 - 3 \cdot 3 = x^2 - 3$.*

Finding prime factors of a polynomial is hard. As we have seen in example XXX, points where a polynomial evaluates to zero, i.e points $x_0 \in R$ with $P(x_0) = 0$ are of special interest, since it can be shown the polynomial $F(x) = (x - x_0)$ is always a factor of $P$. The converse however is not necessarily true, because a polynomial can have irreducible prime factors.

Points where a polynomial evaluates to zero are called the **roots** of the polynomial. To be more precise, let $P \in R[x]$ be a polynomial. Then the set of all roots of $P$ is defined as

$$R_0(P) := \{x_0 \in R \mid P(x_0) = 0\} \tag{4.28}$$

Finding the roots of a polynomial is sometimes called solving the polynomial. It is a hard problem and has been the subject of much research throughout history. In fact it is well known that for polynomials of degree 5 or higher there is, in general, no closed expression, from which the roots can be deduced.

It can be shown, that if $m$ is the degree of a polynomial $P$, then $P$ can not have more than $m$ roots. However in general polynomials can have less then $m$ roots.

**Example 24.** *Consider our integer polynomial $P_7(x) = x^3 - 4x^2 - 11x + 30$ from example XXX again. We know that it's set of roots is given by $R_0(P_7) = \{-3, 2, 5\}$.*

*On the other hand we know from example XXX, that the integer polynomial $x^2 - 3$ is irreducible. It follows that it has no roots, since every root defines a prime factor.*

**Example 25.** *To give another example consider the integer polynomial $P = x^7 + 3x^6 + 3x^5 + x^4 - x^3 - 3x^2 - 3x - 1$. We can invoke sage to compute the roots and prime factors of $P$:*

```
sage: Zx = ZZ['x']
sage: p = Zx(x^7 + 3*x^6 + 3*x^5 + x^4 - x^3 - 3*x^2 - 3*x - 1
   )
sage: p.roots()
[(1, 1), (-1, 4)]
sage: p.factor()
(x - 1) * (x + 1)^4 * (x^2 + 1)
```

*We see that $P$ has the root $1$ and that the associated prime factor $(x-1)$ occurs once in $P$ and that it moreover has the root $-1$, where the associated prime factor $(x+1)$ occurs $4$ times in $P$. This gives the prime factorization*

$$P = (x-1)(x+1)^4(x^2+1)$$

**Lange interpolation** One particularly nice property of polynomials is that a polynomial of degree $m$ is completely determined on $m+1$ evaluation points. Seeing this from a different angle, we can (sometimes) uniquely derive a polynomial of degree $m$ from a set

$$S = \{(x_0,y_0),(x_1,y_1),\ldots,(x_m,y_m) \mid x_i \neq x_j \text{ for all indices i and j}\} \tag{4.29}$$

This "few too many" property of polynomials is used in many places, like for example in erasure codes. It is also of importance in snarks and we therefore need to understand a method to actually compute a polynomial from a set of points.

If the coefficients of the polynomial we want to find have a notion of multiplicative inverse, it is always possible to find such a polynomial and one method is called *Lagrange interpolation*. It works as follows: Give a set like 4.29, a polynomial $P$ of degree $m+1$ with $P(x_i) = y_i$ for all pairs $(x_i, y_i)$ from $S$ is given by the following algorithm:

**Algorithm 4** Lagrange Interpolation

**Require:** $R$ must have multiplicative inverses
**Require:** $S = \{(x_0,y_0),(x_1,y_1),\ldots,(x_m,y_m) \mid x_i,y_i \in R, x_i \neq x_j \text{ for all indices i and j}\}$
  **procedure** LAGRANGE-INTERPOLATION($S$)
    **for** $j \in (0\ldots m)$ **do**
      $l_j(x) \leftarrow \Pi_{i=0;i\neq j}^{m} \frac{x-x_i}{x_j-x_i} = \frac{(x-x_0)}{(x_j-x_0)} \cdots \frac{(x-x_{j-1})}{(x_j-x_{j-1})} \frac{(x-x_{j+1})}{(x_j-x_{j+1})} \cdots \frac{(x-x_m)}{(x_j-x_m)}$
    **end for**
    $P \leftarrow \sum_{j=0}^{m} y_j \cdot l_j$
    **return** $P$
  **end procedure**
**Ensure:** $P \in R[x]$ with $deg(P) = m$
**Ensure:** $P(x_j) = y_j$ for all pairs $(x_j, y_j) \in S$

**Example 26.** *Lets consider the set $S = \{(0,4),(-2,1),(2,3)\}$ and our task is to compute a polynomial of degree $2$ in $\mathbb{Q}[x]$ with fractional number coefficients. Since $\mathbb{Q}$ has multiplicative inverses, we can use the Lagrange interpolation algorithm from XXX, to compute the polyno-*

*mial. We get*

$$
\begin{aligned}
l_0(x) &= \frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2} = \frac{x+2}{0+2}\cdot\frac{x-2}{0-2} = -\frac{(x+2)(x-2)}{4} \\
&= -\frac{1}{4}(x^2-4) \\
l_1(x) &= \frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2} = \frac{x-0}{-2-0}\cdot\frac{x-2}{-2-2} = \frac{x(x-2)}{8} \\
&= \frac{1}{8}(x^2-2x) \\
l_2(x) &= \frac{x-x_0}{x_2-x_0}\cdot\frac{x-x_1}{x_2-x_1} = \frac{x-0}{2-0}\cdot\frac{x+2}{2+2} = \frac{x(x+2)}{8} \\
&= \frac{1}{8}(x^2+2x) \\
P(x) &= 4\cdot(-\frac{1}{4}(x^2-4)) + 1\cdot\frac{1}{8}(x^2-2x) + 3\cdot\frac{1}{8}(x^2+2x) \\
&= -x^2+4+\frac{1}{8}x^2-\frac{1}{4}x+\frac{3}{8}x^2+\frac{3}{4}x \\
&= -\frac{1}{2}x^2+\frac{1}{2}x+4
\end{aligned}
$$

*And indeed evaluation of $P$ on the x-values of $S$ gives the correct points, since $P(0)=4$, $P(-2)=1$ and $P(2)=3$.*

**Example 27.** *To give another example, more relevant to the topics of this book, lets consider the same set $S=\{(0,4),(-2,1),(2,3)\}$ as in the previous example. But this times the task is to compute a polynomial $P\in\mathbb{F}_5[x]$ from this data. Since we know that multiplicative inverses exist in $\mathbb{Z}_5$, algorithm XXX applies and we can compute a unique polynomial of degree 2 in $\mathbb{Z}_5[x]$ from S. We can use the lookup tables XXX for computation in $\mathbb{Z}_5$ and get*

$$
\begin{aligned}
l_0(x) &= \frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2} = \frac{x+2}{0+2}\cdot\frac{x-2}{0-2} = \frac{(x+2)(x-2)}{-4} = \frac{(x+2)(x+3)}{1} \\
&= x^2+1 \\
l_1(x) &= \frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2} = \frac{x-0}{-2-0}\cdot\frac{x-2}{-2-2} = \frac{x}{3}\cdot\frac{x+3}{1} = 2(x^2+3x) \\
&= 2x^2+x \\
l_2(x) &= \frac{x-x_0}{x_2-x_0}\cdot\frac{x-x_1}{x_2-x_1} = \frac{x-0}{2-0}\cdot\frac{x+2}{2+2} = \frac{x(x+2)}{3} = 2(x^2+2x) \\
&= 2x^2+4x \\
P(x) &= 4\cdot(x^2+1)+1\cdot(2x^2+x)+3\cdot(2x^2+4x) \\
&= 4x^2+4+2x^2+x+x^2+2x \\
&= 2x^2+3x+4
\end{aligned}
$$

*And indeed evaluation of $P$ on the x-values of $S$ gives the correct points, since $P(0)=4$, $P(-2)=1$ and $P(2)=3$.*

**Exercise 25.** *Consider example XXX and example XXX again. Why is it not possible to apply algorithm XXX if we consider S as a set of integers, nor as a set in $\mathbb{Z}_6$?*

# 5 Algebra

Todo: Def Subgroup, Fundamental theorem of cyclic groups.

We gave an introduction to the basic computational skills needed for a pen & paper approach to SNARKS in the previous chapter. In this chapter we get a bit more abstract and clarify a lot of mathematical terminology and jargon.

When you read papers about cryptography or mathematical papers in general, you will frequently stumble across algebraic terms like *groups*, *fields*,*rings* and similar. To understand what is going on, it is necessary to get at least some understanding of these terms. In this chapter we therefore with a short introduction to those terms.

In a nutshell, algebraic types like groups or fields define sets that are analog to numbers to various extend, in the sense that you can add, subtract, multiply or divide on those sets.

We know many example of sets that fall under those categories, like the natural numbers, the integers, the rational or the real numbers. they are in some sense already the most fundamental examples.

## 5.1 Groups

Groups are abstractions that capture the essence of mathematical phenomena, like addition and subtraction, multiplication and division, permutations, or symmetries.

To understand groups, remember back in school when we learned about addition and subtraction of integers (Forgetting about integer multiplication for a moment). We learned that we can always add two integers and that the result is guaranteed to be an integer again. We also learned how to deal with brackets, that nothing happens, when we add zero to any integer, that it doesn't matter in which order we add a given set of integers and that for every integer there is always another integer (the negative), such that when we add both together we get zero.

These conditions are the defining properties of a group and mathematicians have recognized that the exact same set of rules can be found in very different mathematical structures. It therefore makes sense to give a formulation of what a group should be, detached from any concrete example. This allows one to 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 definition of a group:

A **group** $(\mathbb{G}, \cdot)$ is a set $\mathbb{G}$, together with a map $\cdot : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, called the group law, such that the following properties hold:

- (Existence of a neutral element) There is a $e \in \mathbb{G}$ for all $g \in \mathbb{G}$, such that $e \cdot g = g$ as well as $g \cdot e = g$.
- (Existence of an inverse) For every $g \in \mathbb{G}$ there is a $g^{-1} \in \mathbb{G}$, such that $g \cdot g^{-1} = e$ as well as $g^{-1} \cdot g = e$.
- (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.

Rephrasing the abstract definition in more layman's terms, a group is something, where we can do computations that resembles the behaviour of addition of integers. Therefore when the reader reads the term group they are advised to think of something where can combine some element with another element into a new element in a way that is reversible and where the order of combining many elements doesn't matter.

**Notation and Symbols 3.** *Let $(\mathbb{G}\cdot)$ be a finite group. If there is no risk of ambiguously we frequently drop the symbol $\cdot$ and simply write $\mathbb{G}$ as a notation for the group keeping the group law implicit.*

As we will see in what follows, groups are all over the place in cryptography and in SNARKS. In particular we will see in XXX, that the set of points on an elliptic curve define a group, which is the most important example in this book. To give some more familiar examples first:

**Example 28** (Integer Addition and Subtraction)**.** *The set $(\mathbb{Z},+)$ of integers together with integer addition is the archetypical example of a group, where the group law is traditionally written as $+$ (instead of $\cdot$). To compare integer addition against the abstract axioms of a group, we first see that the neutral element $e$ is the number $0$, since $a+0=a$ for all integers $a\in$ and that the inverse of a number is the negative, since $a+(-a)=0$, for all $a\in\mathbb{Z}$. In addition we know that $(a+b)+c=a+(b+c)$, so integers with addition are indeed a group in the abstract sense.*

**Example 29** (The trivial group)**.** *The most basic example of a group, is group with just one element $\{\bullet\}$ and the group law $\bullet\cdot\bullet=\bullet$.*

**Commutative Groups** When we look at the general definition of a group we see that it is somewhat different from what we know from integers. For integers we know, that it doesn't matter in which order we add two integers, as for example $4+2$ is the same as $2+4$. However we also know from example XXX, that this is not always the case in groups.

To capture the special case of a group where the order in which the group law is executed doesn't matter, the concept of so called a **commutative group** is introduced. To be more precise a group is called commutative if $g_1\cdot g_2=g_2\cdot g_1$ holds for all $g_1,g_2\in\mathbb{G}$.

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

**Example 30.** *Consider the group of integers with integer addition again. Since $a+b=b+a$ for all integers, this group is the archetypical example of a commutative group. Since there are infinite many integers, $(\mathbb{Z},+)$ is not a finite group.*

**Example 31.** *Consider our definition of modulo $6$ residue classes $(\mathbb{Z}_6,+)$ as defined in the addition table from example XXX. As we see the residue class $0$ is the neutral element in modulo $6$ arithmetics and the inverse of a residue class $r$ is given by $6-r$, since $r+(6-r)=6$, which is congruent to $0$, since $6 \bmod 6=0$. Moreover $(r_1+r_2)+r_3=r_1+(r_2+r_3)$ is inherited from integer arithmetic.*

*We therefore see that $(\mathbb{Z}_6,+)$ is a group and since addition table XX is symmetric, we see $r_1+r_2=r_2+r_1$ which shows that $(\mathbb{Z}_6,+)$ is commutative.*

The previous example provided us with an important example of commutative groups that are important in this book. Abstracting from this example and considering residue classes $(\mathbb{Z}_n,+)$ for arbitrary moduli $n$, it can be shown that $(\mathbb{Z},+)$ is a commutative group with neutral element

0 and additive inverse $n-r$ for any element $r \in \mathbb{Z}_n$. We call such a group the *reminder class groups* of modulus $n$.

Of particular importance for pairing based cryptography in general and snarks in particular are so called *pairing maps* on commutative groups. To be more precise let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ be three commutative groups. For historical reasons, we write the group law on $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation and the group law on $\mathbb{G}_3$ in multiplicative notation. Then a **pairing map** is a function

$$e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3 \tag{5.1}$$

that takes pairs $(g_1, g_2)$ (products) of elements from $\mathbb{G}_1$ and $\mathbb{G}_2$ and maps them somehow to elements from $\mathbb{G}_3$, such that the *bilinearity* property holds: For all $g_1, g_1' \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ we have $e(g_1 + g_1', g_2) = e(g_1, g_2) \cdot e(g_1', g_2)$ and for all $g_1 \in \mathbb{G}_1$ and $g_2, g_2' \in \mathbb{G}_2$ we have $e(g_1, g_2 + g_2') = e(g_1, g_2) \cdot e(g_1, g_2')$.

A pairing map is called *non-degenerated*, if whenever the result of the pairing is the neutral element in $\mathbb{G}_3$, one of the input values must be 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 $g_2 = e_{\mathbb{G}_2}$.

So roughly speaking bilinearity means, that it doesn't matter if we first execute the group law on any side and then apply the bilinear map of if we first apply the bilinear map and then apply the group law. Moreover non-degeneray means that the result of the pairing is zero, only if at least one of the input values is zero.

**Example 32.** *Maybe the most basic example of a non-degenerate pairing is obtained, if we take $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ all to be the group of integers with addition $(\mathbb{Z}, +)$. Then the following map*

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

*defines aa non-degenerate pairing. To see that observe, that bilinearity follows from the distributive law of integers, since for $a,b,c \in \mathbb{Z}$, we have $e(a+b,c) = (a+b)\cdot c = a\cdot c + b \cdot c = e(a,c) + e(b,c)$ and the same reasoning is true for the second argument.*

*To the that $e(\cdot,\cdot)$ is non degenrate, assume that $e(a,b) = 0$. Then $a \cdot b = 0$ and this implies that $a$ or $b$ must be zero.*

**Exercise 26.** *Consider example XXX again and let $\mathbb{F}_5^*$ be the set of all remainder classes from $\mathbb{F}_5$ without the class 0. Then $\mathbb{F}_5^* = \{1,2,3,4\}$. Show that $(\mathbb{F}_5^*, \cdot)$ is a commutative group.*

**Exercise 27.** *Generalizing the previous exercise, consider general moduli $n$ and let $\mathbb{Z}_n^*$ be the set of all remainder classes from $\mathbb{Z}_n$ without the class 0. Then $\mathbb{Z}_n^* = \{1,2,\ldots,n-1\}$. Give 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 proof the commutative group axioms.*

**Exercise 28.** *Consider the remainder class groups $(\mathbb{Z}_n, +)$ for some modulus $n$. Show that the map*

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

*is bilinear. Why is it not a pairing in general and what condition must be imposed on $n$, such that the map is a pairing?*

**Finite groups** As we have seen in the previous examples, groups can either contain infinite many elements (as the 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**.

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

**Example 33.** *Consider the remainder class groups $(\mathbb{Z}_6,+)$ and $(\mathbb{F}_5,+)$ from example XXX and example XXX and the group $(\mathbb{F}_5^*,\cdot)$ from exercise XX. We can easily see that the order of $(\mathbb{Z}_6,+)$ is $6$, the order of $(\mathbb{F}_5,+)$ is five and the order of $(\mathbb{F}_5^*,\cdot)$ is $4$.*

*To be more general, considering arbitrary moduli n, then we know from Euclidean division, that the order of the remainder class group $(\mathbb{Z}_n,+)$ is n.*

**Exercise 29.** *The RSA crypto system is based on a modulus n that is typically the product of two prime numbers of size $2048$-bits. What is (approximately) the order of the rainder class group $(\mathbb{Z}_n,+)$ in this case?*

**Generators** Of special interest, when working with groups are sets of elements that can generate the entire group, by applying the group law repeatedly to those elements or their inverses only.

Of course every group $\mathbb{G}$ has trivially a set of generators, when we just consider every element of the group to be in the generator set. So the more interesting question is to find the smallest set of generators. Of particular interest in this regard are groups that have a single generator, that is there exist an element $g \in \mathbb{G}$, such that every other element from $\mathbb{G}$ can be computed by repeated combination of $g$ and its inverse $g^{-1}$ only. Those groups are called **cyclic groups**.

**Example 34.** *The most basic example of a cyclic group are the integers $(\mathbb{Z},+)$ with integer addition. To see that observe that $1$ is a generator of $\mathbb{Z}$, since every integer can be obtained by repeatedly add either $1$ or its inverse $-1$ to itself. For example $-4$ is generated by $-1$, since $-4 = -1+(-1)+(-1)+(-1)$.*

**Example 35.** *Consider a modulus n and the remainder class groups $(\mathbb{Z}_n,+)$ from example XXX. 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.*

**Example 36.** *Let $p \in ¶$ be prime number and $(\mathbb{F}_p^*,\cdot)$ the finite group from exercise XXX. Then $(\mathbb{F}_p^*,\cdot)$ is cyclic and every element $g \in \mathbb{F}_q^*$ is a generator.*

**The discrete Logarithm problem** In cryptography in general and in snark development in particular, we often do computations "in the exponent" of a generator. To see what this means, 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 is a map, called the **exponential map** with respect to the generator $g$

$$g^{(\cdot)} : \mathbb{Z}_n \to \mathbb{G}\; x \mapsto g^x \tag{5.2}$$

where $g^x$ means "multiply $g$ $x$-times by itself and $g^0 = e_{\mathbb{G}}$. This map has the remarkable property maps the additive group law of the remainder class group $(\mathbb{Z}_n,+)$ in a one-to-one correspondence to the group law of $\mathbb{G}$.

To see that 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 $\mathbb{G}$ and since $g^{x+y} = g^x \cdot g^y$, the map respects the group laws.

Since 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. This is what is usually meant by saying we do our computations "in the exponent".

**Example 37.** *Consider the multiplicative group $(\mathbb{F}_5^*, \cdot)$ from example XXX. We know that $\mathbb{F}_5^*$ is a cyclic group of order* 4 *and that every element is a generator. Choose* $3 \in \mathbb{F}_5^*$*, we then know that the map*

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

*respects the group law of addition in $\mathbb{Z}_4$ and the group law of multiplication in $\mathbb{F}_5^*$. And indeed doing a computation like*

$$\begin{aligned} 3^{2+3-2} &= 3^3 \\ &= 2 \end{aligned}$$

*in the exponent gives the same result as doing the same computation in $\mathbb{F}*_5$, that is*

$$\begin{aligned} 3^{2+3-2} &= 3^2 \cdot 3^3 \cdot 3^{-2} \\ &= 4 \cdot 2 \cdot (-3)^2 \\ &= 3 \cdot 2^2 \\ &= 3 \cdot 4 \\ &= 2 \end{aligned}$$

Since the exponential map is a one-to-one correspondence, that respects the group law, it can be shown that this map has an inverse

$$log_g(\cdot) : \mathbb{G} \to \mathbb{Z}_n \; x \mapsto log_g(x) \tag{5.3}$$

which is called the **discrete logarithm** map with respect to the base $g$. Discrete logarithms are highly importsnt in cryptography as there are groups, such that the exponential map and its inverse the discrete logarithm, are believed to be one way functions, that is while it is possible to compute the exponential map in polynomial time, computing the discrete log takes (sub)-exponential time. We will look at this and similar problems in more detail in the next section.

### 5.1.1 Cryptographic Groups

In this section, we will look at families of groups, which are believed to satisfy certain so called *computational hardness assumptions*, the latter of which is a term to express the hypothesis that a particular problem cannot be solved efficiently (where efficiently typically means "in polynomial time of a given security parameter") in the groups of consideration.

**Example 38.** *To highlight the concept of a computational hardness assumption, consider the group of integers $\mathbb{Z}$ from example XXX. One of the best known and most researched examples of computational hardness is the assumption that the factorization of integers into prime numbers as explained in XXX can not be solved by any algorithm in polynomial time with respect to the bit-length of the integer.*

*To be more precise the computational hardness assumption of integer factorization assumes that given any integer $z \in \mathbb{Z}$ with bit-length $b$, there is no integer $k$ and no algorithm with run time complexity $\mathcal{O}(b^k)$, that is able to find prime numbers $p_1, p_2, \ldots, p_j \in \mathbb{P}$, such that $z = p_1 \cdot p_2 \cdot \ldots \cdot p_j$.*

*Generally speaking, this hardness assumption was proven to be false, since Shor's algorithm shows that integer factorization is at least efficiently possible on a quantum computer, since the run time complexity of this algorithm is $\mathcal{O}(b^3)$. However no such algorithm is known on a classical computer.*

*In the realm of classical computers however, we still have to call the non existence of such an algorithm an "assumption" because to date, there is no proof that it is actually impossible to find some. The problem is that it is hard to reason about algorithms that we don't know.*

*So despite the fact that there is currently no know algorithm that can factor integers efficiently on a classical computer, we can not exclude that such an algorithm might exist in principal and someone eventually will discover it in the future.*

*However what still makes the assumption plausible, despite the absense of any actual proof, is the fact that after decades of extensive search still no such algorithm has been found.*

In what follows, we will describe a few computational hardness assumptions that arrise in the context of groups in cryptography, as we will need them throughout the book.

**The discret logarithm assumption** The so called discrete logarithm problem is one of the most fundamental assumptions in cryptography. To define it, let $\mathbb{G}$ be a finite cyclic group of order $r$ and let $g$ be a generator of $\mathbb{G}$. We know from XXX that there is a so called exponential map $g^{(\cdot)} : \mathbb{Z}_r \to \mathbb{G} : x \mapsto g^x$, which maps the residue classes from module $r$ arithmetic onto the group in a $1:1$ correspondence. The **discrete logarithm problem** is then the task to find inverses to this map, that is, to find a solution $x \in \mathbb{Z}_r$, to the equation

$$h = g^x \tag{5.4}$$

for some given $h \in \mathbb{G}$. The **discrete logarithm assumption (DL-A)** is then the assumption that there exists no algorithm with run time polynimial in the "security parameter $log_2(r)$, that is able to compute some $x$ if ony $h$, $g$ and $g^x$ are given in $\mathbb{G}$. If this is the case for $\mathbb{G}$ we call $\mathbb{G}$ a *DL-A group*.

Rephrasing the previous definition into simple words, DL-A groups are believed to have the property, that it is infeasible to compute some number $x$ that solves the equation $h = g^x$ for given $h$ and $g$, assuming that the size of the group $r$ is large enough.

**Example 39** (Public key cryptography)**.** *One the most basic examples of an application for DL-A groups is in public key cryptography, where some pair $(\mathbb{G}, g)$ is publically agreed on, such that $\mathbb{G}$ is a finite cyclic group sufficiently large order r, where it is believed that the discrete logarithm assumption holds and g is a generator of $\mathbb{G}$.*

*In this setting a secret key is nothing but 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 therefore infisible for an attacker to compute the secret key from the public key, since it is believed to be hard to find solutions x to the equation*

$$pk = g^x$$

As the previous example shows, it is an important practical problem to identify DL-A groups. 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.

**Example 40** (Modular arithmetics for Fermat's primes)**.** *It is widely believed that the discrete logarithm problem is hard in multiplicative groups $\mathbb{Z}_p^*$ of prime number modular arithmetics. However this is not true in general. To see that consider any so called Fermat's prime, which is a prime number $p \in \mathbb{P}$, such that $p = 2^n + 1$ for some number n.*

*We know from XXX, that in this case $\mathbb{Z}_p^* = \{1,2,\ldots,p-1\}$ is group with respect to integer multiplication in modular p arithmetics and since $p = 2^n + 1$, the order of $\mathbb{Z}_p^*$ is $2^n$, which implies that the associated security parameter is given by $log_2(2^n) = n$.*

*We show that in this case $\mathbb{Z}_p^*$ is not a DL-A group, by constructing an algorithm, which is able compute some $x \in \mathbb{Z}_{2^n}$ for any given generator g and arbitry element h of $\mathbb{F}_p^*$, such that*

$$h = g^x$$

*holds and the run time complexity of the constrcted algorithm is $\mathcal{O}(n^2)$, which is quadratic in the security parameter $n = log_2(2^n)$.*

*To define such an algorithm, lets assume that the generator g is a public constant and that a group element h is given. Our task is to compute x efficiently.*

*A first thing to note is that since x is a number in modular $2^n$ aithmetic, we can write the binary representation of x as*

$$x = c_0 \cdot 2^0 + c_1 \cdot 2^1 + \cdots + c_n \cdot 2^n$$

*with binary coefficients $c_j \in \{0,1\}$. In particular x as an n-bit number, if interpreted as an integer.*

*We then use this representation to construct an algorithm that computes the bits $c_j$ one after another, starting at $c_0$. To see how this can be achieved, observe that we can determine $c_0$ by raising the input h to the power of $2^{n-1}$ in $\mathbb{F}_p^*$. We use the exponential laws and compute*

$$\begin{aligned}
h^{2^{n-1}} &= (g^x)^{2^{n-1}} \\
&= \left(g^{c_0\cdot 2^0 + c_1\cdot 2^1 + \ldots + c_n\cdot 2^n}\right)^{2^{n-1}} \\
&= g^{c_0\cdot 2^{n-1}} \cdot g^{c_1\cdot 2^1\cdot 2^{n-1}} \cdot g^{c_2\cdot 2^2\cdot 2^{n-1}} \cdots g^{c_n\cdot 2^n\cdot 2^{n-1}} \\
&= g^{c_0 2^{n-1}} \cdot g^{c_1\cdot 2^0\cdot 2^n} \cdot g^{c_2\cdot 2^1\cdot 2^n} \cdots g^{c_n\cdot 2^{n-1}\cdot 2^n}
\end{aligned}$$

*Now since g is a generator and $\mathbb{F}_p^*$ is cyclic of order $2^n$, we know $g^{2^n} = 1$ and therefore $g^{k\cdot 2^n} = 1^k = 1$. From this follows that all but the first factor in the last expressen are equal to 1 and we can simplify the expression into*

$$h^{2^{n-1}} = g^{c_0 2^{n-1}}$$

*Now in case $c_0 = 0$, we get $h^{2^{n-1}} = g^0 = 1$ and in case $c_0 = 1$ we get $h^{2^{n-1}} = g^{2^{n-1}} \neq 1$ (To see that $g^{2^{n-1}} \neq 1$, recall that g is a generator of $\mathbb{F}_p^*$ and hence is cyclic of order $2^n$, which implies $g^y \neq 1$ for all $y < 2^n$).*

*So raising h to the power of $2^{n-1}$ determines $c_0$ and we can apply the same reasoning to the coefficient $c_1$ by raising $h \cdot g^{-c_0\cdot 2^0}$ to the power of $2^{n-2}$. This approach can then be repeated until all the coefficients $c_j$ of x are found.*

*Assuming that exponentiation in $\mathbb{F}_p^*$ can be done in logarithmic run time complexity $log(p)$, it follows that our algorithm has a run time complexity of $\mathcal{O}(log^2(p)) = \mathcal{O}(n^2)$, since we have to execute n exponentiations to determine the n binary coefficients of x.*

*From this follows that whenever p is a Fermat's prime, the discrete logarithm assumption does not hold in $F_p^*$.*

**The decisional Diffi Hellman assumption** To describe the decisional Diffie–Hellman assumption , let $\mathbb{G}$ be a finite cyclic group od order $r$ and let $g$ be a generator of $\mathbb{G}$. The DDH assumption then assumes that there is no algorithm that has a run time complexity polynomial in the security parameter $s = log(r)$, that is able to distiguish the so called DDH-tripple $(g^a, g^b, g^{ab})$ from any tripple $(g^a, g^b, g^c)$ for randomly and independently choosen parameters $a, b, c \in \mathbb{Z}_r$. If this is the case for $\mathbb{G}$ we call $\mathbb{G}$ a DDH-A group.

It is easy to see that DDH-A is a stronger assumption then DL-A, in the sense that the discrete logarithm assumption is neccessary for the dicisional Diffi Hellman assumption to hold, but not the other way around.

To see why, assume that the discrete logarithm assumption does not hold. In that case given a genrator $g$ and a group element $h$, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. Then the dicisional Diffi-Hellman assumption could not hold, since given some tripple $(g^a, g^b, z)$, one could efficiently decide whether $z = g^{ab}$ by first computing the discrete logarithm $b$ of $g^b$, then compute $g^{ab} = (g^a)^b$ and decide 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 Diffi Hellman assumption does not hold:

**Example 41** (Efficiently computable pairings)**.** *Let $\mathbb{G}$ be a finite, cyclic group of order $r$ with generator $g$, such that the discrete logarithm assumtion holds and such that there is a pairing map $e(\cdot,\cdot) : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ for some target group $\mathbb{G}_T$ that is computable in polynomial time of the parameter $log(r)$.*

*In a setting like this it is easy to show that DDH-A can not hold, since given some tripple $(g^a, g^b, z)$, it is possible to decide in polynomial times w.r.t $log(r)$ whether $z = g^{ab}$ or not. To see that check*

$$e(g^a, g^b) = e(g, z)$$

*Since the bilinierity 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 degeneray property, the equality decides $z = e^{ab}$.*

It follows that DDH-A is indeed weaker then DL-A and groups with efficient pairings can not be DDH-A groups. As the following example shows, another important class of groups, where DDH-A does not hold are the multiplicative groups of prime number residue classes.

**Example 42.** *Let $p$ be a prime number and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ the multiplicative group of modular $p$ arithmetics as in example XXX. As we have seen in XXX, this group is finite and cyclic of order $p-1$ and every element $g \neq 1$ is a generator.*

*To see that $\mathbb{F}_p^*$ can not be a DDH-A group recall from XXX that the Legendre symbol $\left(\frac{x}{p}\right)$ of any $x \in \mathbb{F}_p^*$ is efficiently computable by Euler's formular. But the Legendre symbol of $g^a$ reveals if $a$ is even or odd. Given $g^a$, $g^b$ and $g^{ab}$, one can thus efficiently compute and compare the least significant bit of $a$, $b$ and $ab$, respectively, which provides a probabilistic method to distinguish $g^{ab}$ from a random group element $g^c$.*

**The computational Diffi Hellman assumption** To describe the computational Diffie-Hellman assumption, let $\mathbb{G}$ be a finite cyclic group od order $r$ and let $g$ be a generator of $\mathbb{G}$. The computational Diffi-Hellman assumption, then assumes that given randomly and independently choosen residue classes $a, b \in \mathbb{Z}_r$, 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-A group.

In general it is not know if CDH-A is a stronger assumption then DL-A, or if both assumptions are equivalent. It is known that DL-A is necessary for CDH-A but the other direction is currently not well understood. In particular there are no groups known where DL-A holds but CDH-A does not hold.

To see why the discrete logarithm assumption is necessary, assume that it does not hold. So given a genrator $g$ and a group element $h$, it is easy to compute some residue class $x \in \mathbb{Z}_p$ with $h = g^x$. In that case the computational Diffi-Hellman assumption can not hold, since given $g$, $g^a$ and $g^b$, one can efficiently compute $b$ and hence is able to compute $g^{ab} = (g^a)^b$ from this data.

The computational Diffi-Hellman assumption is a weaker assumption then the decisional Diffi Hellman assumption, which means that there are groups where CDH-A holds and DDH-A does not holdm while there can not be groups such that DDH-A holds but CDH-A 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 of course easy to decide if $z = g^{ab}$ or not.

From the CDH-A various variations and specializations are known. For example the so called *square computational Diffi Hellman assumption* asumes, that given $g$ and $g^x$ it is computationally hard to compute $g^{x^2}$ while the so called *inverse computational Diffi Hellman assumption* asumes, that given $g$ and $g^x$ it is computationally hard to compute $g^{x^{-1}}$.

**Cofactor Clearing** TODO: (theorem: every factor of order defines a subgroup...)

### 5.1.2 Hashing to Groups

**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 general way to represent arbitrary data, we can understand a general **hash function** as a map

$$H : \{0,1\}^* \to \{0,1\}^k \tag{5.5}$$

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. So in our definition a hash function maps binary strings of arbitrary size onto binary strings of size exactly $k$. We call the images of $H$, that is the values returned by the hash function *hash values*, *digests*, or simply *hashes*.

A hash function must be deterministic, that is inserting the same input $x$ into $H$, so image $H(x)$ must always be the same. In addition a hash function should be as uniform as possible, which means that it should map input values as evenly as possible over its output range. In mathematical terms every length $k$ string from $\{0,1\}^k$ should be generated with roughly the same probability.

**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 $s.len() > k$ to a string of size $k$ and by filling any string $s'$ of size $s'.len() < 0$ with zeros. To make this hash function deterministic, we define that both truncation and filling should happen "on the left".*

*For example if $k = 3$, $x_1 = (0000101011101010011101010101)$ and $x_2 = 1$ then $H(x_1) = (101)$ and $H(x_2) = (001)$. It is easy to see that this hash function is deterministic and uniform.*

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 $\{0,1\}^k$ its practically infeasible to find a string $x \in \{0,1\}^*$ such that $H(x) = y$ holds. This property is usually called *preimage-resistence*.

In addition it should be infeasible to find to strings $x_1, x_2 \in \{0,1\}^*$, such that $H(x_1) = H(x_2)$, which is called *collision resistence*. It is important to note though, that collisions always exists, since a function $H : \{0,1\}^* \to \{0,1\}^k$ inevitable maps infinite many values onto 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 flipping a single bit, should generate hash values that look completely different from each other.

As cryptographically secure hash functions map tiny changes in input values onto large change in the output, implementation errors that change the outcome are usually easy to spot by comparing them to expected output values. The definition of cryptographically secure hash function are therefore usually acompanied 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)**.** *Considering the k-truncation hash from example XXX. Since the empty string has length* 0 *it follows that the digest of the empty string is string of length k that only contains* 0*'s. i.e*

$$H_k('') = (000\dots000)$$

*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 onw preimage, since* $H_k(y) = y$ *for every string of size exactly k. Finding preimages is therefore easy.*

*In addition it is easy to construct collusions as all strings of size* $> k$ *that share the same k-bits "on the right" are mapped to the same hash value.*

*Also this hash function is not very chaotic, as changing bits that are not part of the k right most bits don't change the digest at all.*

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

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

*that maps binary strings of arbitrary length onto binary strings of length* 256*. To evaluate a proper implementation of the SHA*256 *hash function the giest of the empty string is supposed to be*

$$SHA256('') = e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855$$

*For better human readability it is common practise to represent the digest of a string, not in its binary form but in a hexadecimal representation. We can use sage to compute SHA*256 *and freely transit between binary, hexadecimal and decimal representations. To do so we have to import PyCrypto and then load SHA_256:*

```
sage: import Crypto
sage: from Crypto.Hash import SHA256
sage: test = 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934
   ca495991b7852b855'
```

```
sage: d = SHA256.new('')
sage: str = d.hexdigest()
sage: type(str)
<type 'str'>
sage: d = ZZ('0x'+ str) # conversion to integer type
sage: d.str(16) == str
True
sage: d.str(16) == test
True
sage: d.str(16)
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b8
   55
sage: d.str(2)
11100011101100001100010001000010100110001111110000011100000101
   00100110101111101111110100110010001001100101101111101110010
   01001000010011110101110010000011110010001100100100110111001
   00110100110010100100100101011001100100011011011110000101001
   01011100001010101
sage: d.str(10)
10298733624955409702953521232258132278979990064819803499337939
   7001115665086549
```

**Hashing to cyclic groups** As we have seen in the previous paragraph general hash functions map binary strings of arbitrary length onto binary strings of length $k$ for some parameter $k$. In various cryptographic primitives it is however desireable to not simply hash to binary strings of fixed length but to hash into algebraic structures like groups, while keeping (some of) the properties like preimiage or collision resistence.

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 usually be extended easily 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

$$H : \{0,1\}^* \to \mathbb{G} \tag{5.6}$$

Common properties of hash functions, like uniformity are desireable but not always realized in actual real world instantiations of hash-to-group functions, so we skip those requirements for now and keep the definition very general.

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

**Example 46** (Naive cyclic group hash). *Let $\mathbb{G}$ be a finite cyclic group. If the task is to implement a hash-to-$\mathbb{G}$ 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 $0 \le z < 2^k$.*

*To be more precise, choose an ordinary hash function $H : \{0,1\}^* \to \{0,1\}^k$ for some parameter $k$ and a generator $g$ of $\mathbb{G}$. Then the expression*

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

*is a positive integer, where $H(s)_j$ means the bit at the j-th position of $H(s)$. A hash-to-group function for the group $\mathbb{G}$ can then be defined as a concatenation of the exponential map $g^{(\cdot)}$ of g with the interpretation of $H(s)$ as an integer:*

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

*Constructing a hash-to-group function like this is easy to implement for cyclic groups and might be good enough in certain applications. It is however almost never adequate in cryptographic applications as discrete log relations might be constructible between two given hash value $H_g(s)$ and $H_g(t)$.*

*To see that, assume that $\mathbb{G}$ is of order r and that $z_{H(s)}$ has a multiplicative inverse in modular r arithmetics. In that case we can compute $x = z_{H(t)} \cdot z_{H(s)}^{-1}$ in $\mathbb{Z}_r$ and have found a discrete log relation between the group hash values, that is we have found some x with $H_g(t) = (H_g(s))^x$ since*

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

Applications where discrete log relations between hash values are undesireable therefore need different approaches and many of those approaches start with a way to hash into the sets $\mathbb{Z}_r$ of modular $r$ arithmetics.

**Hashing to modular arithmetics** One of the most widely used applications of hash-into-group functions are hash functions that map into the set $\mathbb{Z}_r$ of modular $r$ arithmetics for some modulus $r$. Different approaches to construct such a function are known, but probably the most used once are based on the insight that the images of arbitrary hash functions can be interpreted as binary representations of integers as explained in example XXX.

From this interpretation follows that one simple method of hashing into $\mathbb{Z}_r$ is constructed by observing, that if $r$ is a modulus, with a bit-length of $k = r.nbits()$, then every binary string $(b_0, b_1, \ldots, b_{k-2})$ of length $k-1$ defines an integer $z$ in the rage $0 \le z < 2^{k-1} \le r$, by defining

$$z = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_{k-2} \cdot 2^{k-2} \tag{5.7}$$

Now since $z < r$, we know that $z$ is guranteed to be in the set $\{0, 1, \ldots, r-1\}$ and hence can be interpreted as an element of $\mathbb{Z}_r$. From this follows that if $H : \{0,1\}^* \to \{0,1\}^{k-1}$ is a hash function, then a hash-to-group function can be constructed by

$$H_{r.nbits()-1} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-2} \cdot 2^{k-2} \tag{5.8}$$

where $H(s)_j$ means the $j$'s bit of the image binary string $H(s)$ of the original binary hash function.

A drawback of this hash function is that the distribution of the hash values in $\mathbb{Z}_r$ is not necessarily uniform. In fact if $r - 2^{k-1} \neq 0$, then by design $H_{r.nbits()-1}$ will never hash onto values $z \geq 2^{k-1}$. Good moduli $r$ are therefore as close to $2^{k-1}$ as possible, why less good moduli are closer to $2^k$. In the worst case, that is $r = 2^k - 1$, it misses $2^{k-1} - 1$, that is almost half of all elements, from $\mathbb{Z}_r$.

An advantage is that properties like preimage or collision resistence of the original hash function $H(\cdot)$ are preserved.

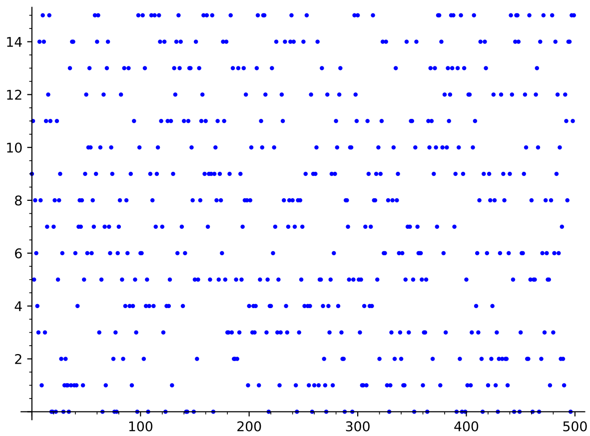
**Example 47.** *To give an implementation of the $H_{r.nbits()-1}$ hash function, we use a 5-bit truncation of the SHA256 hash from example XXX and define a hash into $\mathbb{Z}_{16}$ by*

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

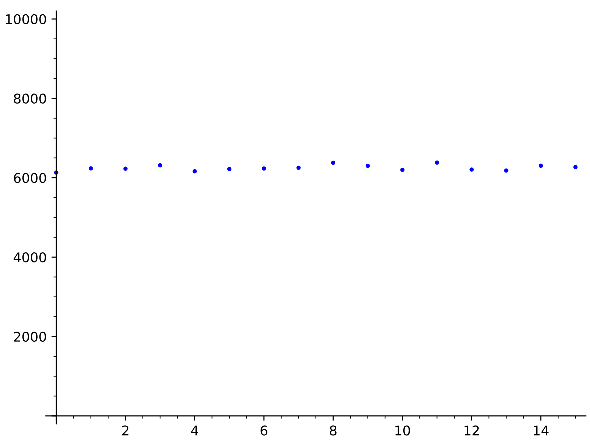
*Since $k = 16.nbits() = 5$ and $16 - 2^{k-1} = 0$ this hash maps uniformly onto $\mathbb{Z}_{16}$. We can invoke sage to implement it e.g. like this:*

```
sage: import Crypto
sage: from Crypto.Hash import SHA256
sage: def Hash5(x):
....:     h = SHA256.new(x)
....:     d = h.hexdigest()
....:     d = ZZ(d, base=16)
....:     d = d.str(2)[-4:]
....:     return ZZ(d, base=2)
sage: Hash5('')
5
```

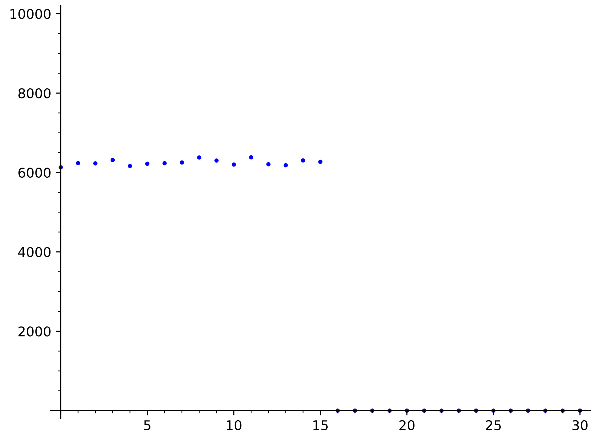
*We can then use sage to apply this function to a large set of input values in order to plot a visualization of the distribution over the set $\{0,\ldots,15\}$. Executing over 500 input values gives:*




*To get an intuition of uniformity, we can count the number of times the hash function $H_{16.nbits()-1}$ 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 uniformity of the distribution problem becomes apparent if we want to constructht a similar hash function for $\mathbb{Z}_r$ for any r in the range $17 \leq r \leq 31$. 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 16. So for example in case of hashing to $\mathbb{Z}_{31}$ the hash function never maps to any value larger then 16, leaving almost half of all numbers out of the image range.*




The second widely used method of hashing into $\mathbb{Z}_r$ is constructed by observing, that if $r$ is a modulus, with a bit-length of $r.bits() = k_1$ and $H : \{0,1\}^* \to \{0,1\}^{k_2}$ is a hash function that produces digests of size $k_2$, with $k_2 \geq k_1$, then a hash-to-group function can be constructed by interpreting the image of $H$ as binary representation of a integer and then take the modulus by $r$ to map into $\mathbb{Z}_r$. To be more precise

$$H'_{mod_r} : \{0,1\}^* \to \mathbb{Z}_r : \; s \mapsto \left( H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k_2} \cdot 2^{k_2} \right) \bmod n \tag{5.9}$$

where $H(s)_j$ means the $j$'s bit of the image binary string $H(s)$ of the original binary hash function.

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

**Example 48.** *To give an implementation of the $H_{mod_r}$ hash function, we use $k_2$-bit truncation of the SHA256 hash from example XXX and define a hashes into $\mathbb{Z}_{23}$ by*

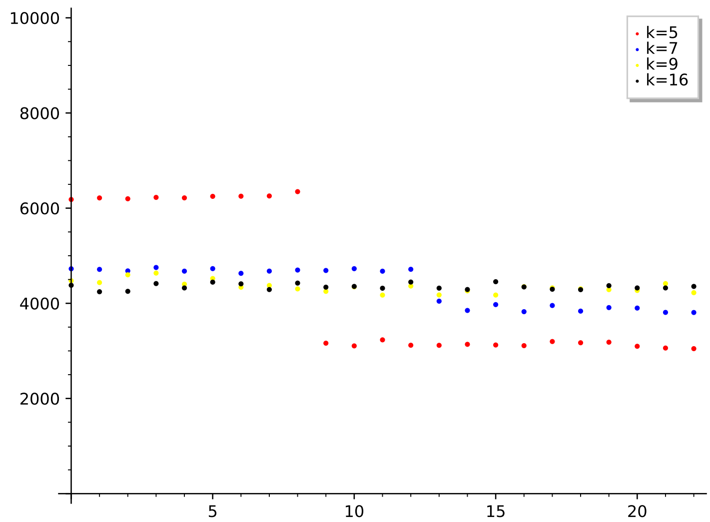
$$\begin{aligned} H_{mod_{23},k_2} &: \{0,1\}^* \to \mathbb{Z}_{23} : \\ & s \mapsto \left( SHA256(s)_0 \cdot 2^0 + SHAH256(s)_1 \cdot 2^1 + \ldots + SHA256(s)_{k_2} \cdot 2^{k_2} \right) \; mod \; 23 \end{aligned}$$

*We want to use various instantiations of $k_2$, to visualize the impact of truncation lenth on the distrubution of the hashes in $\mathbb{Z}_{23}$. We can invoke sage to implement it e.g. like this:*

```
sage: import Crypto                                    172
sage: from Crypto.Hash import SHA256                   173
sage: Z23 = Integers(23)                               174
```

```
sage: def Hash_mod23(x, k2):
....:     h = SHA256.new(x)
....:     d = h.hexdigest()
....:     d = ZZ(d, base=16)
....:     d = d.str(2)[-k2:]
....:     d = ZZ(d, base=2)
....:     return Z23(d)
```

*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*




A third method that can sometimes be found in implementations is the so called *try and increment method*. To underatand this method, we define an integer $z \in \mathbb{Z}$ from any hash value $H(s)$ as we did in the previous methods, that is we define $z = H(s)_0 \cdot 2^0 + H(s)_1 \cdot 2^1 + \ldots + H(s)_{k-1} \cdot 2^k$.

Hashing into $\mathbb{Z}_r$ is then achievable by first computing $z$ and then try to see if $z \in \mathbb{Z}_r$. If this is the case than the hash is done and if not the string $s$ is modified in a deterministic way and the process is repeated until a sutable number $z$ is found. A suiteable, deterministic modification could be to concatenate the original string by some bit counter. A try and increment algorithm would then work like in algorithm XXX

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

Once some hash function into modular arithmetics is found it can often be combined with additional techniques to hash into more general finite cyclif groups. The following paragraphs describes a few of thos methods widely adopted in snark development.

**Pederson Hashes** The so called **Pedersen hash function** provides a way to map binary inputs of fixed size $k$ onto elements of finite cyclic groups, that avoids discrete log relations between the images as they occure in the naive approach XXX. Combining it with a classical hash function provides a hash function that maps strings of arbitrary lenth onto group elements.

**Algorithm 5** Hash-to-$\mathbb{Z}_n$

**Require:** $r \in \mathbb{Z}$ with $r.nbits() = k$ and $s \in \{0,1\}^*$
  **procedure** TRY-AND-INCREMENT$(r,k,s)$
    $c \leftarrow 0$
    **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** $z < r$
    **return** $x$
  **end procedure**
**Ensure:** $z \in \mathbb{Z}_r$

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

$$H_{Ped} : (\mathbb{Z}_r)^j \to \mathbb{G} : \ (x_1,\ldots,x_j) \mapsto \Pi_{i=1}^{j} g_j^{x_j} \tag{5.10}$$

It can be shown, that Pedersen's hash function is collision-resistant under the assumption that $\mathbb{G}$ is a DL-A group. However it is important to note, that Pedersen hashes cannot be assumed to be pseudorandom and should therefore not be used where a hash function serves as an approximation of a random oracle.

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

To see how Pedersen hashes can be used to define an actual hash-to-group function according to our definition, we can use any of the hash-to-$\mathbb{Z}_r$ functions as we have derived them in XXX.

#### MimC Hashes

**Pseudo Random Functions in DDH-A groups** As noted in XXX, Pederson's hash function does not have the properties a random function and should therefore not be instantiated as such. To look at a construction that serves as random oracle function in groups where the dicisional Diffi-Hellman construction is assumed to hold true let $\mathbb{G}$ be a DDH-A group of order $r$ with generator $g$ and $\{a_0,a_1,\ldots,a_k\} \subset \mathbb{Z}_r^*$ a uniform randomly generated set of numbers invertible in modular $r$ arithmetics. Then a pseudo-random function is given by

$$F_{rand} : \{0,1\}^{k+1} \to \mathbb{G} : \ (b_0,\ldots,b_k) \mapsto g^{b_0 \cdot \Pi_{i=1}^{k} a_i^{b_i}} \tag{5.11}$$

Of course if $H : \{0,1\}^* \to \{0,1\}^{k+1}$ is a random oracle, then the concation of $F_{rand}$ and $H$, defines a random oracle

$$H_{rand,\mathbb{G}} : \{0,1\}^* \to \mathbb{G} : \ s \mapsto F_{rand}(H(s)) \tag{5.12}$$

## 5.2 Commutative Rings

Thinking of integers again, we know, that there are actually two operations addition and multiplication and as we know addition defines a group structure on the set of integers. However multiplication does not define a group structure as we know that integers in general don't have multiplicative inverses.

Combinations like this are captured by the concept of a so called *commutative ring with unit*. To be more precise, a commutative ring with unit $(R,+,\cdot,1)$ is a set $R$, provided with two maps $+ : R\cdot R \to R$ and $\cdot : R\cdot R \to R$, called *addition* and *multiplication*, such that the following conditions hold:

- $(R,+)$ is a commutative group, where the neutral element is denoted with 0.
- (Commuativity of the multiplication) We have $r_1 \cdot r_2 = r_2 \cdot r_1$ for all $r_1, r_2 \in R$.
- (Existence of a unit) There is an element $1 \in R$, such that $1 \cdot g$ holds 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 laws $g_1 \cdot (g_2 + g_3) = g_1 \cdot g_2 + g_1 \cdot g_3$ holds.

**Example 49** (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.

**Example 50** (Underlying commutative group of a ring)**.** *Every commutative ring with unit $(R,+,\cdot,1)$ gives rise to group, if we just forget about the multiplication*

The following example is more interesting. The motivated reader is encouraged to think through this example, not so much because we need this in what follows, but more so as it helps to detach the reader from familiar styles of computation.

**Example 51.** *Let $S := \{\bullet, \star, \odot, \otimes\}$ be a set that contains four elements and let addition and multiplication on S be defined as follows:*

| $\cup$ | $\bullet$ | $\star$ | $\odot$ | $\otimes$ |
|---|---|---|---|---|
| $\bullet$ | $\bullet$ | $\star$ | $\odot$ | $\otimes$ |
| $\star$ | $\star$ | $\odot$ | $\otimes$ | $\bullet$ |
| $\odot$ | $\odot$ | $\otimes$ | $\bullet$ | $\star$ |
| $\otimes$ | $\otimes$ | $\bullet$ | $\star$ | $\odot$ |

| $\circ$ | $\bullet$ | $\star$ | $\odot$ | $\otimes$ |
|---|---|---|---|---|
| $\bullet$ | $\bullet$ | $\bullet$ | $\bullet$ | $\bullet$ |
| $\star$ | $\bullet$ | $\star$ | $\odot$ | $\otimes$ |
| $\odot$ | $\bullet$ | $\odot$ | $\bullet$ | $\odot$ |
| $\otimes$ | $\bullet$ | $\otimes$ | $\odot$ | $\star$ |

*Then $(S,\cup,\circ)$ is a ring with unit $\star$ and zero $\bullet$. It therefore makes sense to ask for solutions to equations like this one: Find $x \in S$ such that*

$$\otimes \circ (x \cup \odot) = \star$$

*To see how such a "moonmath equation" can be solved, we have to keep in mind, that rings behaves mostly like normal number when it comes to bracketing and computation rules. The only differences are the symbols and the actual way to add and multiply. With this we solve the* *equation for x in the "usual way"*

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

*So even despite this equation looked really alien on the surface, computation was basically exactly the way "normal" equation like for fractional numbers are done.*

*Note however that in a ring, things can be very different, then most are used to, whenever a multiplicative inverse would be needed to solve an equation in the usual way. For example the equation*

$$\odot \circ x = \otimes$$

*can not be solved for x in the usual way, since there is no multiplicative inverse for $\odot$ in our ring. And in fact looking at the multiplication table we see that no such x exits. On another example the equation*

$$\odot \circ x = \odot$$

*can has not a single solution but two $x \in \{\star, \otimes\}$. Having no or two solutions is certainly not something to expect from types like $\mathbb{Q}$.*

**Example 52.** *Considering polynomials again, we note from their definition, that what we have called the type R of the coefficients, must in fact be a commutative ring with unit, since we need addition, multiplication, commutativity and the existence of a unit for $R[x]$ to have the properties we expect.*

*Now considering R to be a ring, addition and multiplication of polynomials as defined in XXX, actually makes $R[x]$ into a commutative ring with unit, too, where the polynomial 1 is the multiplicative unit.*

**Example 53.** *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 XXX. It can be shown that $(\mathbb{Z}_n, +, \cdot)$ is a commutative ring with unit 1.*

Considering the exponential map from XXX again, let $\mathbb{G}$ be a finite cyclic group of order $n$ with generator $g \in \mathbb{G}$. Then the ring structure of $(\mathbb{Z}_n, +, \cdot)$ is mapped onto the group structure of $\mathbb{G}$ in the following way:

$$
\begin{aligned}
g^{x+y} &= g^x \cdot g^y && \text{for all } x, y \in \mathbb{Z}_n\\
g^{x \cdot y} &= (g^x)^y && \text{for all } x, y \in \mathbb{Z}_n
\end{aligned}
$$

This of particular interest in cryptographic and snarks, as it allows for the evaluation of polynomials with coefficients in $\mathbb{Z}_n$ to be evaluated "in the exponent". To be more precise let $p \in \mathbb{Z}_n[x]$

be a polynomial with $p(x) = a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1x + a_0$. Then the previously defined exponential laws XXX imply that

$$\begin{aligned} g^{p(x)} &= g^{a_m \cdot x^m + a_{m-1}x^{m-1} + \ldots + a_1x + a_0} \\ &= \left(g^{x^m}\right)^{a_m} \cdot \left(g^{x^{m-1}}\right)^{a_{m-1}} \cdot \ldots \cdot (g^x)^{a_1} \cdot g^{a_0} \end{aligned}$$

and hence to evaluate $p$ at some point $s$ in the exponent, we can insert $s$ into the right hand side of the last equation and evaluate the product.

As we will see this is a key insight to understand many snark protocols like e.g. Groth16 or XXX.

**Example 54.** *To give an example for the evaluation of a polynomial in the exponent of a finite cyclic group, consider the exponential map*

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

*from example XXX. Choosing the polynomial $p(x) = 2x^2 + 3x + 1$ from $\mathbb{Z}_4[x]$, we can evaluate the polynomial at say $x = 2$ in the exponent of $3$ in two different ways. On the one hand side we can evaluate $p$ at $2$ and then write the result into the exponent, which gives*

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

*and on the other hand we can use the right hand side of equation to evaluate $p$ at $2$ in the exponent of $3$, which gives:*

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

**Hashing to Commutative Rings** As we have seen in XXX various constructions for hashing-to-groups are known and used in applications. As commutative rings are abelian groups, when we simply forget about the multiplicative structure, hash-to-group constructions can be applied for hashing into commutative rings, too. This is possible in general as the codomain of a general hash function $\{0,1\}^*$ is just the set of binary strings of arbitrary but finite lenth, which hash no algebraic structure that the hash function must respect.

## 5.3 Fields

In this chapter we started with the definition of a group, which we the expended into the definition of a commutative ring with unit. Those rings generalize the behaviour of integers. In this section we will look at the special case of commutative rings, where every element, other than the neutral element of addition, has a multiplicative inverse. Those structures behave very much like the rational numbers $\mathbb{Q}$, which are in a sense an extension of the ring of integers, that is constructed by just including newly defined multiplicative inverses (the fractions) to the integers.

Now considering the definition of a ring XXX again, we define a **field** $(\mathbb{F},+,\cdot)$ to be a set $\mathbb{F}$, together with two maps $+ : \mathbb{F}\cdot\mathbb{F} \to \mathbb{F}$ and $\cdot : \mathbb{F}\cdot\mathbb{F} \to \mathbb{F}$, called *addition* and *multiplication*, such that the following conditions holds

- $(\mathbb{F},+)$ 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) For all $g_1,g_2,g_3 \in \mathbb{F}$ the distributive law $g_1\cdot(g_2+g_3) = g_1\cdot g_2 + g_1\cdot g_3$ holds.

If a field is iven and the definition of its addition and multiplication is not ambiguous, we will often simple write $\mathbb{F}$ instead of $(\mathbb{F},+,\cdot)$ to describe it. We moreover write $\mathbb{F}^*$ to describe the multiplicative group of the field, that is the set of elements, except the neutral element of addition, with the multiplication as group law.

The **characteristic** $char(\mathbb{F})$ of a field $\mathbb{F}$ is the smallest natural number $n \geq 1$, for which the $n$ -fold sum of 1 equals zero, i.e. for which $\sum_{i=1}^{n} 1 = 0$. If such a $n > 0$ exists, the field is also called to have a *finite characteristic*. If, on the other hand, every finite sum of 1 is not equal to zero, then the field is defined to have characteristic 0.

**Example 55** (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 counting number $n \in \mathbb{N}$, such that $\sum_{j=0}^{n} 1 = 0$ in the rational numbers, the characteristic $char(\mathbb{Q})$ of the field $\mathbb{Q}$ is zero. In sage rational numbers are called like this*

```
sage: QQ
Rational Field
sage: QQ(1/5) # Get an element from the field of rational
   numbers
1/5
sage: QQ(1/5) / QQ(3) # Division
1/15
```

**Example 56** (Field with two elements)**.** *It can be shown that in any field, the neutral element* 0 *of addition must be different from the neutral element* 1 *of multiplication, that is we always have $0 \neq 1$ in a field. From this follows that the smallest field must contain at least two elements and 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:*

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| · | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

*Since $1+1=0$ in the field $\mathbb{F}_2$, we know that the characteristic of $\mathbb{F}_2$ is there, that is we have char$(\mathbb{F}_2)=0$.*

*For reasons we will understand better in XXX, sage defines this field as a so called Galois field with 2 elements. It is called like this:*

```
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
```

**Example 57.** *Both the real numbers $\mathbb{R}$ as well as the complex numbers $\mathbb{C}$ are well known examples of fields.*

**Exercise 30.** *Consider our remainder class ring $(\mathbb{F}_5,+,\cdot)$ and show that it is a field. What is the characteristic of $\mathbb{F}_5$?*

**Prime fields** As we have seen in the various examples of the previous sections, modular arithmetics behaves in many ways similar to ordinary arithmetics of integers, which is due to the fact that remainder class sets $\mathbb{Z}_n$ are commutative rings with units.

However at the same time we have seen in XXX, that, whenever the modulus is a prime number, every remainder class other then the zero class, has a modular multiplicative inverse. This is an important observation, since it immediately implies, that in case of 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 finite characteristic $n$

To distinguish this important case from arbitrary reminder class rings, we write $(\mathbb{F}_p,+,\cdot)$ for the field of all remainder classes for a prime number modulus $p \in \mathbb{P}$ and call it the **prime field** of characteristic $p$.

Prime fields are the foundation for many of the contemporary algebra based cryptographic systems, as they have many desirable properties. One of them is, that since these sets are finite and a prime field of characteristic $p$ can be represented on a computer in roughly $log_2(p)$ amount of space, no precision problems occur, that are for example unavoidable for computer representations of rational numbers or even the integers, because those sets are infinite.

Since prime fields are special cases of remainder class rings, all computations remain the same. Addition and multiplication can be computed by first doing normal integer addition and multiplication and then take the remainder modulus $p$. Subtraction and division can be computed by addition or multiplication with the additive or the multiplicative inverse, respectively. The additive inverse $-x$ of a field element $x \in \mathbb{F}_p$ is given by $p-x$ and the multiplicative inverse of $x \neq 0$ is given by $x^{p-2}$, or can be computed using the extended Euclidean algorithm.

Note however that these computations might not be the fastest to implement on a computer. They are however useful in this book as they are easy to compute for small prime numbers.

**Example 58.** *The smallest field is the field $\mathbb{F}_2$ of characteristic 2 as we have seen it in example XXX. It is the prime field of the prime number 2.*

**Example 59.** *To summarize the basic aspects of computation in prime fields, lets consider the prime field $\mathbb{F}_5$ and simplify the following expression*

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

*A first thing to note is that since $\mathbb{F}_5$ is a field all rules like bracketing (distributivity), summing ect. are identical to the rules we learned in school when we where dealing with rational, real or complex numbers. We get*

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

*In this computation we computed the multiplicative inverse of* 3 *using the identity* $x^{-1} = x^{p-2}$ *in a prime field. This impractical for large prime numbers. Recall that another way of computing the multiplicative inverse is the Extended Euclidean algorithm. To see that again, the task is to compute* $x^{-1}\cdot 3+t\cdot 5 = 1$*, but t is actually irrelevant. We get*

| $k$ | $r_k$ | $x_k^{-1}$ | $t_k = (r_k - s_k\cdot a)\; div\; b$ |
|---|---|---|---|
| 0 | 3 | 1 | · |
| 1 | 5 | 0 | · |
| 2 | 3 | 1 | · |
| 3 | 2 | -1 | · |
| 4 | 1 | 2 | · |

*So the multiplicative inverse of* 3 *in* $\mathbb{Z}_5$ *is* 2 *and indeed if compute* $3\cdot 2$ *we get* 1 *in* $\mathbb{F}_5$.

**Square Roots** In this part we deal with square numbers also called *quadratic residues* and *square roots* in prime fields. This is of particular importance in our studies on elliptic curves as only square numbers can actually be points on an elliptic curve.

To make the intuition of quadratic residues and roots precise, let $p \in \mathbb{P}$ be a prime number and $\mathbb{F}_p$ its associate 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 equation

$$x^2 = y \tag{5.13}$$

In this case $y$ is called a **quadratic residue**. On the other hand, if $y$ is given and the quadratic equation has no $x$ solution, we call $y$ as **quadratic non-residue**. For any $y \in \mathbb{F}_p$ we write

$$\sqrt{y} := \{x \in \mathbb{F}_p \mid x^2 = y\} \tag{5.14}$$

for the set of all square roots of $y$ in the prime field $\mathbb{F}_n$. (If $y$ is a quadratic non-residue, then $\sqrt{y} = \emptyset$ and if $y = 0$, then $\sqrt{y} = \{0\}$)

So roughly speaking, quadratic residues are numbers such that we can take the square root from them and quadratic non-residues are numbers that don't have square roots. The situation therefore parallels the know case of integers, where some integers like 4 or 9 have square roots and others like 2 or 3 don't (as integers).

It can be shown that in any prime field every non zero element has either no square root or two of them. We adopt the convention to call the smaller one (when interpreted as an integer) as the **positive** square root and the larger one as the **negative**. This makes sense, as the larger one can always be computed as the modulus minus the smaller one, which is the definition of the negative in prime fields.

**Example 60** (Quadratic (Non)-Residues and roots in $\mathbb{F}_5$)**.** *Let us consider our example prime field $\mathbb{F}_5$ again. All square numbers can be found on the main diagonal of the multiplication table XXX. As you can see, in $\mathbb{Z}_5$ only the numbers $0$, $1$ and $4$ have square roots and we get $\sqrt{0} = \{0\}$, $\sqrt{1} = \{1,4\}$, $\sqrt{2} = \emptyset$, $\sqrt{3} = \emptyset$ 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, why we will recapitulate it here:

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

$$\left(\frac{y}{p}\right) := \begin{cases} 1 & \text{if } y \text{ has square roots} \\ -1 & \text{if } y \text{ has no square roots} \\ 0 & \text{if } y = 0 \end{cases} \tag{5.15}$$

**Example 61.** *Look at the quadratic residues and non residues in $\mathbb{F}_5$ from example XXX again, we can deduce the following Legendre symbols, from example XXX.*

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

The legendre symbol gives 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 theoretic use, as the following so called *Euler criterion* gives 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

$$\left(\frac{y}{p}\right) = y^{\frac{p-1}{2}}\ . \tag{5.16}$$

**Example 62.** *Look at the quadratic residues and non residues in $\mathbb{F}_5$ from example XXX again, we can compute the following Legendre symbols using the Euler criterium:*

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

**Exercise 31.** *Consider the prime field $\mathbb{F}_{13}$. Find the set of all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ that satisfy the equation*

$$x^2 + y^2 = 1 + 7 \cdot x^2 \cdot y^2$$

**Exponentiation** TO APPEAR...

**Hashing into Prime fields** An important problem in snark development is the ability to hash to (various subsets) of elliptic curves. As we will see in XXX those curves are often defined over prime fields and hashing to a curve then might start with hashing to the prime field. It is therefore of importance to understand who to hash into prime fields.

To understand it, note that in XXX we have looked at a few constructions of how to hash into the residue class rings $\mathbb{Z}_n$ for arbitrary $n > 1$. As prime fiels are just special instances of those rings, all hashing into $\mathbb{Z}_n$ functions can be used for hashing into prime fields, too.

**Extension Fields** We defined prime fields in the previous section. They are the basic building blocks for cryptography in general and snarks in particular.

However as we will see in XX so called *pairing based* snark systems are crucially dependent on group pairings XXX defined over the group of rational points of elliptic curves. For those pairings to be non-trivial the elliptic curve must not only be defined over a prime field but over a so called *extension field* of a given prime field.

We therefore have to understand field extensions. To understand them first observe the field $\mathbb{F}'$ is called an *extension* of a field $\mathbb{F}$, if $\mathbb{F}$ is a subfield of $\mathbb{F}'$, that is $\mathbb{F}$ is a subset of $\mathbb{F}'$ and restricting the addition and multiplication laws of $\mathbb{F}'$ to the subset $\mathbb{F}$ recovers the appropriate laws of $\mathbb{F}$.

Now it can be shown, that whenever $p \in \mathbb{P}$ is a prime and $m \in \mathbb{N}$ a natural number, then there is a field $\mathbb{F}_{p^m}$ with characteristic $p$ and $p^m$ elements, such that $\mathbb{F}_{p^m}$ is an extension field of the prime field $\mathbb{F}_p$.

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

To be more precise let $P \in F_p[x]$ be an irreducible polynomial of degree $m$ with coefficients from the given prime field $\mathbb{F}_p$. Then the underlying set $\mathbb{F}_{p^m}$ of the extension field is given by the set of all polynomials with a degree less then $m$:

$$\mathbb{F}_{p^m} := \{a_{m-1}x^{m-1} + a_{k-2}x^{k-2} + \ldots + a_1x + a_0 \mid a_i \in \mathbb{F}_p\} \tag{5.17}$$

which can be shown to be the set of all remainders when dividing any polynomial $Q \in \mathbb{F}_p[x]$ by $P$. So elements of the extension field are polynomials of degree less than $m$. This is analog to how $\mathbb{F}_p$ is the set of all remainders, when dividing integers by $p$.

Addition in then inheritec from $\mathbb{F}_p[x]$, which means that addition on $\mathbb{F}_{p^m}$ is defined as normal addition of polynomials. To be more precise, we have

$$+ : \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \ , \ (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \sum_{j=0}^{m} (a_j + b_j) x^j \tag{5.18}$$

and we can see that the neutral element is (the polynomial) 0 and that the additive inverse is given by the polynomial with all negative coefficients.

Multiplication in inheritec from $\mathbb{F}_p[x]$, too, but we have to divide the result by our modulus polynomial $P$, whenever the degree of the resulting polynomial is equal or greater to $m$. To be more precise, we have

$$\cdot \ \mathbb{F}_{p^m} \times \mathbb{F}_{p^m} \to \mathbb{F}_{p^m} \ , \ (\sum_{j=0}^{m} a_j x^j, \sum_{j=0}^{m} b_j x^j) \mapsto \left( \sum_{n=0}^{2m} \sum_{i=0}^{n} a_i b_{n-i} x^n \right) \bmod P \tag{5.19}$$

and we can see that the neutral element is (the polynomial) 1. It is however not obvious from this definition how the multiplicative inverse looks.

We can easily 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 $p$ 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 $\mathbb{F}_p$ to polynomials of degree zero.

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 product by $P$ 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 identification between all of them and hence from an abstract point of view they are the same thing. From an implementations point of view however some choices are better, because they allow for faster computations.

To summerize we have seen that when a prime field $\mathbb{F}_p$ is given then 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 $m_2$ and from this we can deduces, that for any given fixed prime number, there are nested sequences of fields

$$\mathbb{F}_p \subset \mathbb{F}_{p^{m_1}} \subset \cdots \subset \mathbb{F}_{p^{m_k}} \tag{5.20}$$

whenever the power $m_j$ divides the power $m_{j+1}$, such that $\mathbb{F}_{p^{m_j}}$ is a subfield of $\mathbb{F}_{p^{m_{j+1}}}$.

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

**Example 63** (The Extension field $\mathbb{F}_{3^2}$)**.** *In (XXX) we have constructed the prime field $\mathbb{F}_3$. In this example we apply the definition (XXX) of a field extension to construct $\mathbb{F}_{3^2}$. We start by choosing an irreducibe polynomial of degree 2 with coefficients in $\mathbb{F}_3$. We try $P(t) = t^2 + 1$. Maybe 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 never zero. WE compute*

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

*This implies, that P is irreducible. The set $\mathbb{F}_{3^2}$ then contains all polynomials of degrees lower then two with coefficients in $\mathbb{F}_3$, which is precisely*

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

*So our extension field contains 9 elements as expected. 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 give 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 |

*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.*

*As it was the case in previous examples, we can use the table to deduce 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 $\geq 2$, we have to divide it by $P$ and keep the remainder. To see how this works compute the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$*

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

*So the product of $t+2$ and $2t+2$ in $\mathbb{F}_{3^2}$ is 2. Doing this computation for all elements give the following multiplication table:*

| $\cdot$ | 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 |

*As it was the case in previous examples, we can use the table 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 $(2t+1)\cdot(2t+2)=1$.*

*From the multiplication table we can also see, that the only quadratic residues in $\mathbb{F}_{3^2}$ are 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, like the rational numbers. To see that lets find all $x\in\mathbb{F}_{3^2}$ that solve the quadratic equation $(t+1)(x^2+(2t+$*

$2)) = 2$. *So we compute:*

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

*Computations in extension fields are arguably on the edge of what can reasonbly be done with pen and paper. Fortunately sage provides us with a simple way to do the 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)
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 32.** *Consider the extension field $\mathbb{F}_{3^2}$ from the previous example and find all pairs of elements $(x,y) \in \mathbb{F}_{3^2}$, such that*

$$y^2 = x^3 + 4$$

**Exercise 33.** *Show that the polynomial $P = x^3 + x + 1$ from $\mathbb{F}_5[x]$ 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 Euklidean algorithm. Then find all $x \in \mathbb{F}_{5^3}$ that solve the equation*

$$(2t^2+4)(x-(t^2+4t+2)) = (2t+3)$$

**Hashing into extension fields** In XXX we have seen how to hash into prime fields. As elements of extension fields can be seen as polynomials over prime fields, hashing into extension fields is therefore possible, if every coefficient of the polynimial is hashed independently.

## 5.4 Projective Planes

Projective planes are a certain type of geometry defined over some given field, that in a sense extend the concept of the ordinary Euclidean plane by including "points at infinity".

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

To understand the idea for the construction 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 additional "points at infinity" such that two different lines always intersect in a single point. Parallel lines intersect "at infinity".

To be more precise, let $\mathbb{F}$ be a field, $\mathbb{F}^3 := \mathbb{F} \times \mathbb{F} \times F$ the set of all three tuples over $\mathbb{F}$ and $x \in \mathbb{F}^3$ with $x = (X, Y, Z)$. Then there is exactly one *line* in $\mathbb{F}^3$ that intersects both $(0,0,0)$ and $x$. This line is given by

$$[X : Y : Z] := \{(k \cdot X, k \cdot Y, k \cdot Z) \mid k \in \mathbb{F}\} \tag{5.21}$$

A *point* in the **projective plane** over $\mathbb{F}$ is then defined as such a *line* and the projective plane is the set of all such points, that is

$$\mathbb{F}\mathbb{P}^2 := \{[X : Y : Z] \mid (X, Y, Z) \in \mathbb{F}^3 \text{ with } (X, Y, Z) \neq (0,0,0)\} \tag{5.22}$$

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

To understand why $[X : Y : Z]$ is called a line, consider the situation, where the underlying field $\mathbb{F}$ are the real numbers $\mathbb{R}$. Then $\mathbb{R}^3$ can be seen as the three dimensional space and $[X : Y : Z]$ is then an ordinary line in this 3-dimensional space that intersects zero and the point with coordinates $X$, $Y$ and $Z$.

The key observation here is, that points in the projective plane, are lines in the 3-dimensional space $\mathbb{F}^3$, also for finite fields, the terms space and line share very little visual similarity with their counterparts over the real numbers.

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

**Notation and Symbols 6** (Projective coordinates)**.** *Projective coordinates of the form* $[X : Y : 1]$ *are descriptions of so called* ***affine points*** *and 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****.*

**Example 64.** *Consider the field* $\mathbb{F}_3$ *from example XXX. As this field only contains, three elements it takes not to much effort to construct its associated projective plane* $\mathbb{F}_3\mathbb{P}^2$*, as we knwo that it only contain* 13 *elemts.*

*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 $(0,0,0)$. Since those lines are parameterized by tuples $(x_1, x_2, x_3)$. We compute:*

$$
\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}
$$

*Those lines define the $13$ points in the projective plane $\mathbb{F}_3\mathbb{P}$ as follows*

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

*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, lets consider the point $[1:2:2]$. As this point in the projective plane is a line in $\mathbb{F}_3^3$, it has the projective coordinates $(1,2,2)$ as well as $(2,1,1)$, since the former coordinate give 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 34.** *Construct the so called Fano plane, that is the projective plane over the finite field $\mathbb{F}_2$.*

# 6 Elliptic Curves

TODO: Elliptic Curve asymmetric cryptography examples. Private key, generator, public key. Generally speaking, elliptic curves are "curves" defined in geometric planes like the Euklidean or the projective plane over some given field. One of the key features of elliptic curves over finite fields from the point of view of cryptography is their set of points has a group law, such that the resulting group is finite and cyclic and it is believed that the discrete logarithm problem on these groups is hard.

A special class of elliptic curves are so called *pairing friendly curve*, which have a notation of a group pairing as defined in XXX. This pairing has cryptographicall nice prperties. Those curve are useful in the development of SNAKS, since they allow to compute so called R1CS-satisfiability "in the exponent" (THIS HAS TO BE REWRITTEN WITH WAY MORE DETAIL)

In this chapter we introduce epileptic curves as they are used in pairing based approaches to the construction of snarks. The eliptic curves we consider are all defined over prime fields or prime field extensions and the reader should be familiar with the contend of the previous section on those fields.

In its most generality elliptic curves are defined as a smooth projective curve of genus 1 defined over some field $\mathbb{F}$ with a distinguished $\mathbb{F}$-rational point, but this definition is not very useful for the introductary character of this book. We will therefore look at 3 more practical definitions in the following sections, by introducing Weierstraß, Montgommery and Edwards curves. All of them are useful in cryptography and nesessary to understand for the contnuation of the book.

## 6.1 Elliptic Curve Arithmetics

### 6.1.1 Short Weierstraß Curves

In this section we introduce the so called short Weierstraß curves, which are the most general types of curves over finite fields of characteristic greater then 3.

We start with their reprrsention in affine space. This reprresentation has the advantage that affine points are just pairs of numbers which is more convinient to work with for the beginner. However it has the disadvantage that a special "point at infinity" that is not a point on the curve, is necessary to describe the group structure. We introduce the elliptic curve group law and describe elliptic curve scalar multiplication, which is nothing but an instatation of the exponential map from general cyclic groups.

Then we look at the projective representation of short Weierstrass curves. It has the advantage that no special sybol is necessary to represent the point at infinity but comew with the drawback that projective points are classes of numbers, which might be a bit unusual for a beginner.

We finish this section with an explicit equivalence that transforms affine representations into projective once and vice versa.

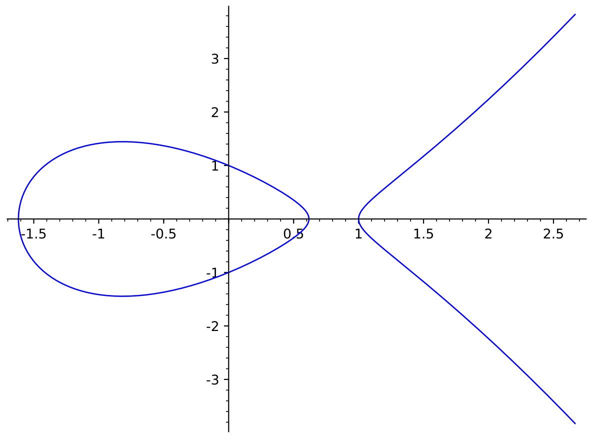
**Affine short Weierstraß form** Probably the least abstract and most straight forward way to introduce elliptic curves for non-mathematicians and beginners is the so called affine representation of a short Weierstraß curve. To see what this is, let $\mathbb{F}$ be a finite field of order $q$ and $a,b \in \mathbb{F}$ two field elements such that $4a^3+27b^2 \mod q \neq 0$. Then a **short Weierstrass elliptic curve** $E(\mathbb{F})$ over $\mathbb{F}$ in its affine representation is the set

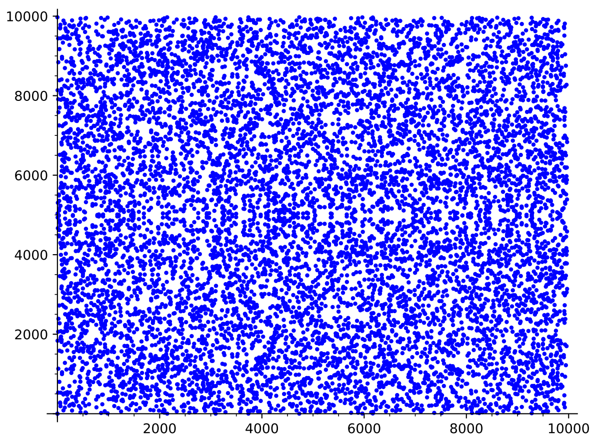
$$E(\mathbb{F}) = \{(x,y) \in \mathbb{F}\times\mathbb{F} \mid y^2 = x^3 + a\cdot x + b\}\bigcup\{\mathcal{O}\} \tag{6.1}$$

of all pairs of field elements $(x,y) \in \mathbb{F}\times\mathbb{F}$, that satisfy the short Weierstrass cubic equation $y^2 = x^3 + a\cdot x + b$, together with a distingushid symbol $\mathcal{O}$, called the **point at infinity**.

**Notation and Symbols 7.** *In the literature, the set $E(\mathbb{F})$, which includs the sybol $\mathcal{O}$ is often called the set of rational points of the elliptic curve, in which case the curve itself is usually written as $E/\mathbb{F}$. However in what follows we will frequently identify an elliptic curve with its set of rational points and therefore use the symbol $E(\mathbb{F})$ instead. This is possible in our case, since we only really care about the group structure of the curve in consideration.*

The term "curve" appears, because in the ordinary 2 dimensional plane $\mathbb{R}^2$, the set of all points $(x,y)$ that satisfy $y^2 = x^3 + a\cdot x + b$ looks like a curve. We should note however, that visualizing elliptic curves over finite fields as "curves" has its limitations and we will therefore not stress the geometric picture too much, but focus on the compuational properties instead. To understand the visual difference, consider the following two elliptic curves:




Both elliptic curves are defined by the same short Weierstraß equation $y^2 = x^3 - 2x + 1$, but the first curve is define in the real affine plane $\mathbb{R}^2$, that is the pair $(x,y)$ contains real numbers, while the second one is defined in the affine plane $\mathbb{F}_{9973}^2$, which means that both $x$ and $y$ are from the prime field $\mathbb{F}_{9973}$. Every blue dot represents a pair $(x,y)$ that is solution to $y^2 = x^3 - 2x + 1$ and as we can see the second curve hardly looks like a geometric structure one would naturally call a curve. So the geometric intuitions from $\mathbb{R}^2$ is kind of obfuscated in curves over finite fields.

The identity $6 \cdot (4a^3 + 27b^2) \bmod q \neq 0$ ensures that the curve is non-singular, which basically means that the curve has no cusps or self-intersections.

When dealing with elliptic curves compuations can quickly become cumbersome and tedious. So on the one hand side the reader is adviced to do as many compuations in a pen and paper style as possible. This helps a lot to get a deeper understanding for the details. On the other hand side however, compuations are sometimes simply to large to be done by hand and one might get lost in the details. Fortunately sage is very helpful in dealing with elliptic curves. It is there a goal of this book to introduce the reader to the great elliptic curve capabilities of sage. One we to define elliptic curves and work is them goes like this:

```
sage: F5 = GF(5) # define the base field
sage: a = F5(2) # parameter a
sage: b = F5(4) # parameter b
sage: # check non-sigularity
sage: F5(6)*(F5(4)*a^3+F5(27)*b^2) != F5(0)
True
sage: # short Weierstrass curve
sage: E = EllipticCurve(F5,[a,b]) # y^2 == x^3 + ax +b
sage: P = E(0,2) # 2^2 == 0^3 + 2*0 + 4
sage: P.xy() # affine coordinates
(0, 2)
sage: INF = E(0) # point at infinity
sage: try:  # point at infinity has no affine coordinates
....:     INF.xy()
....: except ZeroDivisionError:
....:     pass
sage: P = E.plot() # create a plotted version
```

The following three examples will give a more practical understanding of what an elliptic curve is and how we can compute them. The reader is adviced to read them carefully and ideally to parallel the computation themselfs. We will repeatedly build on these example in this chapter and use the second example at various places in this book.

**Example 65.** *To provide the reader with a small example of an elliptic curve, where all computation can e done in a pen and paper style, consider the prime field $\mathbb{F}_5$ from example (XXX). The reader who had worked through the examples and exercises in the previous section knows this prime field well.*

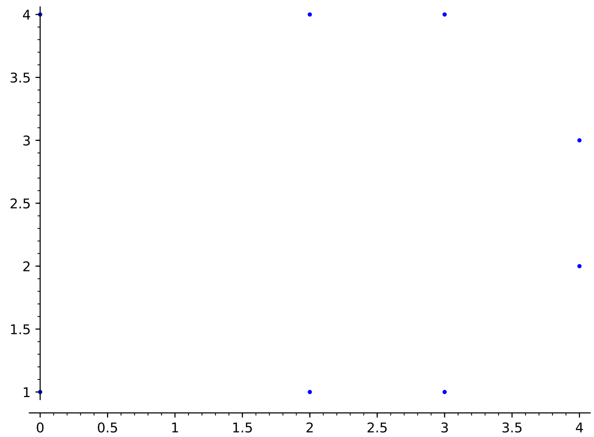
*To define an elliptic curve over $\mathbb{F}_5$, we have to choose to numbers a and b from that field. Assuming we choose $a = 1$ and $b = 1$ then $4a^3 + 27b^2 \equiv 1 \pmod 5$ from which follows that the corresponding elliptic curve $E_1(\mathbb{F}_5)$ is given by the set of all pairs $(x,y)$ from $\mathbb{F}_5$ that satisfy the equation $y^2 = x^3 + x + 1$, together with the special symbol $\mathcal{O}$, which represents the "point at infinity".*

*To get a better understand of that curve, obersever that if we choose arbitrarily the pair $(x,y) = (1,1)$, we see that $1^2 \neq 1^3 + 1 + 1$ and hence $(1,1)$ is not an element of the curve $E_1(\mathbb{F}_5)$. On the other hand chosing for exmple $(x,y) = (2,1)$ gives $1^2 = 2^3 + 2 + 1$ and hence the pair $(2,1)$ is an element of $E_1(\mathbb{F}_5)$ (Remember that all computations are done in modulo 5* *arithmetics).*

*Now since the set $\mathbb{F}_5 \times \mathbb{F}_5$ of all pairs $(x,y)$ from $\mathbb{F}_5$ contains only $5 \cdot 5 = 25$ pairs, we can compute the curve, by just inserting every possible pair $(x,y)$ into the short Weierstraß equation $y^2 = x^3 + x + 1$. If the equation holds, the pair is a curve point, if not that means that the point is not on the curve. Combining the result of this computation with the point at infinity gives the curve as:*

$$E_1(\mathbb{F}_5) = \{\mathcal{O}, (0,1), (2,1), (3,1), (4,2), (4,3), (0,4), (2,4), (3,4)\}$$

*So our elliptic curve is a set of $9$ elements. $8$ of which are pairs of numbers and one special symbol $\mathcal{O}$. Visualizing $E1$ gives:*



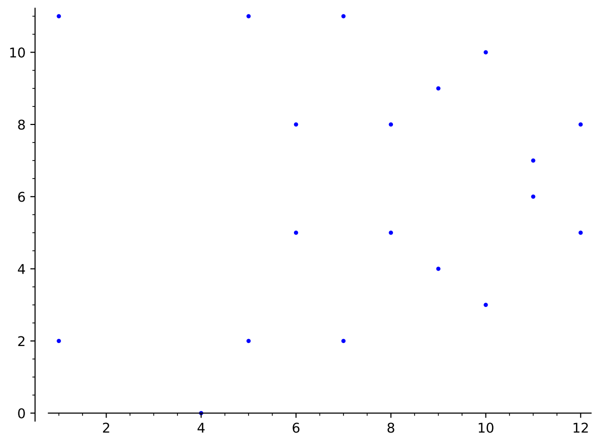

In the development of SNARKS it is sometimes necessary to do elliptic curve cryptograph "in a circuit", which basically means that the elliptic curves needs to be implemented in a certain SNARK-friedly way. We will look at what this means in XXX. To be able to do this efficiently it is desireable to have curves with special properties. The following example is a pen and paper version of such a curve, that parallels the definition of a cryptographically secure curve called *Baby-JubJub* which is extensively used in real world snarks. The interested reader is adviced to read this example carefully as we will use it and build on it in various places throughout the book.

**Example 66** (Pen-JubJub)**.** *Consider the prime field $\mathbb{F}_{13}$ from exercise XXX. If we choose $a = 8$ and $b = 8$ then $4a^3 + 27b^2 \equiv 6 \pmod{13}$ and the corresponding elliptic curve is given by all pairs $(x,y)$ from $\mathbb{F}_13$ such that $y^2 = x^3 + 8x + 8$ holds. We write PJJ_13 for this curve and call it the Pen-JubJub curve.*

*Now since the set $\mathbb{F}_{13} \times \mathbb{F}_{13}$ of all pairs $(x,y)$ from $\mathbb{F}_{13}$ contains only $13 \cdot 13 = 169$ pairs, we can compute the curve, by just inserting every possible pair $(x,y)$ into the short Weierstraß equation $y^2 = x^3 + 8x + 8$. We get*

$$\begin{aligned} PJJ\_13 = \{&\mathcal{O}, (1,2), (1,11), (4,0), (5,2), (5,11), (6,5), (6,8), (7,2), (7,11), \\ &(8,5), (8,8), (9,4), (9,9), (10,3), (10,10), (11,6), (11,7), (12,5), (12,8)\} \end{aligned}$$

*As we can see the curve consist of $20$ points. $19$ points from the affine plane and the point at infinity. To get a visual impression of the PJJ_13 curve, we might plot all of its points (except the point at infinity) in the $\mathbb{F}_{13} \times \mathbb{F}_{13}$ affine plane. We get:*




*As we will see in what follows this curve is kind of special as it is possible to represent it in two alternitive forms, called the Montgomery and the twisted Edwards form (See xxx and XXX).*

Now that we have seen two pen and paper friendly elliptic curves, lets look at a curve that is used in actual cryptography. Cryptographically secure elliptic curve are not qualitatively defferent from the curves we looked at so far. The only difference is that the prime number modulus of the prime field is much larger. Typical examples use prime numbers, which have binary representations in the size of more then double the size of the desired security level. So if for example a security of 128 bit is desired, a prime moduls of binary size $\geq 256$ is choosen. The following example provides such a curve.

**Example 67** (Bitcoin's Secp256k1 curve)**.** *To give an example of a real world, cryptographically secure curve, lets look at curve Secp256k1, which is famous for being used in the public key cryptography of Bitcoin. The prime field $\mathbb{F}_p$ of Secp256k1 if defined by the prime number*

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$

*which has a binary representation that need 256 bits. This implies that the $\mathbb{F}_p$ approximately contains $2^{256}$ many elements. So the underlying field is large. To get an image of how large the base field is, consider that the number $2^{256}$ is approximately in the same order of magnitute as the estimated number of atomes in the observeable universe.*

*Curve Secp256k1 is then defined by the parameters $a, b \in \mathbb{F}_p$ with $a = 0$ and $b = 7$. Since $4 \cdot 0^3 + 27 \cdot 7^2 \bmod p = 1323$, those parameters indeed define an elliptic curve given by*

$$Secp256k1 = \{(x,y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = x^3 + 7\}$$

*Clearly Secp256k1 is a curve, to large to do computations by hand, since it can be shown that Secp256k1 contains*

$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

*many elements, were $r$ is a prime number that also has a binary representation of 256 bits. Cryptographically secure elliptic curves are therefore not useful in pen and paper computations. Fortunately sage handles large curve efficiently:*

```
sage: p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
sage: # Hexadecimal representation
```

```
sage: p.str(16)
fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc
   2f
sage: p.is_prime()
True
sage: p.nbits()
256
sage: Fp = GF(p)
sage: Secp256k1 = EllipticCurve(Fp,[0,7])
sage: r = Secp256k1.order() # number of elements
sage: r.str(16)
fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd03641
   41
sage: r.is_prime()
True
sage: r.nbits()
256
```

**Exercise 35.** *Look-up the definition of curve BLS12-381, implement it in sage and computes its order.*

**Affine compressed representation** As we have seen in example XXX, cryptographically secure elliptic curves are definec over large prime fields, where elements of those fields typically need more then 255 bits storage on a computer. Since elliptic curve points consists of pairs of those field element, they need double that amount of storage.

To reduce the amount of space needed to represent a curve point note however, that up to a sign the $y$-coordinate of a curve point can be computed from the $x$-coordinate, by simply inserting $x$ into the Weierstraß equation and then computing the roots of the result. This gives two results and it follows that we can represent a curve point in **compressed form** by simply storing the $x$-coordinate together with a single sign bit only, the latter of which deterministically decides which of the two roots to choose. In case that the $y$-coordinate is zero, both sign bit give the same result.

For example one convention could be to always choose the root closer to 0, when the sign bit is 0 and the root closer to the order of $\mathbb{F}$ when the sign bit is 1.

**Example 68** (Pen-JubJub)**.** *To understand the concept of compressed curve points a bit better consider the PJJ_13 curve from example XXX again. Since this curve is defined over the prime field $\mathbb{F}_{13}$ and numbers between 0 and 13 need approximately 4 bits to be represented, each PJJ_13 -point need 8-bits of storage in uncompressed form, while it would need only 5 bits in compressed form. To see how this works, recall that in uncompressed form we have*

$$
\begin{aligned}
PJJ\_13 = \{&\mathcal{O},(1,2),(1,11),(4,0),(5,2),(5,11),(6,5),(6,8),(7,2),(7,11),\\
&(8,5),(8,8),(9,4),(9,9),(10,3),(10,10),(11,6),(11,7),(12,5),(12,8)\}
\end{aligned}
$$

*Using the technique of point compression, we can replace the y-ccordinate in each $(x,y)$ pair by a sign bit, indicating, wheather or not $y$ is closer to 0 or to 13. So $y$ values in the range $[0,\ldots,6]$ having sign bit 0 and $y$-values in the range $[7,\ldots,12]$ having sign bit 1. Applying this* *to the points in PJJ_13 gives the compressed representation:*

$$PJJ\_13 = \{\mathcal{O},(1,0),(1,1),(4,0),(5,0),(5,1),(6,0),(6,1),(7,0),(7,1),$$
$$(8,0),(8,1),(9,0),(9,1),(10,0),(10,1),(11,0),(11,1),(12,0),(12,1)\}$$

*Note that the numbers $7,\ldots,12$ are the negatives (additive inverses) of the numbers $1,\ldots,6$ in modular $13$ arithmetics and that $-0 = 0$. Calling the compression bit a "sign bit" therefore makes sense.*

*To recover the uncompressed point of say $(5,1)$, we insert the x-ccordinate $5$ into the Weierstraß equation and get $y^2 = 5^3 + 8\cdot 5 + 8 = 4$. As expected $4$ is a quadratic residue in $\mathbb{F}_{13}$ with roots $\sqrt{4} = \{2,11\}$. Now since the sign bit of the point is $1$, we have to choose the root closer to the modulus $13$ which is $11$. The uncompressed point is therefore $(5,11)$.*

Looking at the previous examples, compression rate looks not very impressive. The followin example therefore looks at the Secp256k1 curve to show that compression is actually useful.

**Example 69.** *Consider the Secp256k1 curve from example XXX again. The following code involves sage to generate a random affine curve point, we then apply our compression method*

```
sage: P = Secp256k1.random_point().xy()
sage: P
(1802010835185979517838070325418112017019563059597668291411379
   3974234835083813, 13223837084754061656794665529570860623892
   5809636771174388128990217564788890 63)
sage: # uncompressed affine point size
sage: ZZ(P[0]).nbits()+ZZ(P[1]).nbits()
507
sage: # compute the compression
sage: if P[1] > Fp(-1)/Fp(2):
....:       PARITY = 1
....: else:
....:       PARITY = 0
sage: PCOMPRESSED = [P[0],PARITY]
sage: PCOMPRESSED
[1802010835185979517838070325418112017019563059597668291411379
   3974234835083813, 0]
sage: # compressed affine point size
sage: ZZ(PCOMPRESSED[0]).nbits()+ZZ(PCOMPRESSED[1]).nbits()
254
```

**Affine group law** One of the key properties of an elliptic curve is that it is possible to define a group law on the set of its rational points, such that the point at infinity serves as the neutral element and inverses are reflections on the $x$-axis.

The origin of this law can be understood in a geometric picture and is known as the *chord-and-tangent rule*. In the affine representation of a short Weierstraß curve, the rule can be described in the following way:

- (Point addition) Let $P,Q \in E(\mathbb{F})\backslash\{\mathcal{O}\}$ with $P \neq Q$ be two distinct points on an elliptic curve, that are both not the point at infinity. Then the sum of $P$ and $Q$ is defined as follows: Consider the line $l$ which intersects the curve in $P$ and $Q$. If $l$ intersects the elliptic curve at a third point $R'$, define the sum $R = P \oplus Q$ of $P$ and $Q$ as the reflection of $R'$ at the $x$-axis. If it does not intersect the curve at a third point define the sum to be the point at infinity $\mathcal{O}$. It can be shown, that no such chord-line will intersect the curve in more then three points, so addition is not ambigious.

- (Point doubling) Let $P \in E(\mathbb{F})\backslash\{\mathcal{O}\}$ be a point on an elliptic curve, that is not the point at infinity. Then the sum of $P$ with itself (the doubling) is defined as follows: Consider the line wich is tangent to the elliptic curve at $P$, if this line intersects the elliptic curve at a second point $R'$. The sum $2P = P + P$ is then the reflection of $R'$ at the $x$-axis. If it does not intersect the curve at a third point define the sum to be the point at infinity $\mathcal{O}$. It can be shown, It can be shown, that no such tangent-line will intersect the curve in more then two points, so addition is not ambigious.

- (Point at infinity) We define the point at infinity $\mathcal{O}$ as the neutral ement of addition, that is we define $P + \mathcal{O} = P$ for all points $P \in E(\mathbb{F})$.

It can be shown that the points of an elliptic curve form a commutative group with respect to the tangent and chord rule, such that $\mathcal{O}$ acts the neutral element and the inverse of any element $P \in E(\mathbb{F})$ is the reflection of $P$ on the $x$-axis.

To translate the geometric description into algebraic equations, first observe that for any two given curve points $(x_1,y_1),(x_2,y_2) \in E(\mathbb{F})$, it can be shown that the identity $x_1 = x_2$ implies $y_2 = \pm y_1$, which shows that the following rules are a complete description of the affine addition law.

- (Neutral element) Point at infinity $\mathcal{O}$ is the neutral element.

- (Additive inverse ) The addivive inverse of $\mathcal{O}$ is $\mathcal{O}$ and for any other curve point $(x,y) \in E(\mathbb{F})\backslash\{\mathcal{O}\}$, the additive inverse is given by $(x,-y)$.

- (Addition rule) For any two curve points $P,Q \in E(\mathbb{F})$ addition is defined by one of the following three cases:
  1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P \oplus Q = P$.
  2. (Adding inverse elements) If $P = (x,y)$ and $Q = (x,-y)$, the sum is defined as $P \oplus Q = \mathcal{O}$.
  3. (Adding non self-inverse equal points) If $P = (x,y)$ and $Q = (x,y)$ with $y \neq 0$, the sum $2P = (x',y')$ is defined by

$$x' = \left(\tfrac{3x^2+a}{2y}\right)^2 - 2x \quad , \quad y' = \left(\tfrac{3x^2+a}{2y}\right)^2 (x - x') - y$$

  4. (Adding non inverse differen points) If $P = (x_1,y_1)$ and $Q = (x_2,y_2)$ such that $x_1 \neq x_2$, the sum $R = P + Q$ with $R = (x_3,y_3)$ is defined by

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

Note that short Weierstraß curve points $P$ with $P = (x,0)$ are inverse to themselfs, which implies $2P = \mathcal{O}$ in this case.

**Notation and Symbols 8.** *Let $\mathbb{F}$ be a field and $E(\mathbb{F})$ be an elliptic curve over $\mathbb{F}$. We write $\oplus$ for the group law on $E(\mathbb{F})$ and $(E(\mathbb{F}),\oplus)$ for the group of rational points.*

As we can see, it is very efficient to compute inverses on elliptic curves. However computing the addition of elliptic curve points in the affine representation needs to consider many cases and involves extensive finite field divisions. As we will see in the next paragraph this can be simplified in projective coordinates.

To get some practical impression of how the group law on an elliptic curve is computed, lets look at some actual cases:

**Example 70.** *Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX again. As we have seen, the set of rational points contains 9 elements and is given by*

$$E_1(\mathbb{F}_5) = \{\mathcal{O},(0,1),(2,1),(3,1),(4,2),(4,3),(0,4),(2,4),(3,4)\}$$

*We know that this set defines a group, so we can add any two elements from $E_1(\mathbb{F}_5)$ to get a third element.*

*To give an example consider the elements $(0,1)$ and $(4,2)$. Neither of these elements is the neutral element $\mathcal{O}$ and since the x-coordinate of $(0,1)$ is different from the x-coordinate of $(4,2)$, we know that we have to use the chord rule, that is rule number 4 from XXX to compute the sum $(0,1)\oplus(4,2)$. We get*

$$\begin{aligned}
x_3 &= \left(\frac{y_2-y_1}{x_2-x_1}\right)^2 - x_1 - x_2 && \text{\# insert points}\\
&= \left(\frac{2-1}{4-0}\right)^2 - 0 - 4 && \text{\# simplify in } \mathbb{F}_5\\
&= \left(\frac{1}{4}\right)^2 + 1 = 4^2 + 1 = 1 + 1 = 2
\end{aligned}$$

$$\begin{aligned}
y_3 &= \left(\frac{y_2-y_1}{x_2-x_1}\right)(x_1 - x_3) - y_1 && \text{\# insert points}\\
&= \left(\frac{2-1}{4-1}\right)(0-2) - 1 && \text{\# simplify in } \mathbb{F}_5\\
&= \left(\frac{1}{4}\right)\cdot 3 + 4 = 4\cdot 3 + 4 = 2 + 4 = 1
\end{aligned}$$

*So in our elliptic curve $E_1(\mathbb{F}_5)$ we get $(0,1)\oplus(4,2) = (2,1)$ and indeed the pair $(2,1)$ is an element of $E_1(\mathbb{F}_5)$ as expected. On the other hand we have $(0,1)\oplus(0,4) = \mathcal{O}$, since both points have equal x-coordinates and inverse y-coordinates rendering them as inverse to each other.*

*Adding the point $(4,2)$ to itself, we have to use the tangent rule, that is rule 3 from XXX. We get*

$$\begin{aligned}
x' &= \left(\frac{3x^2+a}{2y}\right)^2 - 2x && \text{\# insert points}\\
&= \left(\frac{3\cdot 4^2+1}{2\cdot 2}\right)^2 - 2\cdot 4 && \text{\# simplify in } \mathbb{F}_5\\
&= \left(\frac{3\cdot 1+1}{4}\right)^2 + 3\cdot 4 = \left(\frac{4}{4}\right)^2 + 2 = 1+2 = 3
\end{aligned}$$

$$\begin{aligned}
y' &= \left(\frac{3x^2+a}{2y}\right)^2 (x-x') - y && \text{\# insert points}\\
&= \left(\frac{3\cdot 4^2+1}{2\cdot 2}\right)^2 (4-3) - 2 && \text{\# simplify in } \mathbb{F}_5\\
&= 1\cdot 1 + 3 = 4
\end{aligned}$$

*So in our elliptic curve $E_1(\mathbb{F}_5)$ we get the doubling $2\cdot(4,2)$, that is $(4,2)\oplus(4,2) = (3,4)$ and indeed the pair $(3,4)$ is an element of $E_1(\mathbb{F}_5)$ as expected. The group $E_1(\mathbb{F}_5)$ has no self inverse points other then the neutral element $\mathcal{O}$, since no point has $0$ as its y-coordinate. We can invoke sage to double check the computations.*

```
sage: F5 = GF(5)
sage: E1 = EllipticCurve(F5,[1,1])
sage: INF = E1(0) # point at infinity
sage: P1 = E1(0,1)
sage: P2 = E1(4,2)
sage: P3 = E1(0,4)
sage: R1 = E1(2,1)
sage: R2 = E1(3,4)
sage: R1 == P1+P2
True
sage: INF == P1+P3
True
sage: R2 == P2+P2
True
sage: R2 == 2*P2
True
sage: P3 == P3 + INF
True
```

**Example 71** (Pen-JubJub)**.** *Consider the PJJ_13-curve from example XXX again and recall that its group of rational points is given by*

$$\begin{aligned}
PJJ_13 = \{&\mathcal{O},(1,2),(1,11),(4,0),(5,2),(5,11),(6,5),(6,8),(7,2),(7,11),\\
&(8,5),(8,8),(9,4),(9,9),(10,3),(10,10),(11,6),(11,7),(12,5),(12,8)\}
\end{aligned}$$

*In contrast to the group from the previous example, this group contains a self inverse point, which is different from the neutral element, given by $(4,0)$. To see what this means, observe that* *we can not add $(4,0)$ to itself using the tangent rule 3 from XXX, as the y-coordinate is zero. Instead we have to use rule 2, since $0 = -0$. We therefore get $(4,0) \oplus (4,0) = \mathcal{O}$ in PJJ_13. The point $(4,0)$ is therefore inverse to itself, as adding it to itself gives the neutral element.*

```
sage: F13 = GF(13)
sage: MJJ = EllipticCurve(F13,[8,8])
sage: P = MJJ(4,0)
sage: INF = MJJ(0) # Point at infinity
sage: INF == P+P
True
sage: INF == 2*P
True
```

**Example 72.** *Consider the Secp256k1 curve from example XXX again. The following code involves sage to generate a random affine curve point, we then apply our compression method*

```
sage: P = Secp256k1.random_point()
sage: Q = Secp256k1.random_point()
sage: INF = Secp256k1(0)
sage: R1 = -P
sage: R2 = P + Q
sage: R3 = Secp256k1.order()*P
sage: P.xy()
(7721707865487222513509459581338520379311091295732670400641299
    0054569179205123, 553960913395950962941946593355450044623855
    22368079032240696995479053087046612)
sage: Q.xy()
(7409643618109190431684044398639992396573450116458393025381145
    0368725945670910, 325578957542079811335355334140042312884872
    84622288788152010968881537443281894)
sage: (ZZ(R1[0]).str(16), ZZ(R1[1]).str(16))
('aab75156d9008b7176f0757bc2230f9b9df1f0e92c640b7643acc7e2a4
    ddce03', '8586ec64ca93bb9c33cb3e67c46182c0c59bed2564159d1ab
    01f6172cd87805b')
sage: R2.xy()
(2860572976196126894291590597712567902583762293940592241692994
    881927927772427, 678881420135678790732027421257905259057280
    29870262536878372032151036346741086)
sage: R3 == INF
True
sage: P[1]+R1[1] == Fp(0) # -(x,y) = (x,-y)
True
```

**Exercise 36.** *Consider the PJJ_13-curve from example XXX.*

1. *Compute the inverse of $(10,10)$, $\mathcal{O}$, $(4,0)$ and $(1,2)$.*
2. *Compute the expression $3*(1,11)-(9,9)$.*
3. *Solve the equation $x+2(9,4) = (5,2)$ for some $x \in$ PJJ_13*

4. *Solve the equation* $x \cdot (7,11) = (8,5)$ *for* $x \in \mathbb{Z}$

**Scalar multiplication** As we have seen in the previous section, elliptic curves $E(\mathbb{F})$ have the structure of a commutative group associated to them. It can moreover be shown, that this group is finite and cyclic, whenever the field is finite.

To understand the elliptic curve scalar multiplication, recall from XXX that every finite cyclic group of order $q$ has a genenerator $g$ and an associated exponential map $g^{(\cdot)} : \mathbb{Z}_q \to \mathbb{G}$, where $g^n$ is the $n$-fold product of $g$ with itself.

Now, elliptic curve scalar multiplication is then nothing but the exponential map, written in additive notation. To be more precise let $\mathbb{F}$ be a finite field, $E(\mathbb{F})$ an elliptic curve of order $r$ and $P$ a generator of $E(\mathbb{F})$. Then the **elliptic curve scalar multiplication** with base $P$ is given by

$$[\cdot]P : \mathbb{Z}_r \to E(\mathbb{F}); m \mapsto [m]P$$

where $[0]P = \mathcal{O}$ and $[m]P = P + P + \ldots + P$ is the $m$-fold sum of $P$ with itself. Elliptic curve scalar multiplication is therefore nothing but an instantiation of the general exponential map, when using additive instead of multiplicative notation. This map is a homomorphis of groups, which means that $[n+m]P = [n]P \oplus [m]P$.

As with all finite, cyclic groups the inverse of the exponential map exist and is usually called the *elliptic curve discrete logarithm map*. However elliptic curve are believed to be XXX-groups, which means that we don't know of any efficient way to actuall compute this map.

Scalar multiplication and its inverse, the elliptic curve discrete logarithm, define the elliptic curve discrete logarithm *problem*, which consists of finding solutions $m \in \mathbb{Z}_r$, such that

$$P = [m]Q \tag{6.2}$$

holds. Any solution $m$ is usually called a *discrete logarithm* relation between $P$ and $Q$. If $Q$ is a generator of the curve, then there is a discrete logarithm relation between $Q$ and any other point, since $Q$ generates the group by repeatedly adding $Q$ to itself. So for generator $Q$ and point $P$, we know some discrete logarithm relation exist. However since elliptic curves are believed to be XXX-groups, finding actual relations $m$ is computationally hard, with runtimes approximately in the size of the order of the group. It pracis we often need the assumption that a discrete logarithm relation exists, but that at the same time noone knows this relation.

One useful property of the exponential map in regard to the examples in this book, is that it can be used to greatly simplify pen and paper computations. As we have seen in example XXX, computing the elliptic curve addition law takes quit a bit of effort, when done without a computer. However when $g$ is a generator of small pen and paper elliptic curve group of order $r$, we can use the exponential map to write the group as

$$\mathbb{G} = \{[1]g \to [2]g \to [3]g \to \cdots \to [r-1]g \to \mathcal{O}\} \tag{6.3}$$

using cofactor clearing, which implies that $[r]g = \mathcal{O}$. "Logarithmic ordering" like this greatly simplifies complicated elliptic curve addition to the much simpler case of modular $r$ addition. So in order to add two curve points $P$ and $Q$, we only have to look up their discrete log relations with the generator, say $P = [n]g$ and $Q = [m]g$ and compute the sum as $P \oplus Q = [n+m]g$. This is, of course, only possible for small groups which we can organize as in XXX.

In the following example we will look at some implications of the fact that elliptic curves are finite cyclic groups. We will apply the fundamental theorem of finite cyclic groups and look how it reflects on the curves in consideration.

**Example 73.** *Consider the elliptic curve group $E_1(\mathbb{F}_5)$ from example XXX. Since it is a finite cyclic group of order* 9 *and the prime factorization of* 9 *is* $3 \cdot 3$*, we can use the fundamental theorem of finite cyclic groups to reason about all its subgroups. In fact since the only prime factor of* 9 *is* 3*, we know that $E_1(\mathbb{F}_5)$ has the following subgrous:*

- $\mathbb{G}_1 = E_1(\mathbb{F}_5)$ *is a subgroup of order* 9*. By definition any group is a subgroup of itself.*
- $\mathbb{G}_2 = \{(2,1),(2,4),\mathcal{O}\}$ *is a subgroup of order* 3*. This is the subgroup associated to the prime factor* 3*.*
- $\mathbb{G}_3 = \{\mathcal{O}\}$ *is a subgroup of order* 1*. This is the trivial subgroup.*

*Moreover since $E_1(\mathbb{F}_5)$ and all its subgroups are cyclic, we know from XXX, that they must have generators. For example the curve point $(2,1)$ is a generator of the order* 3*-subgroup $\mathbb{G}_2$, since every element of $\mathbb{G}_2$ can be generated, by repeaedly adding $(2,1)$ to itself:*

$$
\begin{aligned}
[1](2,1) &= (2,1)\\
[2](2,1) &= (2,4)\\
[3](2,1) &= \mathcal{O}
\end{aligned}
$$

*Since $(2,1)$ is a generator we know from XXX, that it gives rise to an exponential map from the finite field $\mathbb{F}_3$ onto $\mathbb{G}_2$ defined by scalar multiplication*

$$[\cdot](2,1) : \mathbb{F}_3 \to \mathbb{G}_2 \; : \; x \mapsto [x](2,1)$$

*To give an example of a generator that generates the entire group $E_1(\mathbb{F}_5)$ consider the point $(0,1)$. Applying the tangent rule repeadedly we compute with some efford:*

$$
\begin{array}{rclcrcl}
[0](0,1) & = & \mathcal{O} & \qquad & [1](0,1) & = & (0,1)\\
[2](0,1) & = & (4,2) & & [3](0,1) & = & (2,1)\\
[4](0,1) & = & (3,4) & & [5](0,1) & = & (3,1)\\
[6](0,1) & = & (2,4) & & [7](0,1) & = & (4,3)\\
[8](0,1) & = & (0,4) & & [9](0,1) & = & \mathcal{O}
\end{array}
$$

*Again, since $(2,1)$ is a generator we know from XXX, that it gives rise to an exponential map. However since the group order is not a prime number, the exponential maps, does not map a from any field but from the residue class ring $\mathbb{Z}_9$ only:*

$$[\cdot](0,1) : \mathbb{Z}_9 \to \mathbb{G}_1 \; : \; x \mapsto [x](0,1)$$

*Using the generator $(0,1)$ and its associated exponential map, we can write $E(\mathbb{F}_1)$ i logarithmic order with respect to $(0,1)$ as explained in XXX. We get*

$$E_1(\mathbb{F}_5) = \{(0,1) \to (4,2) \to (2,1) \to (3,4) \to (3,1) \to (2,4) \to (4,3) \to (0,4) \to \mathcal{O}\}$$

*indicating that the first element is a generator and the n-th element is the scalar product of n and the generator. To how this logarithmic orders like this simplify the computations in small elliptic curve groups, consider example XXX again. In that example we use the chord and tangent rule to compute $(0,1) \oplus (4,2)$. Now in the logarithmic order of $E_1(\mathbb{F})$ we can compute that sum much easier, since we can directly see that $(0,1) = [1](0,1)$ and $(4,2) = [2](0,1)$. We can then deduce $(0,1) \oplus (4,2) = (2,1)$ immediately, since $[1](0,1) \oplus [2](0,1) = [3](0,1) = (2,1)$.*

*To give another example, we can immediately see that $(3,4) \oplus (4,3) = (4,2)$, without doing any expensive elliptic curve addition, since we know $(3,4) = [4](0,1)$ as well as $(4,3) = [7](0,1)$ from the logarithmic representation of $E_1(\mathbb{F}_5)$ and since $4+7=2$ in $\mathbb{Z}_9$, the result must be $[2](0,1) = (4,2)$.*

*Finally we can use $E_1(\mathbb{F}_5)$ as an example to understand the concept of cofactor clearing from XXX. Since the order of $E_1(\mathbb{F}_5)$ is 9 we only have a single factor, which happen to be the cofactor as well. Cofactor clearing then implies that we can map any element from $E_1(\mathbb{F}_5)$ onto its prime factor group $\mathbb{G}_2$ by scalar multiplication with 3. For example taking the element $(3,4)$ which is not in $\mathbb{G}_2$ and multiplying it with 3, we get $[3](3,4) = (2,1)$, which is an element of $\mathbb{G}_2$ as expected.*

In the following example we will look at the subgroups of our pen-jubjub curve, define generators and compute the logarithmic order for pen and paper computations. Then we have anothef look at the principle of cofactor clearing.

**Example 74.** *Consider the pen-jubjub curve PJJ_13 from example XXX again. Since the order of PJJ_13 is 20 and the prime factorization of 20 is $2^2 \cdot 5$, we know that the PJJ_13 contains a "large" prime order subgroup of size 5 and a small prime oder subgroup of size 2.*

*To compute those groups we can apply the technique of cofactor clearing in a try and repeat loop. We start the loop by arbitrarily choose an element $P \in PJJ\_13$. Then we multiply that element with the cofactor of the group, we want to compute. If the result is $\mathcal{O}$, we try a different element and repeat the process until the result is different from the point at infinity.*

*To compute a generator for the small prime order subgroup $(PJJ\_13)_2$, first observe that the cofactor is 10, since $20 = 2 \cdot 10$. We then arbitrarily choose the curve point $(5,11) \in PJJ\_13$ and compute $[10](5,11) = \mathcal{O}$. Since the result is the point at infinity, we have to try another curve point, say $(9,4)$. We get $[10](9,4) = (4,0)$ and we can deduce that $(4,0)$ is a generator of $(PJJ\_13)_2$. Logarithmic order of then gives*

$$(PJJ\_13)_2 = \{(4,0) \to \mathcal{O}\}$$

*as expected, since we know from example XXX that $(4,0)$ is self inverse, with $(4,0) \oplus (4,0) = \mathcal{O}$. Double checking the computations using sage:*

```
sage: F13 = GF(13)
sage: PJJ = EllipticCurve(F13,[8,8])
sage: P = PJJ(5,11)
sage: INF = PJJ(0)
sage: 10*P == INF
True
sage: Q = PJJ(9,4)
sage: R = PJJ(4,0)
sage: 10*Q == R
True
```

*We can apply the same reasoning to the "large" prime order subgroup $(PJJ\_13)_5$, which contains 5 elements. To compute a generator for this group, first observe that the associated cofactor is 4, since $20 = 5 \cdot 4$. We choose the curve point $(9,4) \in PJJ\_13$ again and compute $[4](9,4) = (7,11)$ and we can deduce that $(7,11)$ is a generator of $(PJJ\_13)_5$. Using the gen-*

*erator* $(7,11)$*, we compute the exponential map* $[\cdot](7,11) : \mathbb{F}_5 \to$ *PJJ_13 and get*

$$\begin{aligned}
[0](7,11) &= \mathcal{O} \\
[1](7,11) &= (7,11) \\
[2](7,11) &= (8,5) \\
[3](7,11) &= (8,8) \\
[4](7,11) &= (7,2)
\end{aligned}$$

*We can use this computation to write the large order prime group* $(PJJ\_13)_5$ *of the pen-jubjub curve in logarithmic order, which we will use quite frequently in what follows. We get:*

$$(PJJ\_13)_5 = \{(7,11) \to (8,5) \to (8,8) \to (7,2) \to \mathcal{O}\}$$

*From this, we can immediately see that for example* $(8,8) \oplus (7,2) = (8,5)$*, since* $3+4=2$ *in* $\mathbb{F}_5$.

From the previous two examples, the reader might get the impression, that elliptic curve computation can be largely replaced by modular arithmetics. This however is not true in general, but only an arefact of small groups where it is possible to write the entire group in a logarithmic order. The following example gives some understanding, why this is not possible in cryptographically secure groups

**Example 75.** *SEKTP BICOIN. DISCRET LOG HARDNESS PROHIBITS ADDITION IN THE FIELD...*

**Projective short Weierstraß form** As we have seen in the previous section, describing elliptic curves as pairs of points that satisfy a certain equation is relatively straight forward. However in order to define a group structure on the set of points, we had to add a special point at infinity to act as the neutral element.

Recalling from the definition of projective planes XXX we know, that points at infinity are handled as ordinary points in projective geometry. It make therefore sense to look at the definition of a short Weierstraß curve in projective geometry.

To see what a short Weierstraß curve in projective coordinates is, let $\mathbb{F}$ be a finite field of order $q$ and characteristic $> 3$, $a, b \in \mathbb{F}$ two field elements such that $4a^3 + 27b^2 \bmod q \neq 0$ and $\mathbb{F}\mathrm{P}^2$ the projective plane over $\mathbb{F}$. Then a **short Weierstrass elliptic curve** over $\mathbb{F}$ in its projective representation is the set

$$E(\mathbb{F}\mathrm{P}^2) = \{[X : Y : Z] \in \mathbb{F}\mathrm{P}^2 \mid Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3\} \tag{6.4}$$

of all points $[X : Y : Z] \in \mathbb{F}\mathrm{P}^2$ from the projective plane, that satisfy the *homogenous* cubic equation $Y^2 \cdot Z = X^3 + a \cdot X \cdot Z^2 + b \cdot Z^3$.

To understand how the point at infinity is unified in this definition, recall from XXX that, in projective geometry points at infinity are given by homogeneous coordinates $[X : Y : 0]$. Inserting representatives $(x_1, y_1, 0) \in [X : Y : 0]$ from those classes into the defining homogenous cubic equations gives

$$\begin{aligned}
y_1^2 \cdot 0 &= x_1^3 + a \cdot x_1 \cdot 0^2 + b \cdot 0^3 && \Leftrightarrow \\
0 &= x_1^3
\end{aligned}$$

which shows that the only point at infinity that is also a point on a projective short Weierstraß curve is the class

$$[0,1,0] = \{(0,y,0) \mid y \in \mathbb{F}\}$$

This point is the projective representation of $\mathcal{O}$. The projective representation of a short Weierstraß curve therefore has the advantage to not need a special symbol to represent the point at infinity $\mathcal{O}$ from the affine definition.

**Example 76.** *To get an intuition of how an elliptic curve in projective geometry looks, consider curve $E_1(\mathbb{F}_5)$ from example (XXX). We know that in its affine representation, the set of rational points is given by*

$$E_1(\mathbb{F}_5) = \{\mathcal{O},(0,1),(2,1),(3,1),(4,2),(4,3),(0,4),(2,4),(3,4)\}$$

*which is defined as the set of all pairs $(x,y) \in \mathbb{F}_5 \times \mathbb{F}_5$, such that the affine short Weierstrass equation $y^2 = x^3 + ax + b$ with $a = 1$ and $b = 1$ is satisfied.*

*To finde the projective representation of a short Weierstrass curve with the same parameters $a = 1$ and $b = 1$, we have to compute the set of projective points $[X : Y : Z]$ from the projective plane $\mathbb{F}_5\mathrm{P}^2$, that satisfy the homogenous cubic equation*

$$y_1^2 z_1 = x_1^3 + 1 \cdot x_1 z_1^2 + 1 \cdot z_1^3$$

*for any representative $(x_1, y_1, z_1) \in [X : Y : Z]$. We know from XXX, that the projective plane $\mathbb{F}_5\mathrm{P}^2$ contains $5^2 + 5 + 1 = 31$ elements, so we can take the effort and inseret all elements into equation XXX and see if both sides match.*

*For example, consider the projective point $[0 : 4 : 1]$. We know from XXX, that this point in the projective plane represents the line*

$$[0:4:1] = \{(0,0,0),(0,4,1),(0,3,2),(0,2,3),(0,1,4)\}$$

*in the three dimensional space $\mathbb{F}^3$. To check whether or not $[0 : 4 : 1]$ satisfies XXX, we can insert any representative, that is we can insert any element from XXX. Each element satisfies the equation if and only if any other satisfies the equation. So we insert $(0,4,1)$ and get*

$$1^2 \cdot 1 = 0^3 + 1 \cdot 0 \cdot 1^2 + 1 \cdot 1^3$$

*which tells us that the affine point $[0 : 4 : 1]$ is indeed a solution. And as we can see, would just as well insert any other representative. For example inserting $(0,3,2)$ also satisfies XXX, since*

$$3^2 \cdot 2 = 0^3 + 1 \cdot 0 \cdot 2^2 + 1 \cdot 2^3$$

*To find the projective reprresentation of $E_1$, we first observe that the projective line at infinity $[1 : 0 : 0]$ is not a curve point on any projective short Weierstraß curve since it can not satisfy XXX for any parameter a and b. So we can exclude it from our consideration.*

*Moreover a point at infinity $[X : Y : 0]$ can only satisfy equation XXX for any a and b, if $X = 0$, which implies that the only point at infinity relavant for short Weierstrass elliptic curves is $[0 : 1 : 0]$, since $[0 : k : 0] = [0 : 1 : 0]$ for all k from the finite field. So we can exclude all points at infinity except the point $[0 : 1 : 0]$.*

*So all points that remain are the affine points $[X : Y : 1]$. Inserting all of them into XXX we get the set of all projective curve points as*

$$E_1(\mathbb{F}_5\mathrm{P}^2) = \{[0:1:0],[0:1:1],[2:1:1],[3:1:1], \\ [4:2:1],[4:3:1],[0:4:1],[2:4:1],[3:4:1]\}$$

*If we compare this with the affine representation we see that there is a 1:1 correspondence between the points in the affine representation XXX and the affine points in projective geometry and that the point* $[0:1:0]$ *represents the additional point* $\mathcal{O}$ *in the projective representation.*

**Exercise 37.** *Compute the projective representation of the pen-jubjub curve and the logarithmic order of its large prime order subgroup with respect to the generator* $(7,11)$.

**Projective Group law** As we have seen in XXX, one of the key properties of an elliptic curve is that it comes with a definition of a group law on the set of its rational points, described geometrically by the chord and tangent rule. This rule was kind of intuitive, with the exception of the distinguished point at infinity, which appered whenever the chord or the tangent did not have a third intersection point with the curve.

One of the key features of projective coordinates is now, that in projective space it is guranteed that any chord will always intersect the curve in three points and any tangent will intersect in two points including the tangent point. So the geometric picture simplifies as we don't need to consider external symbols and associated cases.

Again, it can be shown that the points of an elliptic curve in projective space form a commutative group with respect to the tangent and chord rule, such that the projective point $[0:1:0]$ is the neutral element and the additive inverse of a point $[X:Y:Z]$ is given by $[X:-Y:Z]$. The addition law is then usually described by the following algorithm, that minimizes the number of needed additions and multiplications in the base field.

**Exercise 38.** *Compare that affine addition law for short Weierstraß curves with the projective addition rule. Which branch in the projective rule corresponds to which case in the affine law?*

**Coordinate Transformations** As we have seen in example XXX, there was a close relation between the affine and the projective representation of a short Weierstrass curve. This was no accident. In fact from a mathematical point of view projective and affine short Weierstraß curves describe the same thing as there is a one-to-one correspondence (an isomorphism) between both representations for any given parameters $a$ and $b$.

To specify the isomorphism, let $E(\mathbb{F})$ and $E(\mathbb{F}\mathrm{P}^2)$ be an affine and a projective short Weierstraß curve defined for the same parameters $a$ and $b$. Then the map

$$\Phi : E(\mathbb{F}) \to E(\mathbb{F}\mathrm{P}^2)\ :\ \begin{array}{lcl} (x,y) & \mapsto & [x:y:1] \\ \mathcal{O} & \mapsto & [0:1:0] \end{array} \tag{6.5}$$

maps points from a the affine representation to points from the projective representation of a short Weierstraß curve, that is if the pair of points $(x,y)$ saisfies the affine equation $y^2 = x^3 + ax + b$, then all homogeneous coordinates $(x_1,y_1,z_1) \in [x:y:1]$ satisfy the projective equation $y_1^2 \cdot z_1 = x_1^3 + ay_1 \cdot z_1^2 + b \cdot z_1^3$. The inverse is given by the map

$$\Phi^{-1} : E(\mathbb{F}\mathbb{P}^2) \to E(\mathbb{F})\ :\ [X:Y:Z] \mapsto \begin{cases} (\frac{X}{Z}, \frac{Y}{Z}) & \text{if } Z \neq 0 \\ \mathcal{O} & \text{if } Z = 0 \end{cases} \tag{6.6}$$

Note the only projective point $[X:Y:Z]$ with $Z \neq 0$ that satisfys XXX is given by the class $[0:1:0]$.

One key feature of $\Phi$ and its inverse is, that it respects the group structure, which means that $\Phi((x_1,y_1) \oplus (x_2,y_2))$ is equl to $\Phi(x_1,y_1) \oplus \Phi(x_2,y_2)$. The same holds true for the inverse map $\Phi^{-1}$.

Maps with these properties are caled *group isomorphisms* and from a mathematical point of view the exisrence of $\Phi$ implies, that both definition are equivalent and implementations can choose freely between both representations.

**Algorithm 6** Projective Weierstraß Addition Law

**Require:** $[X_1 : Y_1 : Z_1], [X_2 : Y_2 : Z_2] \in E(\mathbb{FP}^2)$

```
procedure ADD-RULE([X1 : Y1 : Z1], [X2 : Y2 : Z2])
    if [X1 : Y1 : Z1] == [0 : 1 : 0] then
        [X3 : Y3 : Z3] ← [X2 : Y2 : Z2]
    else if [X2 : Y2 : Z2] == [0 : 1 : 0] then
        [X3 : Y3 : Z3] ← [X1 : Y1 : Z1]
    else
        U1 ← Y2 · Z1
        U2 ← Y1 · Z2
        V1 ← X2 · Z1
        V2 ← X1 · Z2
        if V1 == V2 then
            if U1 ≠ U2 then [X3 : Y3 : Z3] ← [0 : 1 : 0]
            else
                if Y1 == 0 then [X3 : Y3 : Z3] ← [0 : 1 : 0]
                else
                    W ← a · Z1^2 + 3 · X1^2
                    S ← Y1 · Z1
                    B ← X1 · Y1 · S
                    H ← W^2 − 8 · B
                    X′ ← 2 · H · S
                    Y′ ← W · (4 · B − H) − 8 · Y1^2 · S^2
                    Z′ ← 8 · S^3
                    [X3 : Y3 : Z3] ← [X′ : Y′ : Z′]
                end if
            end if
        else
            U = U1 − U2
            V = V1 − V2
            W = Z1 · Z2
            A = U^2 · W − V^3 − 2 · V^2 · V2
            X′ = V · A
            Y′ = U · (V^2 · V2 − A) − V^3 · U2
            Z′ = V^3 · W
            [X3 : Y3 : Z3] ← [X′ : Y′ : Z′]
        end if
    end if
    return [X3 : Y3 : Z3]
end procedure
```

**Ensure:** $[X_3 : Y_3 : Z_3] == [X_1 : Y_1 : Z_1] \oplus [X_2 : Y_2 : Z_2]$

### 6.1.2 Montgomery Curves

History and use of them (otimized scalar multiplication)

**Affine Montgomery Form** To see what a Montgomery curve in affine coordinates is, let $\mathbb{F}$ be a finite field of characteristic $> 2$ and $A, B \in \mathbb{F}$ two field elements such that $B \neq 0$ and $A^2 \neq 4$. Then a **Montomery elliptic curve** $M(\mathbb{F})$ over $\mathbb{F}$ in its affine representation is the set

$$M(\mathbb{F}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid B \cdot y^2 = x^3 + A \cdot x^2 + x\} \bigcup \{\mathcal{O}\} \tag{6.7}$$

of all pairs of field elements $(x,y) \in \mathbb{F} \times \mathbb{F}$, that satisfy the Montgomery cubic equation $B \cdot y^2 = x^3 + A \cdot x^2 + x$, together with a distingushid symbol $\mathcal{O}$, called the **point at infinity**.

Despite the fact that Montgomery curves look different then short Weierstrass curve, they are in fact just a special way to describe certain short Weierstrass curves. In fact every curve in affine Montgomery form can be transformed into an elliptic curve in Weierstrass form. To see that assume that a curve in Montgomery form $By^2 = x^3 + Ax^2 + x$ is given. The associated Weierstrass form is then

$$y^2 = x^3 + \frac{3 - A^2}{3B^2} \cdot x + \frac{2A^3 - 9A}{27B^3}$$

On the other hand, an elliptic curve $E(\mathbb{F})$ over base field $\mathbb{F}$ in Weierstrass form $y^2 = x^3 + ax + b$ can be converted to Montgomery form if and only if the following conditions hold:

- The number of points on $E(F)$ is divisible by 4
- The polynomial $z^3 + az + b \in \mathbb{F}[z]$ has at least one root $z_0 \in \mathbb{F}$
- $3z_0^2 + a$ is a quadratic residue in $\mathbb{F}$.

When these conditions are satisfied, then for $s = (\sqrt{3z_0^2 + a})^{-1}$ the equivalent Montgomery curve is defined by the equation

$$sy^2 = x^3 + (3z_0 s)x^2 + x$$

If those properties are meet it is therefore possible to transform certain Weierstrass curve into Montgomery form. In the following example we will look at our pen-jubjub curve again and show that it is actually a Montgomery curve.

**Example 77.** *Consider the prime field $\mathbb{F}_{13}$ and the pen-jubjub curve PJJ_13 from example XXX. To see that it is a Montgomery curve, we have to check the properties from XXX:*

*Since the order of PJJ_13 is 20, which is divisible by 4, the first requirement is meet. Next, since $a = 8$ and $b = 8$, we have check if the polynomial $P(z) = z^3 + 8z + 8$ has a root in $\mathbb{F}_{13}$. We simply evaluate $P$ at all numbers $z \in \mathbb{F}_{13}$ a find that $P(4) = 0$, so a root is given by $z_0 = 4$. In a last step we have to check, that $3 \cdot z_0^2 + a$ has a root in $\mathbb{F}_{13}$. We compute*

$$\begin{aligned} 3z_0^2 + a &= 3 \cdot 4^2 + 8 \\ &= 3 \cdot 3 + 8 \\ &= 9 + 8 \\ &= 4 \end{aligned}$$

*To see if 4 is a quadratic residue, we can use Eulers criterium XXX to compute the Legendre symbol of 4. We get:*

$$\left(\frac{4}{13}\right) = 4^{\frac{13-1}{2}} = 4^6 = 1$$

*so 4 indeed has a root in $\mathbb{F}_{13}$. In fact computing a root of 4 in $\mathbb{F}_{13}$ is easy, since the integer root 2 of 4 is also one of its roots in $\mathbb{F}_{13}$. The other root is given by $13-4=9$.*

*Now since all requiremts are meet, we have shown that PJJ_13 is indeed a Montgomery curve and we can use XXX to compute its associated Montgomery form. We compute*

$$\begin{aligned}
s &= \left(\sqrt{3\cdot z_0^2+8}\right)^{-1} \\
&= 2^{-1} && \text{\# Fermat's little theorem} \\
&= 2^{13-2} && \text{\# } 2048 \text{ mod } 13 = 7 \\
&= 7
\end{aligned}$$

*The defining equation for the Montgomery form of our pen-jubjub curve is then given by the following equation*

$$\begin{aligned}
sy^2 &= x^3 + (3z_0s)x^2 + x && \Rightarrow \\
7\cdot y^2 &= x^3 + (3\cdot 4\cdot 7)x^2 + x && \Leftrightarrow \\
7\cdot y^2 &= x^3 + 6x^2 + x
\end{aligned}$$

*So we get the defining parameters as $B=7$ and $A=6$ and we can write the pen-jubjub curve in its affine Montgomery representation as*

$$PJJ\_13 = \{(x,y)\in\mathbb{F}_{13}\times\mathbb{F}_{13} \mid 7\cdot y^2 = x^3+6x^2+x\}\bigcup\{\mathcal{O}\}$$

*Now that we have the abstract definition of our pen-jubjub curve in Montgomery form, we can compute the set of points, by inserting all pairs $(x,y)\in\mathbb{F}_{13}\times\mathbb{F}_{13}$ similar to how we computed the curve points in its Weierstraß representation. We get*

$$\begin{aligned}
PJJ_13 = \{&\mathcal{O},(0,0),(1,4),(1,9),(2,4),(2,9),(3,5),(3,8),(4,4),(4,9),\\
&(5,1),(5,12),(7,1),(7,12),(8,1),(8,12),(9,2),(9,11),(10,3),(10,10)\}
\end{aligned}$$

```
sage: F13 = GF(13)
sage: L_MPJJ = []
....: for x in F13:
....:     for y in F13:
....:         if F13(7)*y^2 == x^3 + F13(6)*x^2 +x:
....:             L_MPJJ.append((x,y))
sage: MPJJ = Set(L_MPJJ)
sage: # does not compute the point at infinity
```

**Affine Montgomery coordinate transformation** Comparing the Montogomery representation of the previous example with the Weierstraß representation of the same curve, we see that there is a 1:1 correspondence between the curve points in both examples. This is no accident. In fact if $M_{A,B}$ is a Montgomery curve and $E_{a,b}$ a Weierstraß curve with $a = \frac{3-A^2}{3B^2}$ and $b = \frac{2A^2-9A}{27B^3}$ then the function

$$\Phi : M_{A,B} \to E_{a,b} \; : \; (x,y) \mapsto \left(\frac{3x+A}{3B}, \frac{y}{B}\right) \tag{6.8}$$

maps all points in Montgomery representation onto the points in Weierstraß representation. This map is a 1:1 correspondence (am isomorphism) and its inverse map is given by

$$\Phi^{-1} : E_{a,b} \to M_{A,B} \; : \; (x,y) \mapsto (s \cdot (x - z_0), s \cdot y) \tag{6.9}$$

where $z_0$ is a root of the polynomial $z^3 + az + b \in \mathbb{F}[z]$ and $s = (\sqrt{3z_0^2 + a})^{-1}$. Using this map, it is therefore possible for implementations of Montgomery curves to freely transit between the Weierstraß and the Montgomery representation. Note however that according to XXX not every Weierstraß curve is a Montgomery curve, as all of the properties from XXX have to be satisfied. The map $\Phi^{-1}$ therefore does not always exists.

**Example 78.** *Consider our pen-jubjub curve again. In example XXX we derive its Weierstraß representation and in example XXX we derive its Montgomery representation.*

*To see how the coordinate transformation $\Phi$ works in this example, lets map points from the Montogomery representation onto points from the Weierstraß representation. Inserting for example the point $(0,0)$ from the Montomery representation XXX into $\Phi$ gives*

$$\begin{aligned}
\Phi(0,0) &= \left(\frac{3 \cdot 0 + A}{3B}, \frac{0}{B}\right) \\
&= \left(\frac{3 \cdot 0 + 6}{3 \cdot 7}, \frac{0}{7}\right) \\
&= \left(\frac{6}{8}, 0\right) \\
&= (4,0)
\end{aligned}$$

*So the Montgomery point $(0,0)$ maps to the self inverse point $(4,0)$ of the Weierstraß representation. On the other hand we can use our computations of $s = 7$ and $z_0 = 4$ from XXX, to compute the inverse map $\Phi^{-1}$, which maps point on the Weiertraß representation to points on the Mongomery form. Inserting for example $(4,0)$ we get*

$$\begin{aligned}
\Phi^{-1}(4,0) &= (s \cdot (4 - z_0), s \cdot 0) \\
&= (7 \cdot (4-4), 0) \\
&= (0,0)
\end{aligned}$$

*So as expected, the inverse maps maps the Weierstraß point back to where it came from on the Montgomery form. We can invoke sage to proof that our computation of $\Phi$ is correct:*

```
sage: # Compute PHI of Montgomery form:
sage: L_PHI_MPJJ = []
```

```
sage: for (x,y) in L_MPJJ: # LMJJ as defined previously
....:     v = (F13(3)*x + F13(6))/(F13(3)*F13(7))
....:     w = y/F13(7)
....:     L_PHI_MPJJ.append((v,w))
sage: PHI_MPJJ = Set(L_PHI_MPJJ)
sage: # Computation Weierstrass form
sage: C_WPJJ = EllipticCurve(F13,[8,8])
sage: L_WPJJ = [P.xy() for P in C_WPJJ.points() if P.order() > 1]
sage: WPJJ = Set(L_WPJJ)
sage: # check PHI(Montgomery) == Weierstrass
sage: WPJJ == PHI_MPJJ
True
sage: # check the inverse map PHI^(-1)
sage: L_PHIINV_WPJJ = []
sage: for (v,w) in L_WPJJ:
....:     x = F13(7)*(v-F13(4))
....:     y = F13(7)*w
....:     L_PHIINV_WPJJ.append((x,y))
sage: PHIINV_WPJJ = Set(L_PHIINV_WPJJ)
sage: MPJJ == PHIINV_WPJJ
True
```

**Montgomery group law** So we see that Montgomery curves a special cases of short Weierstrass curves. As such they have a group structure defined on the set of their points, which can also be derived from a chord and tangent rule. In accordance with short Weierstrass curves, it can be shown that the identity $x_1 = x_2$ implies $y_2 = \pm y_1$, which shows that the following rules are a complete description of the affine addition law.

- (Neutral element) Point at infinity $\mathcal{O}$ is the neutral element.
- (Additive inverse ) The addivive inverse of $\mathcal{O}$ is $\mathcal{O}$ and for any other curve point $(x,y) \in M(\mathbb{F}_q)\backslash\{\mathcal{O}\}$, the additive inverse is given by $(x,-y)$.
- (Addition rule) For any two curve points $P,Q \in M(\mathbb{F}_q)$ addition is defined by one of the following cases:
  1. (Adding the neutral element) If $Q = \mathcal{O}$, then the sum is defined as $P+Q = P$.
  2. (Adding inverse elements) If $P = (x,y)$ and $Q = (x,-y)$, the sum is defined as $P+Q = \mathcal{O}$.
  3. (Adding non self-inverse equal points) If $P = (x,y)$ and $Q = (x,y)$ with $y \neq 0$, the sum $2P = (x',y')$ is defined by

     $$x' = (\tfrac{3x_1^2+2Ax_1+1}{2By_1})^2 \cdot B - (x_1+x_2) - A \quad , \quad y' = \tfrac{3x_1^2+2Ax_1+1}{2By_1}(x_1 - x') - y_1$$

  4. (Adding non inverse differen points) If $P = (x_1,y_1)$ and $Q = (x_2,y_2)$ such that $x_1 \neq x_2$, the sum $R = P+Q$ with $R = (x_3,y_3)$ is defined by

     $$x' = (\tfrac{y_2-y_1}{x_2-x_1})^2 B - (x_1+x_2) - A \quad , \quad y' = \tfrac{y_2-y_1}{x_2-x_1}(x_1 - x') - y_1$$

### 6.1.3 Twisted Edwards Curves

As we have seen in XXX both Weierstrass and Montgomery curves have somewhat complicated addition and doubling laws as many cases have to be distinguished. Those cases translate to branches in computer programs.

In the context of SNARK development two compuational models for bounded computations, called *circuits* and *rank-1 constraint systems*, are used and program-branches are undesireably costly, when implemented in those models. It is therefore advantageous to look for curves with an addition/doubling rule, that requires no branches and as few field operations as possible.

Twisted Edwards curves are particular useful here as a subclass of these curves has a compact and easy to implement addition law that works for all point, including the point at infinity. Implementing that rule therefore needs no branching.

**Twisted Edwards Form** To see what an affine **twisted Edwards curve** looks like, let $\mathbb{F}$ be a finite field of characteristic $> 2$ and $a,d \in \mathbb{F}\backslash\{0\}$ two non zero field elements with $a \neq d$. Then a **twisted Edwards ellipti curve** in its affine representation is the set

$$E(\mathbb{F}) = \{(x,y) \in \mathbb{F}\times\mathbb{F} \mid a\cdot x^2 + y^2 = 1 + d\cdot x^2y^2\} \tag{6.10}$$

of all pairs $(x,y)$ from $\mathbb{F}\times\mathbb{F}$, that satisfy the twisted Edwards equation $a\cdot x^2 + y^2 = 1 + d\cdot x^2y^2$. A twisted Edwards curve is called an Edwards curve (non twisted), if the parameter $a$ is equal to 1 and is called a **snark friendly** twisted Edwards curve if the parameter $a$ is a quadratic residue and the parameter $d$ is a quadratic non residue.

As we can see from the definition, affine twisted Edwards curve look somewhat different from Weierstraß curves as their affine representation does not need a special symbol to represent the point at infinity. In fact we we will see that the pair $(0,1)$ is always a point on any twisted Edwards curve and that it takes the role of the point at infinity.

Despite the different looks however, twisted Edwards curves are equivalent to Montgomery curves in the sense that for every twisted Edwards curve there is a Montgomery curve and a way to map the points of one curve in a 1:1 correspondence onto the other and vice versa. To see that assume that a curve in twisted Edwards form $a\cdot x^2 + y^2 = 1 + d\cdot x^2y^2$ is given. The associated Montgomery curve is then defined by the Montgomery equation

$$\frac{4}{a-d}y^2 = x^3 + \frac{2(a+d)}{a-d}\cdot x^2 + x \tag{6.11}$$

On the other hand a Montgomery curve $By^2 = x^3 + Ax^2 + x$ with $B \neq 0$ and $A^2 \neq 4$ can gives rise to a twisted Edwards curve defined by the equation

$$(\frac{A+2}{B})x^2 + y^2 = 1 + (\frac{A-2}{B})x^2y^2 \tag{6.12}$$

Recalling from XXX that Montgomery curves are just a special class of Weierstraß, we now know that twisted Edwards curve are special Weierstraß curves too. So the more general way to describe elliptic curves are Weierstraß curves.

**Example 79.** *Consider the pen jubjub curve from example XXX again. We know from XXX that it is a Montgomery curve and since Montgomery curves are equivalent to twisted Edwards curve, we want to write that curve in twisted Edwards form. We use XXX and compute the* *parameters a and d as*

$$\begin{aligned}
a &= \frac{A+2}{B} && \text{\# insert A=6 and B=7}\\
&= \frac{8}{7} = 3 && \# \, 7^{-1} = 2
\end{aligned}$$

$$\begin{aligned}
d &= \frac{A-2}{B}\\
&= \frac{4}{7} = 8
\end{aligned}$$

*So we get the defining parameters as $a = 3$ and $d = 8$. Since our goal is to use this curve later on in implementations of pen and paper snarks, lets show that tiny-jubjub is moreover a snark friendly twisted Edwards curve. To see that, we have to show that a is a quadratic residue and d is a quadratic non residue. We therefore compute the Legendre symbols of a and d using the Euler criterium. We get*

$$\begin{aligned}
\left(\frac{3}{13}\right) &= 3^{\frac{13-1}{2}}\\
&= 3^6 = 1
\end{aligned}$$

$$\begin{aligned}
\left(\frac{8}{13}\right) &= 8^{\frac{13-1}{2}}\\
&= 8^6 = 12 = -1
\end{aligned}$$

*which proofs that tiny-jubjub is snark friendly. We can write the tiny-jubjub curve in its affine twisted Edwards representation as*

$$TJJ\_13 = \{(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13} \mid 3 \cdot x^2 + y^2 = 1 + 8 \cdot x^2 \cdot y^2\}$$

*Now that we have the abstract definition of our pen-jubjub curve in twisted Edwards form, we can compute the set of points, by inserting all pairs $(x,y) \in \mathbb{F}_{13} \times \mathbb{F}_{13}$ similar to how we computed the curve points in its Weierstraß or Edwards representation. We get*

$$\begin{aligned}
PJJ_13 = \{&(0,1),(0,12),(1,2),(1,11),(2,6),(2,7),(3,0),(5,5),(5,8),(6,4),\\
&(6,9),(7,4),(7,9),(8,5),(8,8),(10,0),(11,6),(11,7),(12,2),(12,11)\}
\end{aligned}$$

```
sage: F13 = GF(13)
sage: L_EPJJ = []
....: for x in F13:
....:     for y in F13:
....:         if F13(3)*x^2 + y^2 == 1+ F13(8)*x^2*y^2:
....:             L_EPJJ.append((x,y))
sage: EPJJ = Set(L_EPJJ)
```

**Twisted Edwards group law** As we have seen, twisted Edwards curves are equivalent to Montgomery curves and as such also have a group law. However, in contrast to Montgomery and Weierstraß curves, the group law of snark friendly twisted Edwards curves can be described by single computation, that works in all cases, no matter if we add the neutral element, inverse, or if have to double a point. To see how the group law looks like, first observe that the point $(0,1)$ is a solution to $a\cdot x^2+y^2=1+d\cdot x^2\cdot y^2$ for any curve. The sum of any two points $(x_1,y_1)$, $(x_2,y_2)$ on an Edwards curve $E(\mathbb{F})$ is then given by

$$(x_1,y_1)\oplus(x_2,y_2)=\left(\frac{x_1y_2+y_1x_2}{1+dx_1x_2y_1y_2},\frac{y_1y_2-ax_1x_2}{1-dx_1x_2y_1y_2}\right)$$

and it can be shown that the point $(0,1)$ serves as the neutral element and the inverse of a point $(x_1,y_1)$ is given by $(-x_1,y1)$.

**Example 80.** *Lets look at the tiny-jubjub curve in Edwards form from example XXX again. As we have seen, this curve is given by*

$$\begin{aligned}PJJ\_13=\{&(0,1),(0,12),(1,2),(1,11),(2,6),(2,7),(3,0),(5,5),(5,8),(6,4),\\&(6,9),(7,4),(7,9),(8,5),(8,8),(10,0),(11,6),(11,7),(12,2),(12,11)\}\end{aligned}$$

*To get an undertanding of the twisted Edwards addition law, lets first add the neutral element* $(0,1)$ *to itself. We apply the group law XXX and get*

$$\begin{aligned}(0,1)\oplus(0,1)&=\left(\frac{0\cdot1+1\cdot0}{1+8\cdot0\cdot0\cdot1\cdot1},\frac{1\cdot1-3\cdot0\cdot0}{1-8\cdot0\cdot0\cdot1\cdot1}\right)\\&=(0,1)\end{aligned}$$

*So as expected, adding the neutral element to itself gives the neutral element again. Now lets add the neutral element to some other curve point. We get*

$$\begin{aligned}(0,1)\oplus(8,5)&=\left(\frac{0\cdot5+1\cdot8}{1+8\cdot0\cdot8\cdot1\cdot5},\frac{1\cdot5-3\cdot0\cdot8}{1-8\cdot0\cdot8\cdot1\cdot5}\right)\\&=(8,5)\end{aligned}$$

*Again as expected adding the neutral element to any element will give the element again. Given any curve point* $(x,y)$*, we know that the inverse is given by* $(-x,y)$*. To see how the addition of a point to its inverse works out we therefore compute*

$$\begin{aligned}(5,5)\oplus(8,5)&=\left(\frac{5\cdot5+5\cdot8}{1+8\cdot5\cdot8\cdot5\cdot5},\frac{5\cdot5-3\cdot5\cdot8}{1-8\cdot5\cdot8\cdot5\cdot5}\right)\\&=\left(\frac{12+1}{1+5},\frac{12-3}{1-5}\right)\\&=\left(\frac{0}{6},\frac{12+10}{1+8}\right)\\&=\left(0,\frac{9}{9}\right)\\&=(0,1)\end{aligned}$$

*So adding a curve point to its inverse gives the neutral element, as expected. As we have seen from these examples the twisted Edwards addition law handles edge cases particulary nice and in a unified way.*

## 6.2 Elliptic Curves Pairings

As we have seen in XXX some groups comes with the notation of a so called pairing map, which is a non-degenerate bilinar map, from two groups into another group.

In this section, we discuss *pairings on elliptic curves*, which form the basis of several zk-SNARKs and other zero knowledge proof schemes. The SNARKs derived from pairings have the advantage of constant-sized proof sizes, which is crucial to blockchains.

We start out by defining elliptic curve pairings and discussing a simple application which bears some resemblance to the more advanced SNARKs. We then introduce the pairings arising from elliptic curves and describe Miller's algorithm which makes these pairings practical rather than just theoretically interesting.

Elliptic curves have a few structures, like the Weil or the Tate map, that qualifies as pairing.

**Embedding Degrees** As we will see in what follows, every elliptic curve gives rise to a pairing map. However as we will see in example XXX, not every such pairing is efficiently computeable. So in order to distingush curves with efficiently computable pairings from the rest, we need to start with an introduction to the so called **embedding degree** of a curve.

To understand this term, let $\mathbb{F}$ be a finite field, $E(\mathbb{F})$ an elliptic curve over $\mathbb{F}$, and $n$ a prime number that divides the order of $E(\mathbb{F})$. The embedding degree of $E(\mathbb{F})$ with respect to $n$ is then the smallest integer $k$ such that $n$ divides $q^k - 1$.

Fermat's little theorem XXX implies, that every curve has at least *some* embedding degree $k$, since at least $k = n-1$ is always a solution to the congruency $q^k \equiv 1 \pmod{n}$ which implies that the remainder of the integer division of $q^k - 1$ by $n$ is 0.

**Example 81.** *To get a better intuition of the embedding degree, lets consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We know from XXX that the order of $E_1(\mathbb{F}_5)$ is 9 and since the only prime factor of 9 is 3, we compute the ebedding degree of $E_1(\mathbb{F}_5)$ with respect to 3.*

*To find that embedding degree we have to find the smallest integer k, such that 3 divides $q^k - 1 = 5^k - 1$. We try and increment until we find a proper k.*

$$
\begin{aligned}
k = 1\text{: } & 5^1 - 1 = 4 && \textit{not divisible by } 3 \\
k = 2\text{: } & 5^2 - 1 = 24 && \textit{divisible by } 3
\end{aligned}
$$

*So we know that the embedding degree of $E_1(\mathbb{F}_5)$ is 2 relative to the the prime factor 3.*

**Example 82.** *Lets consider the tiny jubjub curve TJJ_13 from example XXX. We know from XXX that the order of TJJ_13 is 20 and that the order therefore has two prime factors. A "large" prime factor 5 and a small prime factor 2.*

*We start by computing the ebedding degree of TJJ_13 with respect to the large prime factor 5. To find that embedding degree we have to find the smallest integer k, such that 5 divides $q^k - 1 = 13^k - 1$. We try and increment until we find a proper k.*

$$
\begin{aligned}
k = 1\text{: } & 13^1 - 1 = 12 && \textit{not divisible by } 5 \\
k = 2\text{: } & 13^2 - 1 = 168 && \textit{not divisible by } 5 \\
k = 3\text{: } & 13^3 - 1 = 2196 && \textit{not divisible by } 5 \\
k = 4\text{: } & 13^4 - 1 = 28560 && \textit{divisible by } 5
\end{aligned}
$$

*So we know that the embedding degree of TJJ_13 is 4 relative to the the prime factor 5.*

*In real world applications, like on pairing friendly elliptic curves as for example BLS_12-381, usually only the embedding degree of the large prime factor are relevant, which in case of out tiny-jubjub curve, is represented by* $5$. *It should however be noted that every prime factor of a curves order has its own notation of embedding degree despite the fact that this is mostly irrelevant in applications.*

*To find the embedding degree of the small prime factor* $2$ *we have to find the smallest integer* $k$*, such that* $2$ *divides* $q^k - 1 = 13^k - 1$. *We try and increment until we find a proper* $k$.

$$k = 1\text{: } 13^1 - 1 = 12 \qquad\qquad \text{divisible by } 2$$

*So we know that the embedding degree of TJJ_13 is* $1$ *relative to the the prime factor* $2$. *So as we have seen, different prime factors can have different embedding degrees in general.*

```
sage: p = 13
sage: # large prime factor
sage: n = 5
sage: for k in range(1,5): # Fermat's little theorem
....:     if (p^k-1)%n == 0:
....:         break
sage: k
4
sage: # small prime factor
sage: n = 2
sage: for k in range(1,2): # Fermat's little theorem
....:     if (p^k-1)%n == 0:
....:         break
sage: k
1
```

**Example 83.** *To give an example of a cryptographically secure real worl elliptic curve that does not have a small embedding degree lets look at curve secp256k1 again. We know from XXX that the order of this curve is a prime number, so we only have a single embedding degree.*

*To test potential embedding degrees* $k$*, say in the range* $1 \ldots 1000$*, we can invoke sage and compute:*

```
sage: p = 115792089237316195423570985008687907853269984665640
   564039457584007908834671663
sage: n = 115792089237316195423570985008687907852837564279074
   904382605163141518161494337
sage: for k in range(1,1000):
....:     if (p^k-1)%n == 0:
....:         break
sage: k
999
```

*So we see that secp256k1 has at least no embedding degree* $k < 1000$*, which renders secp256k1 as a curve that has no small embedding degree. A property that is of importance later on.*

**Elliptic Curves over extension fields** Suppose that $p$ is a prime number and $\mathbb{F}_p$ its associated prime field. We know from XXX, that the fields $\mathbb{F}_{p^m}$ are extensions of $\mathbb{F}_p$ in the sense that $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p^m}$. This implies that we can extend the affine plane an elliptic curve is defined on, by changing the base field to any extension field. To be more precise let $E(\mathbb{F}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + a\cdot x + b\}$ be an affine short Weierstrass curve, with parameters $a$ and $b$ taken from $\mathbb{F}$. If $\mathbb{F}'$ is any extension field of $\mathbb{F}$, then we extend the domain of the curve by defining

$$E(\mathbb{F}') = \{(x,y) \in \mathbb{F}' \times \mathbb{F}' \mid y^2 = x^3 + a\cdot x + b\} \tag{6.13}$$

So while we did not change the defining parameters, we consider curve points from the affine plane over the extension field now. Since $\mathbb{F} \subset \mathbb{F}'$ it can be shown that the original elliptic curve $E(\mathbb{F})$ is a sub curve of the extension curve $E(\mathbb{F}')$.

**Example 84.** *Conider the prime field $\mathbb{F}_5$ from example XXX and the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. Since we know from XXX that $\mathbb{F}_{5^2}$ is an extension field of $\mathbb{F}_5$, we can extend the definition of $E_1(\mathbb{F}_5)$ to define a curve over $\mathbb{F}_{5^2}$:*

$$E_1(\mathbb{F}_{5^2}) = \{(x,y) \in \mathbb{F} \times \mathbb{F} \mid y^2 = x^3 + x + 1\}$$

*Since $\mathbb{F}_{5^2}$ contains 25 points, in order to compute the set $E_1(\mathbb{F}_{5^2})$, we have to try $25 \cdot 25 = 625$ pairs, which is probably a bit to much for the avarage motivated reader. Instaed we involve sage to compute the curve for us. To do so choose the representation of $\mathbb{F}_{5^2}$ from XXX. We get:*

```
sage: F5= GF(5)
sage: F5t.<t> = F5[]
sage: P = F5t(t^2+2)
sage: P.is_irreducible()
True
sage: F5_2.<t> = GF(5^2, name='t', modulus=P)
sage: E1F5_2 = EllipticCurve(F5_2,[1,1])
sage: E1F5_2.order()
27
```

*So curve $E_1(\mathbb{F}_{5^2})$ consist of 27 points, in contrast to curve $E_1(\mathbb{F}_5)$, which consists of 9 points. Printing the points gives*

$$\begin{aligned} E_1(\mathbb{F}_{5^2}) = \{&\mathcal{O},(0,4),(0,1),(3,4),(3,1),(4,3),(4,2),(2,4),(2,1),\\ &(4t+3,3t+4),(4t+3,2t+1),(3t+2,t),(3t+2,4t),\\ &(2t+2,t),(2t+2,4t),(2t+1,4t+4),(2t+1,t+1),\\ &(2t+3,3),(2t+3,2),(t+3,2t+4),(t+3,3t+1),\\ &(3t+1,t+4),(3t+1,4t+1),(3t+3,3),(3t+3,2),(1,4t)\} \end{aligned}$$

*As we can see, curve $E_1(\mathbb{F}_5)$ sits inside curve $E(\mathbb{F}_{5^2})$, which is implied from $\mathbb{F}_5$ being a subfield of $\mathbb{F}_{5^2}$.*

**Full Torsion groups** The fundamental theorem of finite cyclic groups XXX implies, that every prime factor $n$ of a cyclic groups order defines a subgroup of the size of the prime factor. We called such a subgroup an $n$-torsion group. We have seen many of those subgroups in the examples XXX and XXX.

Now when we consider elliptic curve extensions as defined in XXX, we could ask, what happens to the $n$-torsion groups in the extension. One might intuitively think that their expension just parallels the extension of the curve. For example when $E(\mathbb{F}_p)$ is a curve over prime field

$\mathbb{F}_p$, with some $n$-torsion group $\mathbb{G}$ and when we extend the curve to $E(\mathbb{F}_{p^m})$, then there is a bigger $n$-torstion group, such that $\mathbb{G}$ is a subgroup. Naively this would make sense, as $E(\mathbb{F}_p)$ is a subcurve of $E(\mathbb{F}_{p^m})$.

However the real situation is a bit more surprising then that. To see that, let $\mathbb{F}_p$ be a prime field and $E(\mathbb{F}_p)$ an elliptic curve of order $r$, with embedding degree $k$ and $n$-torsion group $E(\mathbb{F}_p)[n]$ for same prime factor $n$ of $r$. Then it can be shown that the $n$-torsion group $E(\mathbb{F}_{p^m})[n]$ of a curve extension is equal to $E(\mathbb{F}_p)[n]$, as long as the power $m$ is less then the embedding degree $k$ of $E(\mathbb{F}_p)$.

However for the prime power $p^m$, for any $m \geq k$, $E(\mathbb{F}_{p^m})[n]$ is strictly larger then $E(\mathbb{F}_p)[n]$ and contains $E(\mathbb{F}_p)[n]$ as a subgroup. We call the $n$-torsion group $E(\mathbb{F}_{p^k})[n]$ of the extension of $E$ over $\mathbb{F}_{p^k}$ the **full $n$-torsion group** of that elliptic curve. It can be shown that it contains $n^2$ many elements and consists of $n+1$ subgroups, one of which is $E(\mathbb{F}_p)[n]$.

So roughly speaking, when we consider towers of curve extensions $E(\mathbb{F}_{p^m})$, ordered by the prime power $m$, then the $n$-torsion group stays constant for every level $m$ small then the embedding degee, while it suddenly blossoms into a larger group on level $k$, with $n+1$ subgroups and it then stays like that for any level $m$ larger then $k$. In other words, once the extension field is big enough to find on emore point of order $n$ (that is not defined over the base field), then we actually find all of the points in the full torsion group.

**Example 85.** *Consider curve $E_1(\mathbb{F}_5)$ again. We know that it contains a $3$-torsion group and that the embedding degree of $3$ is $2$. From this we can deduce that we can find the full $3$-torsion group $E_1[3]$ in the curve extension $E_1(\mathbb{F}_{5^2})$, the latter of which we computed in XXX.*

*Since that curve is small, in order to find the full $3$-torsion, we can loop through all elements of $E_1(\mathbb{F}_{5^2})$ and check check the defining equation $[3]P = \mathcal{O}$. Invoking sage we compute*

```
sage: INF = E1F5_2(0) # Point at infinity
sage: L_E1_3 = []
sage: for p in E1F5_2:
....:     if 3*p == INF:
....:         L_E1_3.append(p)
sage: E1_3 = Set(L_E1_3) # Full 3-torsion set
```

*we get*

$$E_1[3] = \{\mathcal{O}, (1,t), (1,4t), (2,1), (2,4), (2t+1, t+1), (2t+1, 4t+4), (3t+1, t+4), (3t+1, 4t+1)\}$$

**Example 86.** *Consider the tiny jubjub curve from example XXX. we know from XXX that it contains a $5$-torsion group and that the embedding degree of $5$ is $4$. This implies that we can find the full $5$-torsion group TJJ_13$[5]$ in the curve extension TJJ_13$(\mathbb{F}_{13^4})$.*

*To compute the full torsion, first observe that since $\mathbb{F}_{13^4}$ contains $28561$ element, computing TJJ_13$(\mathbb{F}_{13^4})$ means checking $28561^2 = 815730721$ elements. From each of these curve points $P$, we then have to check the equation $[5]P = \mathcal{O}$. Doing this for $815730721$ is a bit to slow even on a computer.*

*Fortunate sage has a way to loop through points of given order efficiently. The following sage code then gives a way to compute the full torsion group:*

```
sage: # define the extension field
sage: F13= GF(13) # prime field
sage: F13t.<t> = F13[] # polynomials over t
sage: P = F13t(t^4+2) # irreducible polynomial of degree 4
```

```
sage: P.is_irreducible()
True
sage: F13_4.<t> = GF(13^4, name='t', modulus=P) # F_{13^4}
sage: TJJF13_4 = EllipticCurve(F13_4,[8,8]) # tiny jubjub
    extension
sage: # compute the full 5-torsion
sage: L_TJJF13_4_5 = []
sage: INF = TJJF13_4(0)
sage: for P in INF.division_points(5): # [5]P == INF
....:     L_TJJF13_4_5.append(P)
sage: len(L_TJJF13_4_5)
25
sage: TJJF13_4_5 = Set(L_TJJF13_4_5)
```

*So as expected we get a group that contains $5^2 = 25$ elements. As its rather tedious to write this group down and as we don't need in what follows we skipp writing it. To see that the embedding degree $4$ is actually the smallerst prime power to find the full $5$-torsion group, lets compute the $5$-torsion group over of the tiny-jubjub curve the extension field $\mathbb{F}_{13^3}$. We get*

```
sage: # define the extension field
sage: P = F13t(t^3+2) # irreducible polynomial of degree 3
sage: P.is_irreducible()
True
sage: F13_3.<t> = GF(13^3, name='t', modulus=P) # F_{13^3}
sage: TJJF13_3 = EllipticCurve(F13_3,[8,8]) # tiny jubjub
    extension
sage: # compute the 5-torsion
sage: L_TJJF13_3_5 = []
sage: INF = TJJF13_3(0)
sage: for P in INF.division_points(5): # [5]P == INF
....:     L_TJJF13_3_5.append(P)
sage: len(L_TJJF13_3_5)
5
sage: TJJF13_3_5 = Set(L_TJJF13_3_5) # full $5$-torsion
```

*So as we can see the $5$-torsion group of tiny-jubjub over $\mathbb{F}_{13^3}$ is equal to the $5$-torsion group of tiny-jubjub over $\mathbb{F}_{13}$ itself.*

**Example 87.** *Lets look at curve Secp256k1. We know from XXX that the curve is of some prime order r and hence the only n-torsion group to consider is the curve itself. So the curve group is the r-torsion.*

*However in order to find the full r-torsion of Secp256k1, we need to compute the embedding degree k and as we have seen in XXX it is at least not small. We know from Fermat's little theorem that a finite embedding degree must exist, though. It can be shown that it is given by*

$$k = 19298681539552699237261830834781317547292737984581739710086052358636024 9056$$

*which is a 256bit number. So the embedding degree is huge, which implies that the fiel extension $\mathbb{F}_{p^k}$ is huge too. To understand how big $\mathbb{F}_{p^k}$ is, recall that an element of $\mathbb{F}_{p^m}$ can be represented as a string $[x_0, \ldots, x_m]$ of m elements, each containing a number from the prime field $\mathbb{F}_p$. Now in the case of Secp256k1, such a representation has k-many entries, each of $256$ bits in size.*

*So without any optimizations, representing such an element would need $k \cdot 256$ bits, which is to much to be reprsented in the observable universe.*

**Torsion-Subgroups** As we have stated above, any full $n$-torsion group contains $n+1$ cyclic subgroups, two of which are of particular interest in pairing based elliptic curve cryptography. To characzerize these groups we need to consider the so called *Frobenious* endomorphism

$$\pi : E(\mathbb{F}) \to E(\mathbb{F}) : \quad \begin{array}{ccc} (x,y) & \mapsto & (x^p, y^p) \\ \mathcal{O} & \mapsto & \mathcal{O} \end{array} \tag{6.14}$$

of an alliptic curve $E(\mathbb{F})$ over some finite field $\mathbb{F}$ of characteristic $p$. It can be shown that $\pi$ maps curve points to curve points. The first thing to note is that in case that $\mathbb{F}$ is a prime field, the Frobenious endomorphism acts trivially, since $(x^p, y^p) = (x,y)$ on prime fields, due to Fermat's little theorem XX. So the Frobenious map is more interesting over prime field extensions.

With the Frobenious map at hand, we can now characterise two important subgroups of the full $n$-torsion. The first subgroup is the $n$-torsion group that already exists in the curve over the base field. In pairing based cryptography this group is usually written as $\mathbb{G}_1$, assuming that the prime factor '$n$' in the definition is implicitly given. Since we know that the Frobenious map, acts trivially on curve over the prime field we can define $\mathbb{G}_1$ as:

$$\mathbb{G}_1[n] := \{(x,y) \in E[n] \mid \pi(x,y) = (x,y)\ \} \tag{6.15}$$

In more mathematical terms this definition means, that $\mathbb{G}_1$ is the *Eigenspace* of the Frobenious map with respect to the *Eigenvalue* 1.

Now it can be shown, that there is another subgroup of the full $n$-torsion group that can be characterized by the Frobenious map. In the context of so called type 3 pairing based cyrptography this subgroup is usually called $\mathbb{G}_2$ and it defined as

$$\mathbb{G}_2[n] := \{(x,y) \in E[n] \mid \pi(x,y) = [p](x,y)\ \} \tag{6.16}$$

So in mathematical terms $\mathbb{G}_2$ is the *Eigenspace* of the Frobenious map with respect to the *Eigenvalue $p$*.

**Notation and Symbols 9.** *If the prime factor n of the curves order is clear from the contect, we sometimes simply write $\mathbb{G}_1$ and $\mathbb{G}_2$ to mean $\mathbb{G}_1[n]$ and $\mathbb{G}_2[n]$, respectively.*

It should be noted however that sometimes other definitions of $\mathbb{G}_2$ appear in the literature, however in the context of pairing based cryptography, this is the most common one. It is particulary useful, as we can define hash functions that map into $\mathbb{G}_2$, which is not possible for all subgroups of the full $n$-torsion.

**Example 88.** *Consider the curve $E_1(\mathbb{F}_5)$ from example XXX again. As we have seen this curve has embedding degree $k = 2$ and a full 3-torsion group is given by*

$$E_1[3] = \{\mathcal{O}, (2,1), (2,4), (1,t), (1,4t), (2t+1, t+1), (2t+1, 4t+4), (3t+1, t+4), (3t+1, 4t+1)\}$$

*According to the general theory, $E_1[3]$ contains 4 soubgroups and we chan characterise the subgroups $\mathbb{G}_1$ and $\mathbb{G}_2$ using the Frobenious endomorphism. Unfortunately at the time of this writing sage did have a predifined Frobenious endomorphism for elliptic curves, so we have to use the Frobenious endomorphism of the underlying field as a temorary workaround. We compute*

```
sage: L_G1 = []
sage: for P in E1_3:
....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
....:     if P == PiP:
....:         L_G1.append(P)
sage: G1 = Set(L_G1)
```

*So as expected the group $\mathbb{G}_1 = \{\mathcal{O}, (2,4), (2,1)\}$ is identical to the 3-torsion group of the (unextended) curve over the prime field $E_1(\mathbb{F}_5)$. We can use almost the same algorithm to compute the group $\mathbb{G}_2$ and get*

```
sage: L_G2 = []
sage: for P in E1_3:
....:     PiP = E1F5_2([a.frobenius() for a in P]) # pi(P)
....:     pP = 5*P # [5]P
....:     if pP == PiP:
....:         L_G2.append(P)
sage: G2 = Set(L_G2)
```

*so we compute the the second subgroup of the full 3-torsion group of curve $E_1$ as the set $\mathbb{G}_2 = \{\mathcal{O}, (1,t), (1,4t)\}$.*

**Example 89.** *Considering the tiny-jubjub curve TJJ_13 from example XXX. In example XXX we computed its full 5 torsion, which is a group that has 6 subgroups. We compute G1 using sage as:*

```
sage: L_TJJ_G1 = []
sage: for P in TJJF13_4_5:
....:     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P)
....:     if P == PiP:
....:         L_TJJ_G1.append(P)
sage: TJJ_G1 = Set(L_TJJ_G1)
```

*We get $\mathbb{G}1 = \{\mathcal{O}, (7,2), (8,8), (8,5), (7,11)\}$*

```
sage: L_TJJ_G1 = []
sage: for P in TJJF13_4_5:
....:     PiP = TJJF13_4([a.frobenius() for a in P]) # pi(P)
....:     pP = 13*P # [5]P
....:     if pP == PiP:
....:         L_TJJ_G1.append(P)
sage: TJJ_G1 = Set(L_TJJ_G1)
```

$\mathbb{G}_2 = \{\mathcal{O}, (9t^2+7, t^3+11t,), (9t^2+7, 12t^3+2t), (4t^2+7, 5t^3+10t), (4t^2+7, 8t^3+3t)\}$

**Example 90.** *Consider Bitcoin's curve Secp256k1 again. Since the group $\mathbb{G}_1$ is identical to the torsion group of the unextended curve and since Secp256k1 has prime order, we know, that in thi case $\mathbb{G}_1$ is identical to Secp256k1. It is however infeasible not just to compute $\mathbb{G}_2$ itself, but to even compute an avarage element of $\mathbb{G}_2$ as elements need to much storage to be representable in this universe.*

**The Weil Pairing** In this part we consider a pairing function defined on the subgroups $\mathbb{G}_1[r]$ and $\mathbb{G}_2[r]$ of the full $r$-torsion $E[r]$ of a short Weierstraß elliptic curve. To be more precise let

$E(\mathbb{F}_p)$ be an elliptic curve of embedding degree $k$, such that $r$ is a prime factor of its order. Then the **Weil pairing** is a bilinear, non-degenerate map

$$e(\cdot,\cdot): \mathbb{G}_1[r] \times \mathbb{G}_2[r] \to \mathbb{F}_{p^k} \;;\; (P,Q) \mapsto (-1)^r \cdot \frac{f_{r,P}(Q)}{f_{r,Q}(P)} \tag{6.17}$$

where the extension field elements $f_{r,P}(Q), f_{r,Q}(P) \in \mathbb{F}_{p^k}$ are computed by **Miller's algorithm**: Understanding in detail how the algorithm works requires the concept of *divisors*, which we

**Algorithm 7** Miller's algorithm for short Weierstraß curves $y^2 = x^3 + ax + b$

**Require:** $r > 3$, $P \in E[r]$, $Q \in E[r]$ and
  $b_0, \ldots, b_t \in \{0,1\}$ with $r = b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + b_t \cdot 2^t$ and $b_t = 1$
  **procedure** MILLER'S ALGORITHM($P,Q$)
    **if** $P = \mathcal{O}$ or $Q = \mathcal{O}$ or $P = Q$ **then**
      **return** $f_{r,P}(Q) \leftarrow (-1)^r$
    **end if**
    $(x_T, y_T) \leftarrow (x_P, y_P)$
    $f_1 \leftarrow 1$
    $f_2 \leftarrow 1$
    **for** $j \leftarrow t-1, \ldots, 0$ **do**
      $m \leftarrow \frac{3 \cdot x_T^2 + a}{2 \cdot y_T}$
      $f_1 \leftarrow f_1^2 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$
      $f_2 \leftarrow f_2^2 \cdot (x_Q + 2x_T - m^2)$
      $x_{2T} \leftarrow m^2 - 2x_T$
      $y_{2T} \leftarrow -y_T - m \cdot (x_{2T} - x_T)$
      $(x_T, y_T) \leftarrow (x_{2T}, y_{2T})$
      **if** $b_j = 1$ **then**
        $m \leftarrow \frac{y_T - y_P}{x_T - x_P}$
        $f_1 \leftarrow f_1 \cdot (y_Q - y_T - m \cdot (x_Q - x_T))$
        $f_2 \leftarrow f_2 \cdot (x_Q + (x_P + x_T) - m^2)$
        $x_{T+P} \leftarrow m^2 - x_T - x_P$
        $y_{T+P} \leftarrow -y_T - m \cdot (x_{T+P} - x_T)$
        $(x_T, y_T) \leftarrow (x_{T+P}, y_{T+P})$
      **end if**
    **end for**
    $f_1 \leftarrow f_1 \cdot (x_Q - x_T)$
    **return** $f_{r,P}(Q) \leftarrow \frac{f_1}{f_2}$
  **end procedure**

don't really need in this book. The interested reader might look at [REFERENCES]

In real world application of pairing friendly elliptic curves, the embedding degree is usuall a small number like 2, 4, 6 or 12 and the number $r$ is the largest prime factor of the curves order.

**Example 91.** *Consider curve $E_1(\mathbb{F}_5)$ from example XXX. Since the only prime factor of the groups order is 3 we can not compute the Weil pairing on this group using our definition of Miller's algorithm. In fact since $\mathbb{G}_1$ is of order 3, executing the if statement on line XXX will lead to a division by zero error in the computation of the slope $m$.*

**Example 92.** *Consider the tiny-jubjub curve TJJ_13$(\mathbb{F}_{13})$ from example XXX again. We want to instantiate the general definition of the Weil pairing for this example. To do so, recall that* *we have see in example XXX, its embedding degree is* 4 *and that we have the following type-3 pairing groups:*

$$\mathbb{G}_1 = \{\mathcal{O}, (7,2), (8,8), (8,5), (7,11)\}$$
$$\mathbb{G}_2 = \{\mathcal{O}, (9t^2+7, t^3+11t), (9t^2+7, 12t^3+2t), (4t^2+7, 5t^3+10t), (4t^2+7, 8t^33+3t)\}$$

*where* $\mathbb{G}_1$ *and* $\mathbb{G}_2$ *are subgroups of the full* 5*-torsion found in the curve TJJ_13*$(\mathbb{F}_{13^4})$*. The type-3 Weil pairing is a map* $e(\cdot,\cdot) : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_{13^4}$*. From the first if-statement in Miller's algorithm, we can deduce that* $e(\mathcal{O}, Q) = 1$ *as well as* $e(P, \mathcal{O}) = 1$ *for all arguments* $P \in \mathbb{G}_1$ *and* $Q \in \mathbb{G}_2$*. So in order to compute a non-trivial Weil pairing we choose the arguments* $P = (7,2)$ *and* $Q = (9t^2+7, 12t^3+2t)$*.*

*In order to compute the pairing* $e((7,2), (9t^2+7, 12t^3+2t))$ *we have to compute the extension field elements* $f_{5,P}(Q)$ *and* $f_{5,Q}(P)$ *applying Miller's algorithm. Do do so first observe that we have* $5 = 1\cdot 2^0 + 0\cdot 2^1 + 1\cdot 2^2$*, so we get* $t = 2$ *as well as* $b_0 = 1$*,* $b_1 = 0$ *and* $b_2 = 1$*. The loop therefore needs to be executed two times.*

*Computing* $f_{5,P}(Q)$*, we initiate* $(x_T, y_T) = (7,2)$ *as well as* $f_1 = 1$ *and* $f_2 = 1$*. Then*

| $j$ | $b_j$ | $m$ | $f_1$ | $f_2$ | $x_{2T}$ | $y_{2T}$ | $x_{T+P}$ | $y_{T+P}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | $\cdot$ | | | | | | | |

$$\begin{aligned} m &= \frac{3\cdot x_T^2 + a}{2\cdot y_T} \\ &= \frac{3\cdot 2^2+1}{2\cdot 4} = \frac{3}{3} \\ &= 1 \end{aligned}$$

$$\begin{aligned} f_1 &= f_1^2 \cdot (y_Q - y_T - m\cdot(x_Q - x_T)) \\ &= 1^2\cdot(t - 4 - 1\cdot(1-2)) = t - 4 + 1 \\ &= t + 2 \end{aligned}$$

$$\begin{aligned} f_2 &= f_2^2 \cdot (x_Q + 2x_T - m^2) \\ &= 1^2\cdot(1 + 2\cdot 2 - 1^2) = (1 + 4 - 1) \\ &= 4 \end{aligned}$$

$$\begin{aligned} x_{2T} &= m^2 - 2x_T \\ &= 1^2 - 2\cdot 2 = -3 \\ &= 2 \end{aligned}$$

$$\begin{aligned} y_{2T} &= -y_T - m\cdot(x_{2T} - x_T) \\ &= -4 - 1\cdot(2-2) = -4 \\ &= 1 \end{aligned}$$

*So we update* $(x_T, y_T) = (2,1)$ *and since* $b_0 = 1$ *we have to execute the if statement on line XXX in the for loop. However since we only loop a single time, we don't need to compute* $y_{T+P}$*, since* *we only need the updated $x_T$ in the final step. We get:*

$$m = \frac{y_T - y_P}{x_T - x_P} = \frac{1-4}{2-x_P}$$

$$f_1 = f_1 \cdot (y_Q - y_T - m\cdot(x_Q - x_T))$$

$$f_2 = f_2 \cdot (x_Q + (x_P + x_T) - m^2)$$

$$x_{T+P} = m^2 - x_T - x_P$$

## 6.3 Hashing to Curves

Elliptic curve cryptography frequently requires the ability to hash data onto elliptic curves. If the order of the curve is not a prime number hashing to prime number subgroups is also of importance. In the context of pairing friendly curves it is also sometimes necessary to hash specifically onto the group $\mathbb{G}_1$ or $\mathbb{G}_2$.

As we have seen in XXX, many general methods are know to hash into groups in general and finite cyclic groups in particular. As elliptic groups are cyclic those method can be utilized in this case, too. However in what follows we want to describe some methods special to elliptic curves, that are frequently used in applications.

**Try and increment hash functions** One of the most straight forward ways to hash a bitstring onto an eliptic curve point, in a secure way, is to use a cryptographic hash function together with one of the methods we described in XXX to hash to the modular arithmetics base field of the curve. Ideally the hash function generates an image that is at least one bit longer then the bit representation of the base field modulus.

The image in the base field can then be interpreted as the $x$-coordinate of the curve point and the two possible $y$-coordinates are then derived from the curve equation, while one of the bits that exceeded the modulus determines which of the two $y$-coordinates to choose.

Such an approach would be easy to implement and deterministic and it will conserve the cryptographic properties of the original hash function. However not all $x$-ccordinates genrated in such a way, will result in quadratic residues, when inserted into the defining equation. It follows that not all field elements give rise to actual curve points. In fact on a prime field, only half of the field elements are quadratic residues and hence assuming an even distribution of the hash values in the field, this method would fail to generate a curve point in about half of the attempts.

One way to account for this problem is the so called *try and increment* method. Its basic assumption is, that hashing different values, the result will eventually lead to a valid curve point.

Therefore instead of simply hashing a string $s$ to the field the concatenation of $s$ with additional bytes is hashed to the field instead, that is a try and increment hash as described in XXX is used. If the first *try* of hashing to the field does not result in a valid curve point, the counter is *incremented* and the hashing is repeated again. This is done until a valid curve point is found eventually.

---

**Algorithm 8** Hash-to-$E(\mathbb{F}_r)$

---

**Require:** $r \in \mathbb{Z}$ with $r.nbits() = k$ and $s \in \{0,1\}^*$
**Require:** Curve equation $y^2 = x^3 + ax + b$ over $\mathbb{F}_r$
  **procedure** TRY-AND-INCREMENT$(r,k,s)$
    $c \leftarrow 0$
    **repeat**
      $s' \leftarrow s||c\_bits()$
      $z \leftarrow H(s')_0 \cdot 2^0 + H(s')_1 \cdot 2^1 + \ldots + H(s')_k \cdot 2^k$
      $x \leftarrow z^3 + a \cdot z + b$
      $c \leftarrow c + 1$
    **until** $z < r$ and $x^{\frac{r-1}{2}} \bmod r = 1$
    **if** $H(s')_{k+1} == 0$ **then**
      $y \leftarrow \sqrt{x}$ #(root in $\mathbb{F}_r$)
    **else**
      $y \leftarrow r - \sqrt{x}$ #(root in $\mathbb{F}_r$)
    **end if**
    **return** $(x,y)$
  **end procedure**
**Ensure:** $(x,y) \in E(\mathbb{F}_r)$

---

This method has the advantegous that is relatively easy to implement in code and that it preserves the cryptographic properties of the original hash function. However it is not guranteed to find a valid curve point, as there is a chance that all possible values in the choosen size of the counter will fail to generate a quadratic residue. Fortunately it is possible to make the probability for this arbitrarily small by choosing large enough counters and relying on the (approximate) uniformity of the hash-to-field function.

If the curve is not of prime order, the result will be a general curve point that might not be in the "large" prime order subgroup. A so called *cofactor clearing* step is then necessary to project the curve point onto the subgroup. This is done by scalar multiplication with the cofactor of prime order with respect to the curves order.

**Example 93.** *Consider the tiny jubjub curve from example XXX. We want to construct a try and increment hash function, that hashes a binary string s of arbitrary length onto the large prime order subgroup of size* 5.

*Since the curve as well as our targetet subgroup are defined over the field* $\mathbb{F}_{13}$ *and the binary representation of* 13 *is* $13.bits() = 1101$*, we apply SHA256 from sage's crypto library on the concatension* $s||c$ *for some binary counter string and use the first* 4 *bits of the image to try to hash into* $\mathbb{F}_{13}$*. In case we are able to hash to a value z, such that* $z^3 + 8 \cdot z + 8$ *is a quadratic residue in* $\mathbb{F}_{13}$*, we use the* 5*-th bit to decide which of the two possible roots of* $z^3 + 8 \cdot z + 8$ *we will choose as the y-coordinate. The result is then a curve point different from the point at infinity. To project it to a point of* $\mathbb{G}_1$*, we multiply it with the cofactor* 4*. If the result is still not the point at infinity, it is the result of the hash.*

*To make this concrete let* $s =' 1001100111101011010000011 1'$ *be our binary string that we want to hash onto* $\mathbb{G}_1$*. We use a* 4*-bit binary counter, starting at zero, i.e we choose* $c = 0000$*. Invoking sage we define the try-hash function as*

```
sage: import Crypto
sage: from Crypto.Hash import SHA256
sage: def try_hash(s,c):
....:     s_1 = s+c
....:     h = SHA256.new(s_1)
....:     d = h.hexdigest()
....:     d = Integer(d,base=16)
....:     sign = d.str(2)[-5:-4]
....:     d = d.str(2)[-4:]
....:     z = Integer(d,base=2)
....:     return (z,sign)
sage: try_hash('10011001111010110100000111','0000')
(15, '1')
```

*As we can see, our first apptempt to hash into $\mathbb{F}_{13}$ was not sucessfull as 15 is not a number in $\mathbb{F}_{13}$, so we increment the binary counter by 1 and try again:*

```
sage: try_hash('10011001111010110100000111','0001')
(3, '0')
```

*And we find a hash into $\mathbb{F}_{13}$. However this point is not guranteed to define a curve point. To see that we insert $z = 3$ into the right side of the Weierstraß equation of the tiny.jubjub curve and compute $3^3 + 8 * 3 + 8 = 7$, but 7 is not a quadratic residue in $\mathbb{F}_{13}$ since $7^{\frac{13-1}{2}} = 7^6 = 12 = -1$. So 3 is a not a suteable point and we have to increment the counter two more times:*

```
sage: try_hash('10011001111010110100000111','0010')
(3, '0')
sage: try_hash('10011001111010110100000111','0011')
(6, '1')
```

*Since $6^3 + 8 \cdot 6 + 8 = 12$ and we have $\sqrt{12} \in \{5, 8\}$, we finally found the valid x coordinate $x = 6$ for the curve point hash. Now since the sign bit of this hash is 1, we choose the larger root $y = 8$ as the y-coordinate and get the hash*

$$H('10011001111010110100000111') = (6, 8)$$

*which is a valid curve point point on the tiny jubjub curve. Now in order to project it onto the "large" prime order subgroup we have to do cofactor clearing, that is we have to multiply the point with the cofactor 4. We get*

$$[4](6, 8) = \mathcal{O}$$

*so the hash value is still not right. We therefore have to increment the counter two times again, until we finally find a correct hash to $\mathbb{G}_1$*

```
sage: try_hash('10011001111010110100000111','0100')
(0, '1')
sage: try_hash('10011001111010110100000111','0101')
(12, '0')
```

*Since $12^3 + 8 \cdot 12 + 8 = 12$ and we have $\sqrt{12} \in \{5, 8\}$, we found another valid x coordinate $x = 12$ for the curve point hash. Now since the sign bit of this hash is 0, we choose the smaller root $y = 5$ as the y-coordinate and get the hash*

$$H('10011001111010110100000111') = (12, 5)$$

*which is a valid curve point point on the tiny jubjub curve and in order to project it onto the "large" prime order subgroup we have to do cofactor clearing, that is we have to multiply the point with the cofactor* 4*. We get*

$$[4](12,5) = (8,5)$$

*So hashing the binary string* $'10011001111010110100000111'$ *onto* $\mathbb{G}_1$ *gives the hash value* $(8,5)$ *as a result.*

## 6.4 Constructing elliptic curves

Cryptographically secure elliptic curves like Secp256k1 from example XXX are known for quite some time. In the latest advancements of cryptography, it is however often necessary to design and instantiate elliptic curves from scratch, that satisfy certain very specific properties.

For example, in the context of SNARK development it was necessary to design a curve that can be efficiently implemented incide of a so called circuit, in order to enable primitives like elliptic curve signature schemes in a zero knowledge proof. Such a curve is give by the Baby-JubJub curve [XXX] and we have paralled its definition by introducing the tiny-JubJub curve from example XX. As we have seen those curves are instances of so called twisted Edwards curves and as such have easy to implement addition laws that work without branching. However we introduced the tiny-jubjub curve out of thin air, as we just gave the curve parameters without explaining how we came up with them.

Another requirement in the context of many so called pairing based zero knowledge proofing systems is the existing of a suitable, pairing friendly curve with a specified security level and a low embedding degree as defined in XXX. Famous examples are the BLS_12 or the NMT curves.

The major goal of this section is to explain the most important method to design elliptic curves with predefined properties from sratch, called the *complex multiplication method*. We will apply this method in section to synthezise a particular BLS_6 curve, the most insecure BLS_6 curve, which will serve as the main curve to build our pen and paper snarks on. As we will see, this curve has a "large" prime factor subgroup of order 13, which implies, that we can use our tiny-jubjub curve to implement certain elliptic curve cryptographic primitives in circuits over that BLS_6 curve.

Before we introduce the complex multiplication method, we have to explain a few properties of elliptic curves that are of key importants in understanding the complex multiplication method.

**The Trace of Frobenious** To understand the complex multiplication method of elliptic curve, we have to define the so called *trace* of an elliptic curve first.

We know from XXX that elliptic curves over finite fields are cyclic groups of finite order. An interesting question therefore is, if it is possible to estimate the number of elements that curve contains. Since an affine short Weierstraß curve consists of pairs $(x,y)$ of elements from a finite field $\mathbb{F}_q$ plus the point at infinity and the field $\mathbb{F}_q$ contains $q$ elements, the number of curve points can not be arbitrarily large, since it can contain at most $q^2+1$ many elements.

There is however a more precise estimation, usually called the **Hasse bound**. To understand it, let $E(\mathbb{F}_q)$ be an affine short Weierstraß curve over a finite field $\mathbb{F}_w$ of order $q$ and let $|E(\mathbb{F}_q)|$ be the order of the curve. Then there is an integer $t \in \mathbb{Z}$ called the **trace of Frobenious** of the curve, such that $|t| \leq 2\sqrt{q}$ and

$$|E(\mathbb{F})| = q+1-t \tag{6.18}$$

A positive trace therefore implies, that the curve contains less points then the underlying field and a negative trace means that the curve contains more point. However the estimation $|t| \leq 2\sqrt{q}$ implies that the difference is not very large in either direction and the number of elements in an elliptic curve is always approximately in the same order of magnitute as the size of the curve's basefield.

**Example 94.** *Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We know that it contains 9 curve points. Since the order of $\mathbb{F}_5$ is 5 we compute the trace of $E_1(\mathbb{F})$ to be $t = -3$, since the Hasse bound is given by*

$$9 = 5 + 1 - (-3)$$

*And indeed we have $|t| \leq 2\sqrt{q}$, since $\sqrt{5} > 2.23$ and $|-3| = 3 \leq 4.46 = 2 \cdot 2.23 < 2 \cdot \sqrt{5}$.*

**Example 95.** *To compute the trace of the tiny-jubjub curve, oberse from example XXX, that the order of PJJ_13 is 20. Since the order of $\mathbb{F}_{13}$ is 13, we can therefore use the Hasse bound and compute the trace as $t = -6$, since*

$$20 = 13 + 1 - (-6)$$

*Again we have $|t| \leq 2\sqrt{q}$, since $\sqrt{13} > 3.60$ and $|-6| = 6 \leq 7.20 = 2 \cdot 3.60 < 2 \cdot \sqrt{13}$.*

**Example 96.** *To compute the trace of Secp256k1, recall from example XXX, that this curve is defined over a prime field with p elements and that the order of that group is given by r, with*

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$
$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

*Using the Hesse bound $r = p + 1 - t$, we therefore compute $t = p + 1 - r$, which gives the trace of curve Secp256k1 as*

$$t = 432420386565659656852420866390673177327$$

*So as we can see Secp256k1 contains less elements then its underlying field. However the difference is tiny, since the order of Secp256k1 is in the same order of magnitute as the order of the underlying field. Compared to p and r, t is tiny.*

```
sage: p = 115792089237316195423570985008687907853269984665640564
   039457584007908834671663
sage: r = 115792089237316195423570985008687907852837564279074904
   382605163141518161494337
sage: t = p + 1 -r
sage: t.nbits()
129
sage: abs(RR(t)) <= 2*sqrt(RR(p))
True
```

**The $j$-invariant** As we have seen in XXX two elliptic curve $E_1(\mathbb{F})$ defined by $y^2 = x^3 + ax + b$ and $E_2(\mathbb{F})$ defined by $y^2 + a'x + b'$ are strictly isomorphic, if and only if there is a quadratic residue $d \in \mathbb{F}$, such that $a' = ad^2$ and $b' = bd^3$.

There is however a more general way to classify elliptic curves over finite fields $\mathbb{F}_q$, based on the so called $j$-invariant of an elliptic curve:

$$j(E(\mathbb{F}_q)) = (1728 \bmod q)\frac{4 \cdot a^3}{4 \cdot a^3 + (27 \bmod q) \cdot b^2} \tag{6.19}$$

with $j(E(\mathbb{F}_q)) \in \mathbb{F}_q$. We will not go into the details of the $j$-invriant, but state only, that two elliptic curves $E_1(\mathbb{F})$ and $E_2(\mathbb{F}')$ are isomorphic over the algebraic closures of $\mathbb{F}$ and $\mathbb{F}'$, if and only if $\overline{\mathbb{F}} = \overline{\mathbb{F}'}$ and $j(E_1) = j(E_2)$.

So the $j$-invariant is an important tool to clasify elliptic curves and it is needed in the complex multuiplication method to decide on an actual curve instantiation, that implements abstractly choosen properties.

**Example 97.** *Consider the elliptic curve $E_1(\mathbb{F}_5)$ from example XXX. We compute its $j$-invariant as*

$$\begin{aligned} j(E_1(\mathbb{F}_5)) &= (1728 \; mod \; 5) \frac{4 \cdot 1^3}{4 \cdot 1^3 + (27 \; mod \; 5) \cdot 1^2} \\ &= 3 \frac{4}{4+2} \\ &= 3 \cdot 4 & = 2 \end{aligned}$$

**Example 98.** *Consider the elliptic curve PJJ_13 from example XXX. We compute its $j$-invariant as*

$$\begin{aligned} j(E_1(\mathbb{F}_5)) &= (1728 \; mod \; 13) \frac{4 \cdot 8^3}{4 \cdot 8^3 + (27 \; mod \; 13) \cdot 8^2} \\ &= 12 \cdot \frac{4 \cdot 5}{4 \cdot 5 + 1 \cdot 12} \\ &= 12 \cdot \frac{7}{7+12} \\ &= 12 \cdot 7 \cdot 6^{-1} \\ &= 12 \cdot 7 \cdot 11 \\ & 01 \end{aligned}$$

**Example 99.** *Consider Secp256k1 from example XXX. We compute its $j$-invariant using sage:*

```
sage: p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
sage: F = GF(p)
sage: j = F(1728)*((F(4)*F(0)^3)/(F(4)*F(0)^3+F(27)*F(7)^2))
sage: j == F(0)
True
```

**The Complex Multiplication Method** As we have seen in the previous sections, elliptic curves have various defining properties, like their order and their prime factors, the embedding degree, or the cardinality of the base field. The so called *complex multiplication* (CM) gives a practical method for constructing elliptic curves with pre-defined restrictions on the order and the base field.

The method usually starts by choosing a base field $\mathbb{F}_q$ of the curve $E(\mathbb{F}_q)$ we want to construct, such that $q = p^m$ for some prime number $p$ and counting number $m \in \mathbb{N}$ with $m \geq 1$. We assume $p > 3$ to simplify things in what follows.

Next the trace of Frobenious $t \in \mathbb{Z}$ of the curve is choosen, such that $p$ and $t$ are coprime, i.e. such that $gcd(p,t) = 0$ holds true. The choice of $t$ also defines the curves order $r$, since

$r = p+1-t$ by the Hasse bound XXX, so choosing $t$, will define the large order subgroup as well as all small cofactors. $r$ has to be defined in such a way, that the elliptic curve meats the security requirements of the application it is designed for.

Note that the choice of $p$ and $t$ also determines the embedding degree $k$ of any prime order subgroup of the curve, since $k$ is defined as the smallest number, such that the prime order $n$ divides the number $q^k - 1$.

In order for the complex multiplication method to work, both $q$ and $t$ can not be arbitrary, but must be choosen in such a way that two additional integers $D \in \mathbb{Z}$ and $v \in \mathbb{Z}$ exist, such that $D < 0$ as well as $D \bmod 4 = 0$ or $D \bmod 4 = 1$ and the equation

$$4q = t^2 + |D|v^2 \tag{6.20}$$

holds. If those numbers exist, we call $D$ the *CM-discriminant* and we know that we can construct a curve $E(\mathbb{F}_q)$ over a finite field $\mathbb{F}_q$, such that the order of the curve is $|E(\mathbb{F}_q)| = q+1-t$.

It is the content of the complex multiplication method to actually construct such a curve, that is finding the parameters $a$ and $b$ from $\mathbb{F}_q$ in the defining Weiertraß equation, such that the curve has the desired order $r$.

Finding solutions to equation XXX, can be achieved in different ways, which we will not look much into. In general it can be said, that there are well known constrctions for elliptic curve families like the BLS (ECT) families, that provides families of solutions. In what follows we will look at one type curves the BLS-family, which gives an entire rane of solutions.

Assuming that proper parameters $q$, $t$, $D$ and $v$ are found, we have to compute the so called *Hilbert class polynomial* $H_D \in \mathbb{Z}[x]$ of the CM-discriminant $D$, which is a polynomial with integer coefficients. To do so, we first have to compute the following set:

$$\begin{aligned} ICG(D) = \{(A,B,C) \mid A,B,C \in \mathbb{Z}, D = B^2 - 4AC, gcd(A,B,C) = 1, \\ |B| \leq A \leq \sqrt{\frac{|D|}{3}}, A \leq C, \text{ if } B < 0 \text{ then } |B| < A < C\} \end{aligned}$$

One way to compute this set, is to first compute the integer $A_{max} = Floor(\sqrt{\frac{|D|}{3}})$, then loop through all the integers $A$ in the range $[0,\ldots,A_{max}]$ as well as through all the integers $B$ in the range $[-A_{max},\ldots,A_{max}]$ and to see if there is an integer $C$, that satisfies $D = B^2 - 4AC$ and the rest of the requirements in XXX.

To compute the Hilbert class polynomial, the so called *j-function* (or $j$-invariant) is needed, which is a complex function defined on the upper half $\mathbb{H}$ of the complex plane $\mathbb{C}$, usually written as

$$j : \mathbb{H} \to \mathbb{C} \tag{6.21}$$

Roughly speaking what this means is that the $j$-functions takes complex numbers $(x + i \cdot y)$ with positive imaginary part $y > 0$ as inputs and returns a complex number $j(x + i \cdot y)$ as result.

For the sake of this book it is not importanted to actually understand the $j$-function and we can use sage to compute it in a similar way as we would use sage to computer any other well known function. It should be noted however, that the computation of the $j$-function in sage is sometimes prone to precision errors. For example the $j$-function has a root in $\frac{-1+i\sqrt{3}}{2}$, which sage only approximates. Therefore using sage to compute the $j$-function, we need to take precision loss into account and eventually round to the nearest integer.

```
sage: z = ComplexField(100)(0,1)
sage: z # (0+1i)
```

```
1.0000000000000000000000000000*I
sage: elliptic_j(z)
1728.0000000000000000000000000
sage: # j-function only defined for positive imaginary arguments
sage: z = ComplexField(100)(1,-1)
sage: try:
....:     elliptic_j(z)
....: except PariError:
....:     pass
sage: # root at (-1+i sqrt(3))/2
sage: z = ComplexField(100)(-1,sqrt(3))/2
sage: elliptic_j(z)
-2.6445453750358706361219364880e-88
sage: elliptic_j(z).imag().round()
0
sage: elliptic_j(z).real().round()
0
```

With a way to compute the $j$-function and the precomputed set $ICG(D)$ at hand, we can now compute the Hilbert class polynomial as

$$H_D(x) = \Pi_{(A,B,C)\in ICG(D)}\left(x - j\left(\frac{-B+\sqrt{D}}{2A}\right)\right) \tag{6.22}$$

So what we do is we loop over all elements $(A,B,C)$ from the set $ICG(D)$ and compute the $j$-function at the point $\frac{-B+\sqrt{D}}{2A}$, where $D$ is the CM-discriminant that we decided in a previous step. The result then defines a factor of the Hilbert class polynomial and all factors are multiplied together.

It can be shown, that the Hilbert class polynomial is an integer polynomial, but actual computations need high pecision arithmetics to avoid approximation errors, that usually occure in computer approximations of the $j$-function as shown above. So in case the calculated Hilbert class polynomial does not have integer coefficients, we need to round the result to the nearest integer. Given that the precision we used was high enough, the result will be correct.

In the next step we use the Hilbert class polynomial $H_D \in \mathbb{Z}[x]$ and project it to a polynomial $H_{D,q} \in \mathbb{F}_q[x]$ with coefficients in the base field $\mathbb{F}_q$ as choosen in the first step. We do this by simply computing the new coefficients as the old coefficients modulus $p$, that is if $H_D(x) = a_m x^m + a_{m-1}x^{m-1} + \ldots + a_1 x + a_0$ we compute the $q$-modulus of each coefficient $\tilde{a}_j = a_j \bmod p$ which defines the *projected Hilbert class polynomial* as

$$H_{D,p}(x) = \tilde{a}_m x^m + \tilde{a}_{m-1}x^{m-1} + \ldots + \tilde{a}_1 x + \tilde{a}_0$$

We then search for roots of $H_{D,p}$, since every root $j_0$ of $H_{D,p}$ defines a family of elliptic curves over $\mathbb{F}_q$, which all have a $j$-invariant XXX equal to $j_0$. We can pick any root and all of them will lead to proper curves eventually.

However some of the curves with the correct $j$-invariant might have an order different from the one we initially decided on. So we need a way to decide on a curve with the correct order.

To compute such a curve, we have to distinguish a few different cases, based on our choice of the root $j_0$ and of the CM-discriminant $D$. If $j_0 \neq 0$ or $j_0 \neq 1728 \bmod q$ we compute

$c_1 = \frac{j_0}{(1728 \bmod q) - j_0}$ and then we chose some arbitrary quadratic non-residue $c_2 \in \mathbb{F}_q$ and some arbitrary cubic non residue $c_3 \in \mathbb{F}_q$.

The following table is guranteed to define a curve with the correct order $r = q + 1 - t$, for the trace of Frobenious $t$ we initally decided on:

- Case $j_0 \neq 0$ and $j_0 \neq 1728 \bmod q$. A curve with the correct order is defined by one of the following equations

$$y^2 = x^3 + 3c_1 x + 2c_1 \quad \text{or} \quad y^2 = x^3 + 3c_1 c_2^2 x + 2c_1 c_2^3 \tag{6.23}$$

- Case $j_0 = 0$ and $D \neq -3$. A curve with the correct order is defined by one of the following equations

$$y^2 = x^3 + 1 \quad \text{or} \quad y^2 = x^3 + c_2^3 \tag{6.24}$$

- Case $j_0 = 0$ and $D = -3$. A curve with the correct order is defined by one of the following equations

$$\begin{gathered} y^2 = x^3 + 1 \quad \text{or} \quad y^2 = x^3 + c_2^3 \quad \text{or} \\ y^2 = x^3 + c_3^2 \quad \text{or} \quad y^2 = c_3^2 c_2^3 \quad \text{or} \\ y^2 = x^3 + c_3^{-2} \quad \text{or} \quad y^2 = x^3 + c_3^{-2} c_2^3 \end{gathered}$$

- Case $j_0 = 1728 \bmod q$ and $D \neq -4$. A curve with the correct order is defined by one of the following equations

$$y^2 = x^3 + x \quad \text{or} \quad y^2 = x^3 + c_2^2 x \tag{6.25}$$

- Case $j_0 = 1728 \bmod q$ and $D = -4$. A curve with the correct order is defined by one of the following equations

$$\begin{gathered} y^2 = x^3 + x \quad \text{or} \quad y^2 = x^3 + c_2 x \quad \text{or} \\ y^2 = x^3 + c_2^2 x \quad \text{or} \quad y^2 = x^3 + c_2^3 x \end{gathered}$$

To decide the proper defining Weierstraß equation, we therefore have to compute the order of any of the potential curves above and then choose the one that fits out initial requirements. Since it can be shown that the Hilbert class polynomials for the CM-discriminants $D = -3$ and $D = -4$ are given by $H_{-3,q}(x) = x$ and $H_{-4,q} = x - (1728 \bmod q)$ (EXERCISE) the previous cases are exhaustive.

To summarize, using the complex multiplication method, it is possible to synthezise elliptic curve with predefined order over predefined base fields from scratch. However the curves that are constructed this way are just some representatives of a larger class of curves, all of which have the same order. In applications it is therefore sometimes more advantageous to choose a different representative from that class. To do so recall from XXX, that any curve defined by the Weierstraß equation $y^2 = x^3 + axb$ is isomorphic to a curve of the form $y^2 = x^3 + ad^2 x + bd^3$ for some quadratic residue $d \in \mathbb{F}_q$.

So in order to find a nice representative (e.g. with small parameters $a$ and $b$) in a last step, the designer might choose a quadratic residue $d$ such that the transformed curve looks the way they wanted it.

**Example 100.** *Consider curve $E_1(\mathbb{F}_5)$ from example XXX. We want to use the complex multiplication method to derive that curve from scratch. Since $E_1(\mathbb{F}_5)$ is a curve of order $r = 9$ over the prime field of order $q = 5$, we know from example XX that its trace of Frobenius is $t = -3$, which also implies that $q$ and $|t|$ are coprime.*

*We then have to find parameters $D, v \in \mathbb{Z}$ with $D < 0$ and $D$ mod $4 = 0$ or $D$ mod $4 = 1$, such that $4q = t^2 + |D|v^2$ holds. We get*

$$\begin{aligned} 4q &= t^2 + |D|v^2 && \Rightarrow \\ 20 &= (-3)^2 + |D|v^2 && \Leftrightarrow \\ 11 &= |D|v^2 \end{aligned}$$

*Now since 11 is a prime number, the only solution is $|D| = 11$ and $v = 1$ here. So $D = -11$ and since the Euklidean division of $-11$ by 4 is $-11 = -3 \cdot 4 + 1$ we have $-11$ mod $4 = 1$, which shows that $D = -11$ is a proper choice.*

*In the next step, we have to compute the Hilbert class polynomial $H_{-11}$ and to do so, we first have to find the set $ICG(D)$. To comute that set, observe, that since $\sqrt{\frac{|D|}{3}} \approx 1.915 < 2$, we know from $A \leq \sqrt{\frac{|D|}{3}}$ and $A \in \mathbb{Z}$ that $A$ must be either 0 or 1.*

*For $A = 0$, we know $B = 0$ from the constraint $|B| \leq A$, but in this case there can be no $C$ satisfying $-11 = B^2 - 4AC$. So we try $A = 1$ and deduce $B \in \{-1, 0, 1\}$ from the constraint $|B| \leq A$. The case $B = -1$ can be excluded since then $B < 0$ has to imply $|B| < A$. In addition, the case $B = 0$ can be exclude as there can be integer $C$ with $-11 = -4C$ since 11 is a prime number.*

*This leaves the case $B = 1$ and we compute $C = 3$ from the equation $-11 = 1^2 - 4C$, which gives the solution $(A, B, C) = (1, 1, 3)$ and we get*

$$ICG(D) = \{(1, 1, 3)\}$$

*With the set $ICG(D)$ at hand we can compute the Hilbert class polynomial of $D = -11$. To do so, we have to insert the term $\frac{-1+\sqrt{-11}}{2 \cdot 1}$ into the j-function. To do so first observe that $\sqrt{-11} = i\sqrt{11}$, where $i$ is the imaginary unit, defined by $i^2 = -1$. Using this, we can invoke sagemath to compute the j-invariant and get*

$$H_{-11}(x) = x - j\left(\frac{-1 + i\sqrt{11}}{2}\right) = x + 32768$$

*So as we can see, in this particular case, the Hilbert class polynomial is a linear function with a single integer corefficiant. In the next step we have to project it onto a polynomial from $\mathbb{F}_5[x]$, by computing the modular 5 remainder of the corefficuents 1 and 32768. We get 32768 mod $5 = 3$ from which follows that the projected Hilbert class polynomial is*

$$H_{-11,5}(x) = x + 3$$

*considered as a polynomial from $\mathbb{F}_5[x]$. As we can see the only root of this polynomial is $j = 2$, since $H_{-11,5}(2) = 2 + 3 = 0$. We therefore have a situation with $j \neq 0$ and $j \neq 1728$, which tells us that we have to compute the parameter*

$$c_1 = \frac{2}{1728 - 2}$$

*in modular 5 arithmetics. Since* $1728 \bmod 5 = 3$*, we get* $c_1 = 2$*. Then we have to check if the curve* $E(\mathbb{F}_5)$ *defined by the Weierstraß* $y^2 = x^3 + 3 \cdot 2x + 2 \cdot 2$ *has the correct order. We invoke sage and find that the order is indeed* 9*, so it is a curve with the required parameters and we are done.*

*Note however that in real world applications, it might be usefull to choose parameters a and b that have certain properties, e.g. to be a small as possible. As we know from XXX, choosing any quadratic residue* $d \in \mathbb{F}_5$ *gives a curve of the same order defined by* $y^2 = x^2 + ak^2x + bk^3$*. Since* 4 *is a quadratic residue in* $\mathbb{F}_4$*, we can transform the curve defined by* $y^2 = x^3 + x + 4$ *into the curve* $y^2 = x^3 + 4^2 + 4 \cdot 4^3$ *which gives*

$$y^2 = x^3 + x + 1$$

*which is the curve* $E_1(\mathbb{F}_5)$*, that we used extensively throughout this book. So using the complex multiplication method, we were able to derive a curve with spacific properties from scratch.*

**Example 101.** *Consider the tiny jubjub curve TJJ_13 from example XXX. We want to use the complex multiplication method to derive that curve from scratch. Since TJJ_13 is a curve of order* $r = 20$ *over the prime field of order* $q = 13$*, we know from example XX that its trace of Frobenious is* $t = -6$*, which also implies that q and* $|t|$ *are coprime.*

*We then have to find parameters* $D, v \in \mathbb{Z}$ *with* $D < 0$ *and* $D \bmod 4 = 0$ *or* $D \bmod 4 = 1$*, such that* $4q = t^2 + |D|v^2$ *holds. We get*

$$\begin{aligned}
4q &= t^2 + |D|v^2 && \Rightarrow \\
4 \cdot 13 &= (-6)^2 + |D|v^2 && \Rightarrow \\
52 &= 36 + |D|v^2 && \Leftrightarrow \\
16 &= |D|v^2
\end{aligned}$$

*This equation has two solutions for* $(D, v)$*, given by* $(-4, \pm 2)$ *and* $(-16, \pm 1)$*. Now looking at the first solution, we know that the case* $D = -4$ *impplies* $j = 1728$ *and the constructed curve is defined by a Weierstraß equation XXX that has a vanishing parameter* $b = 0$*. We can therforefore conclude that choosing* $D = -4$ *will not help us reconstructing TJJ_13. It will produce curves with order* 20*, just not the one we are looking for.*

*So we choose the second solution* $D = -16$ *and in the next step, we have to compute the Hilbert class polynomial* $H_{-16}$*. To do so, we first have to find the set* $ICG(D)$*. To comute that set, observe, that since* $\sqrt{\frac{|-16|}{3}} \approx 2.31 < 3$*, we know from* $A \leq \sqrt{\frac{|-16|}{3}}$ *and* $A \in \mathbb{Z}$ *that A must be in the range* 0..2*. So we loop through all possible values of A and through all possible values of B under the constraints* $|B| \leq A$ *and if* $B < 0$ *then* $|B| < A$ *and the compute potential C's from* $-16 = B^2 - 4AC$*. We get the following two solution* $(1, 0, 4)$ *and* $(2, 0, 2)$*, giving we get*

$$ICG(D) = \{(1,0,4), (2,0,2)\}$$

*With the set* $ICG(D)$ *at hand we can compute the Hilbert class polynomial of* $D = -16$*. We can invoke sagemath to compute the j-invariant and get*

$$\begin{aligned}
H_{-16}(x) &= \left(x - j\left(\frac{i\sqrt{16}}{2}\right)\right)\left(x - j\left(\frac{i\sqrt{16}}{4}\right)\right) \\
&= (x - 287496)(x - 1728)
\end{aligned}$$

*So as we can see, in this particular case, the Hilbert class polynomial is a quadratic function with two integer corefficiant. In the next step we have to project it onto a polynomial from* $\mathbb{F}_5[x]$, *by computing the modular* 5 *remainder of the corefficuents* 1, 287496 *and* 1728. *We get* 287496 *mod* 13 = 1 *and* 1728 *mod* 13 = 2 *from which follows that the projected Hilbert class polynomial is*

$$H_{-11,5}(x) = (x-1)(x-12) = (x+12)(x+1)$$

*considered as a polynomial from* $\mathbb{F}_5[x]$. *So we have two roots given by* $j = 1$ *and* $j = 12$. *We alrady know that* $j = 12$ *is the wrong root to construct the tiny jubjub curve, since* 1728 *mod* 13 = 2 *and that case can not construct a curbe with* $b \neq 0$. *So we choose* $j = 1$.

*Another way to decide the proper root, is to compute the* $j$*-invariant of the tiny-jubjub curve. We get*

$$\begin{aligned} j(TJJ\_13) &= 12\frac{4\cdot 8^3}{4\cdot 8^3 + 1\cdot 8^2} \\ &= 12\frac{4\cdot 5}{4\cdot 5+12} \\ &= 12\frac{7}{7+12} \\ &= 12\frac{7}{7+12} \\ &= 1 \end{aligned}$$

*which is equal to the root* $j = 1$ *of the Hilbert class polynomial* $H_{-16,13}$ *as expected. We therefore have a situation with* $j \neq 0$ *and* $j \neq 1728$, *which tells us that we have to compute the parameter*

$$c_1 = \frac{1}{12-1} = 6$$

*in modular* 5 *arithmetics. Since* 1728 *mod* 13 = 12, *we get* $c_1 = 6$. *Then we have to check if the curve* $E(\mathbb{F}_5)$ *defined by the Weierstraß* $y^2 = x^3 + 3\cdot 6x + 2\cdot 6$ *which is equivalent to*

$$y^2 = x^3 + 5x + 12$$

*has the correct order. We invoke sage and find that the order is* 8, *which pmplies that the trace of this curve is* 6 *not* $-6$ *as required. So we have to consider the second possiblity and choose some quadratic non-residue* $c_2 \in \mathbb{F}_{13}$. *We choose* $c_2 = 5$ *and compute the Weierstraß equation* $y^2 = x^3 + 5c_2^2 + 12c_2^3$ *as*

$$y^2 = x^3 + 8x + 5$$

*We invoke sage and find that the order is* 20, *which is indeed the correct one. As we know from XXX, choosing any quadratic residue* $d \in \mathbb{F}_5$ *gives a curve of the same order defined by* $y^2 = x^2 + ad^2x + bd^3$. *Since* 12 *is a quadratic residue in* $\mathbb{F}_{13}$, *we can transform the curve defined by* $y^2 = x^3 + 8x + 5$ *into the curve* $y^2 = x^3 + 12^2\cdot 8 + 5\cdot 12^3$ *which gives*

$$y^2 = x^3 + 8x + 8$$

*which is the ziny jubjub curve, that we used extensively throughout this book. So using the complex multiplication method, we were able to derive a curve with spacific properties from scratch.*

**Example 102.** *To consider a real world example, we want to use the complex multiplication method in combination with sage to compute Secp256k1 from scratch. So by example XXX, we decided to compute an elliptic curve over a prime field $\mathbb{F}_p$ of order $r$ for the security parameters*

$$p = 115792089237316195423570985008687907853269984665640564039457584007908834671663$$
$$r = 115792089237316195423570985008687907852837564279074904382605163141518161494337$$

*which, according to example XXX gives the trace of Frobeniois $t = 432420386565659656852420866390673177327$. We also decided that we want a curve of the form $y^2 = x^3 + b$, that is we want the parameter $a$ to be zero. This implies, the $j$-invariant of our curve must be zero.*

*In a first step we have to find a CM-discriminant $D$ and some integer $v$, such that the equation*

$$4p = t^2 + |D|v^2$$

*is satisfied. Since we aim for a vanishing $j$-invariant, the first thing to try is $D = -3$. In this case we can compute $v^2 = (4p - t^2)$ and if $v^2$ happens to be an integers that has a square root $v$, we are done. Invoking sage we compute*

```
sage: D = -3
sage: p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
sage: r = 115792089237316195423570985008687907852837564279074904382605163141518161494337
sage: t = p+1-r
sage: v_sqr = (4*p - t^2)/abs(D)
sage: v_sqr.is_integer()
True
sage: v = sqrt(v_sqr)
sage: v.is_integer()
True
sage: 4*p == t^2 + abs(D)*v^2
True
sage: v
303414439467246543595250775667605759171
```

*So indeed the pair $(D, v) = (-3, 303414439467246543595250775667605759171)$ solves the equation, which tells us that there is a curve of order $r$ over a prime field of order $p$, defined by a Weierstraß equation $y^2 = x^3 + b$ for some $b \in \mathbb{F}_p$. So we need to compute $b$.*

*Now for $D = -3$ we already know that the associated Hilbert class polynomial is given by $H_{-3}(x) = x$, which gives the projected Hilbert class polynomial as $H_{-3,p} = x$ and the $j$-invariant of our curve is guranteed to be $j = 0$. Now looking into table XXX, we see that there are 6 possible cases to construct a curve with the correct order $r$. In order to construct the curves of those case we have to choose some arbitrary quadratic and cubic non residue. So we loop through $\mathbb{F}_p$ to find them, invoking sage:*

```
sage: F = GF(p)
sage: for c2 in F:
....:     try: # quadratic residue
....:         _ = c2.nth_root(2)
....:     except ValueError: # quadratic non residue
....:         break
```

```
sage: c2
3
sage: for c3 in F:
....:     try:
....:         _ = c3.nth_root(3)
....:     except ValueError:
....:         break
sage: c3
2
```

*So we found the quadratic non residue $c_2 = 3$ and the cubic non residue $c_3 = 2$. Using those numbers we check the six cases against the the expected order r of the curve we want to syntesize:*

```
sage: C1 = EllipticCurve(F,[0,1])
sage: C1.order() == r
False
sage: C2 = EllipticCurve(F,[0,c2^3])
sage: C2.order() == r
False
sage: C3 = EllipticCurve(F,[0,c3^2])
sage: C3.order() == r
False
sage: C4 = EllipticCurve(F,[0,c3^2*c2^3])
sage: C4.order() == r
False
sage: C5 = EllipticCurve(F,[0,c3^(-2)])
sage: C5.order() == r
False
sage: C6 = EllipticCurve(F,[0,c3^(-2)*c2^3])
sage: C6.order() == r
True
```

*So as expeced we found an elliptic curve of the correct order r over a prime field of size p. So in principal we are done, as we have found a curve with the same basic properties as Secp256k1. However the curve is defined by the equation*

$$y^2 = x^3 + 86844066927987146567678238756515930889952488499230423029593188005931626003754$$

*that use a very large parameter $b_1$, which might perform slow in certain algorithms. It is also not very elegant to be written down by hand. It might therefore be advantegous to find an isonmorphic curve with the smallest possible parameter $b_2$. So in order to find such a $b_2$, we have to choose a quadratic residue d, such that $b_2 = b_1 \cdot d^3$ is as small as possibl. To do so we rewrite the last equation into*

$$d = \sqrt[3]{\frac{b_2}{b_1}}$$

*and then invoke sage to loop through values $b_2 \in \mathbb{F}_p$ until it finds some number such that the quotient $\frac{b_2}{b_1}$ has a cube root d and this cube root itself is a quadratic residue.*

```
sage: b1=86844066927987146567678238756515930889952488499230423
    029593188005931626003754
```

```
sage: for b2 in F:
....:     try:
....:         d = (b2/b1).nth_root(3)
....:         try:
....:             _ = d.nth_root(2)
....:             if d != 0:
....:                 break
....:         except ValueError:
....:             pass
....:     except ValueError:
....:         pass
sage: b2
7
```

*So indeed the smallest possible value is $b_2 = 7$ and the defining Weierstrass equation of a curve over $\mathbb{F}_p$ with prime order $r$ is*

$$y^2 = x^3 + 7$$

*which we might call secp256k1. As we have seen the complex multiplication method is powerful enough to derive cryptographically secure curves like Secp256k1 from scratch.*

**The $BLS6\_6$ pen& paper curve** In this paragraph we to summarize our understanding of elliptic curves to derive our main pen & paper example for the rest of the book. To do so, we want to use the complex multiplication method, to derive a pairing friendly elliptic curve that has similar properties to curves that are used in actual cryptographic protocols. However we design the curve specifically to be useful in pen&paper examples, which mostly means that the curve should contain only a few points, such that we are able to derive exhaustive addition and pairing tables.

A well understood family of pairing friendly curves are the BLS curves (STUFF ABOUT THE HISTORY AND THE NAMING CONVENTION), which are derived in [XXX]. BLS curves are particular useful in our case if the embedding degree $k$ satisfies $k \equiv 6 \pmod 0$. Of course the smallest embedding degree $k$ that satisfies this congruency, is $k = 6$ and we therefore aim for a BLS6 curve as our main pen&paper example.

To apply the complex multiplication method from XXX, recall that this method starts with a definition of the base field $\mathbb{F}_{p^m}$ as well as the trace of Frobenious $t$ and the order of the curve. If the order $p^m + 1 - t$ is not a prime number, then what is necessary to control is the order $r$ of the largest prime factor group.

In the case of BLS_6 curves, the parameter $m$ is choosen to be 1, which means that the curves are defined over prime fields. All relevent parameters $p$, $t$ and $r$ are then themselfs parameterized by the following three polynomials

$$\begin{aligned}
r(x) &= \Phi_6(x) \\
t(x) &= x + 1 \\
q(x) &= \frac{1}{3}(x-1)^2(x^2 - x + 1) + x
\end{aligned}$$

where $\Phi_6$ is the 6-th cyclotomic polynomial and $x \in \mathbb{N}$ is a parameter that the designer has to choose in such a way that the evaluation of $p$, $t$ and $r$ at the point $x$ gives integers that have the proper size to meet the security requirements of the curve that they want to design. It is then guaranteed that the complex multiplication method can be used in combination with those parameters to define an elliptic curve with CM-discriminant $D=-3$ and embedding degree $k=6$ and curve equation $y^2=x^3+b$ for some $b\in\mathbb{F}_p$.

For example if the curve should target the 128-bit security level, due to the Pholaard-rho attack (TODO) the parameter $r$ shouls be prime number of at least 256 bits.

In order to design the smallest, most unsecure BLS_6 curve, we therefore have to find a parameter $x$, such that $r(x)$, $t(x)$ and $q(x)$ are the smallest natural numbers, that satisfy $q(x)>3$ and $r(x)>3$.

We therefore initiate the design process of our $BLS6$ curve by looking-up the 6-th cyclotomic polynimial which is $\Phi_6=x^2-x+1$ and then insert small values for $x$ into the defining polynomials $r,t,q$. We get the following results:

$$\begin{array}{lcr}
x=1 & (r(x),t(x),q(x)) & (1,2,1)\\
x=2 & (r(x),t(x),q(x)) & (3,3,3)\\
x=3 & (r(x),t(x),q(x)) & (7,4,\frac{37}{3})\\
x=4 & (r(x),t(x),q(x)) & (13,5,43)
\end{array}$$

Since $q(1)=1$ is not a prime number, the first $x$ that gives a proper curve is $x=2$. However such a curve would be defined over a base field of characteristic 3 and we would rather like to avoid that. We therefore find $x=4$, which defines a curve over the prime field of characteristic 43, that has a trace of Frobenious $t=5$ and a larger order prime group of size $r=13$.

Since the prime field $\mathbb{F}_{43}$ has 43 elements and 43's binary representation is $43_2=101011$, which are 6 digits, the name of our pen&paper curve should be $BLS6$_6, since its is common behaviour to name a BLS curve by its embedding degree and the bit-length of the modulus in the base field. We call $BLS6$_6 the **moon-math-curve**.

Racalling from XXX, we know that the Hasse bound implies that $BLS6$_6 will contain exactly 39 elements. Since the prime factorization of 39 is $39=3\cdot 13$, we have a "large" prime factor group of size 13 as expected and a small cofactor group of size 3. Fortunately a subgroup of order 13 is well suited for our purposes as 13 elements can be easily handled in the associated addition, scalar multiplication and pairing tables in a pen and paper style.

We can check that the embedding degree is indeed 6 as expected, since $k=6$ is the smallest number $k$ such that $r=13$ divides $43^k-1$.

```
sage: for k in range(1,42): # Fermat's little theorem
....:     if (43^k-1)%13 == 0:
....:         break
sage: k
6
```

In order to compute the defining equation $y^2=x^3+ax+b$ of BLS6-6, we use the complex multiplication method as described in XXX. The goal is to find $a,b\in\mathbb{F}_{43}$ representations, that are particulary nice to work with. The authors of XXX showed that the CM-discriminant of every BLS curve is $D=-3$ and indeed the equation

$$\begin{aligned}
4p &= t^2+|D|v^2 && \Rightarrow\\
4\cdot 43 &= 5^2+|D|v^2 && \Rightarrow\\
172 &= 25+|D|v^2 && \Leftrightarrow\\
49 &= |D|v^2
\end{aligned}$$

has the four solutions $(D,v)\in\{(-3,-7),(-3,7),(-49,-1),(-49,1)\}$ if $D$ is required to be negative, as expected. So $D=-3$ is indeed a proper CM-discriminant and we can deduce that the parameter $a$ has to be 0 and that the Hilbert class polynomial is given by

$$H_{-3,43}(x) = x$$

This implies that the $j$-invariant of $BLS6\_6$ is given by $j(BLS6\_6) = 0$. We therefore have to look at case XXX in table XXX to derive a parameter $b$. To decide the proper case for $j_0 = 0$ and $D = -3$, we therefore have to choose some arbitrary quadratic non residue $c_2$ and cubic non residue $c_3$ in $\mathbb{F}_{43}$. We choose $c_2 = 5$ and $c_3 = 36$. We check

```
sage: F43 = GF(43)
sage: c2 = F43(5)
....: try: # quadratic residue
....:     c2.nth_root(2)
....: except ValueError: # quadratic non residue
....:     c2
sage: c3 =F43(36)
....: try:
....:     c3.nth_root(3)
....: except ValueError:
....:     c3
```

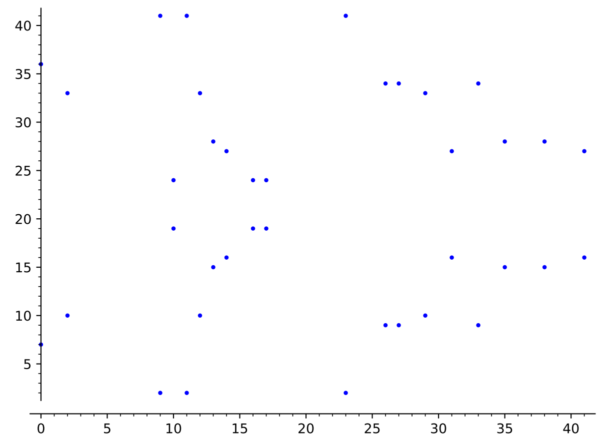
Using those numbers we check the six possible cases from XXX against the the expected order 39 of the curve we want to syntesize:

```
sage: BLS61 = EllipticCurve(F43,[0,1])
sage: BLS61.order() == 39
False
sage: BLS62 = EllipticCurve(F43,[0,c2^3])
sage: BLS62.order() == 39
False
sage: BLS63 = EllipticCurve(F43,[0,c3^2])
sage: BLS63.order() == 39
True
sage: BLS64 = EllipticCurve(F43,[0,c3^2*c2^3])
sage: BLS64.order() == 39
False
sage: BLS65 = EllipticCurve(F43,[0,c3^(-2)])
sage: BLS65.order() == 39
False
sage: BLS66 = EllipticCurve(F43,[0,c3^(-2)*c2^3])
sage: BLS66.order() == 39
False
sage: BLS6 = BLS63 # our BLS6 curve in the book
```

So as expeced we found an elliptic curve of the correct order 39 over a prime field of size 43, defined by the equation

$$BLS6\_6 := \{(x,y) \mid y^2 = x^3 + 6 \text{ for all } x,y \in \mathbb{F}_{43}\} \tag{6.26}$$

There are other choice for $b$ like $b = 10$ or $b = 23$, but all these curves are isomorphic and hence represent the same curve really but in a different way only. Since BLS6-6 only contains 39 points it is possible to give a visual impression of the curve:




As we can see our curve is somewhat nice, as it does not contain self inverse points that is points with $y = 0$. It follows that the addition law can be optimized, since the branch for those cases can be eliminated.

Summarizing the previous procedure, we have used the method of Barreto, Lynn and Scott to construct a pairing friendly elliptic curve of embedding degree 6. However in order to do elliptic curve cryptography on this curve note that since the order of $BLS6\_6$ is 39 its group of rational points is not a finite cyclic group of prime order. We therefore have to find a suitable subgroup as our main target and since $39 = 13 \cdot 3$, we know that the curve must contain a "large" prime order group of size 13 and a small cofactor group of order 3.

It is the content of the following step to construct this group. One way to do so is to find a generator. We can achieve this by choosing an abitrary element of the group that is not the point at infinity and then multiply that point with the cofactor of the groups order. If the result is not the point at infinity, the result will be a generator and if it is the point at infinity we have to choose a different element.

So in order to find a generator for the large order subgroup of size 13, we first notice that the cofactor of 13 is 3, since $39 = 3 \cdot 13$. We then need to construct an arbitray element from $BLS6\_6$. To do so in a pen and paper style, we can choose some $arbitraryx \in \mathbb{F}_{43}$ and see if there is some ssolution $y \in \mathbb{F}_{43}$ that satisfies the defining Weierstrass equation $y^2 = x^3 + 6$. We choose $x = 9$. Then $y = 2$ is a proper solution, since

$$
\begin{aligned}
y^2 &= x^3 + 6 && \Rightarrow \\
2^2 &= 9^3 + 6 && \Leftrightarrow \\
4 &= 4
\end{aligned}
$$

and this implies that $P = (9,2)$ is therefore a point on $BLS6\_6$. To see if we can project this point onto a generator of the large order prim group $BLS6\_6[13]$, we have to multiply $P$ with the cofactor, that is we have to compute $[3](9,2)$. After some computation (EXERCISE) we get $[3](9,2) = (13,15)$ and since this is not the point at infinity we know that $(13,15)$ must be a generator of $BLS6\_6[13]$. We write

$$
g_{BLS6\_6[13]} = (13,15) \tag{6.27}
$$

as we will need this generator in pairing computations all over the book. Since $g_{BLS6\_6[13]}$ is a generator, we can use it to construct the subgoup $BLS6\_6[13]$, by repeately addding the generator to itself. We use sage and get

```
sage: P = BLS6(9,2)
```
598

```
sage: Q = 3*P
sage: Q.xy()
(13, 15)
sage: BLS6_13 = []
sage: for x in range(0,13): # cyclic of order 13
....:     P = x*Q
....:     BLS6_13.append(P)
```

Repeadly adding a generator to itself as we just did, will generate small groups in logarithmic order with respect to the generator as explained in XXX. We therefore get the following description of the large prime order subgroup of $BLS6\_6$:

$$BLS6\_6[13] = \\ \{(13,15) \to (33,34) \to (38,15) \to (35,28) \to (26,34) \to (27,34) \to \\ (27,9) \to (26,9) \to (35,15) \to (38,28) \to (33,9) \to (13,28) \to \mathcal{O}\} \quad (6.28)$$

Having a logarithmic description of this group is temendously helpfull in pen and paper computations. To see that, observe that we know fromXXX that there is an exponential map from the scalar field $\mathbb{F}_{13}$ to $BLS6\_6[13]$ with respect to our generator

$$[\cdot]_{(13,15)} : \mathbb{F}_{13} \to BLS6\_6[13] \; ; \; x \mapsto [x](13,15)$$

which generates the group in logarithmic order. So for example we have $[1]_{(13,15)} = (13,15)$, $[7]_{(13,15)} = (27,9)$ and $[0]_{(13,15)} = \mathcal{O}$ and so on. The point for our purposes is, that we can use this representation to do computations in $BLS6\_6[13]$ efficiently in our head using XXX. For example

$$\begin{aligned}(27,34) \oplus (33,9) &= [6](13,15) \oplus [11](13,15) \\ &= [6+11](13,15) \\ &= [4](13,15) \\ &= (35,28)\end{aligned}$$

So XXX is really all we need to do computations in $BLS6\_6[13]$ in this book efficiently. However out of convinience, the following picture lists the entire addition table of that group. It might be useful in pen and paper computations:

| $\oplus$ | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{O}$ | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) |
| (13,15) | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ |
| (33,34) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) |
| (38,15) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) |
| (35,28) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) |
| (26,34) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) |
| (27,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) |
| (27,9) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) |
| (26,9) | (26,9) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) |
| (35,15) | (35,15) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) |
| (38,28) | (38,28) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) |
| (33,9) | (33,9) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) |
| (13,28) | (13,28) | $\mathcal{O}$ | (13,15) | (33,34) | (38,15) | (35,28) | (26,34) | (27,34) | (27,9) | (26,9) | (35,15) | (38,28) | (33,9) |

Now that we have constructed a "large" cyclic prime order subgroup of $BLS6\_6$ sutable for many pen and paper computations in elliptic curve cryptography, we have to look at how to do pairings in this context. We know that $BLS6\_6$ is a pairing friendly curve by design, since it has a small embedding degree $k = 6$. It is therefore possible to compute Weil pairings efficiently. However in order to do so, we have to decides the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ as explained in XXX.

Since $BLS6\_6$ has two non trivial subgroups it would be possible to use any of them as the $n$-torsion group. However in cryptography the only secure choice is to use the large prime order subgroup, which in our case is $BLS6\_6[13]$. we therefore decide to consider the 13-torsion and define

$$\mathbb{G}_1[13] = \{(13,15) \to (33,34) \to (38,15) \to (35,28) \to (26,34) \to (27,34) \to \\ (27,9) \to (26,9) \to (35,15) \to (38,28) \to (33,9) \to (13,28) \to \mathcal{O}\}$$

as the first argument for the Weil pairing function.

In order to construct the domain for the second argument, we need to construct $\mathbb{G}_2[13]$, which, according to the general theory should be defined by those elements $P$ of the full 13-torsion group $BLS6\_6[13]$, that are mapped to $43 \cdot P$ under the Frobenius endomorphism XXX.

To compute $\mathbb{G}_2[13]$ we therefore have to find the full 13-torsion group first. To do so, we use the technique from XXX, which tells us, that the full 13-torsion can be found in the curve extension

$$BLS6\_6 := \{(x,y) \mid y^2 = x^3 + 6 \text{ for all } x,y \in \mathbb{F}_{43^6}\} \tag{6.29}$$

over the extension field $\mathbb{F}_{43^6}$, since the embedding degree of $BLS6\_6$ is 6. So we have to construct $\mathbb{F}_{43^6}$, a field that contains 6321363049 many elements. In order to do so we use the procedure of XXX and start by choosing a non-reducible polynomial of degree 6 from the ring of polynomials $\mathbb{F}_{43}[t]$. We choose $p(t) = t^6 + 6$. Using sage we get

```
sage: F43 = GF(43)
sage: F43t.<t> = F43[]
sage: p = F43t(t^6+6)
sage: p.is_irreducible()
True
sage: F43_6.<v> = GF(43^6, name='v', modulus=p)
```

Recall from XXX that elements $x \in \mathbb{F}_{43^6}$ can be seen as polynomials $a_0 + a_1 v + a_2 v^2 + \ldots + a_5 v^5$ with the usual addition of polynomials and multiplication modulo $t^6 + 6$.

In order to compute $\mathbb{G}_2[13]$ we first have to extend $BLS6\_6$ to $\mathbb{F}_{43^6}$, that is we keep the defining equation but expend the domain from $\mathbb{F}_{43}$ to $\mathbb{F}_{43^6}$. After that we have to find at least one element $P$ from that curve, that is not the point at infinity, that is in the full 13-torsion and that satisfies the identity $\pi(P) = [43]P$. We can then use this element as our generator of $\mathbb{G}_2[13]$ and construct all other elements by repeated addition to itself.

Since $BLS6(\mathbb{F}_{43^6})$ contains 6321251664 elements, its not a good strategy to simply loop through all elements. Fortunately sage has a way to loop through elements from the torsion group directly. We get

```
sage: BLS6 = EllipticCurve (F43_6,[0 ,6]) # curve extension
sage: INF = BLS6(0) # point at infinity
sage: for P in INF.division_points(13): # full 13-torsion
....: # PI(P) == [q]P
....:     if P.order() == 13: # exclude point at infinity
....:         PiP = BLS6([a.frobenius() for a in P])
....:         qP = 43*P
```

```
....:            if PiP == qP:
....:                break
sage: P.xy()
(7*v^2, 16*v^3)
```

So we found an element from the full 13-torsion, that is in the Eigenspace of the Eigenvalue 43, which implies that it is an element of $\mathbb{G}_2[13]$. As $\mathbb{G}_2[13]$ is cyclic of prime order this element must be a generator and we write

$$g_{\mathbb{G}_2[13]} = (7v^2, 16v^3) \tag{6.30}$$

Wee can use this generator to compute $\mathbb{G}_2$ is logarithmic order with respect to $g_{\mathbb{G}_[13]}$. Using sage we get

```
sage: Q = BLS6(7*v^2,16*v^3)
sage: BLS6_13_2 = []
sage: for x in range(0,13):
....:     P = x*Q
....:     BLS6_13_2.append(P)
```

$$\begin{aligned}\mathbb{G}_2 = \{(7v^2,16v^3) &\to (10v^2,28v^3) \to (42v^2,16v^3) \to (37v^2,27v^3) \to \\ &(16v^2,28v^3) \to (17v^2,28v^3) \to (17v^2,15v^3) \to (16v^2,15v^3) \to \\ &\quad (37v^2,16v^3) \to (42v^2,27v^3) \to (10v^2,15v^3) \to (7v^2,27v^3) \to \mathcal{O}\}\end{aligned}$$

Again, having a logarithmic description of $\mathbb{G}_2[13]$ is temendously helpfull in pen and paper computations, as it reduces complicated computation in the extended curve to modular 13 arithmetics. For example

$$\begin{aligned}(17v^2,28v^3) \oplus (10v^2,15v^2) &= [6](7v^2,16v^3) \oplus [11](7v^2,16v^3) \\ &= [6+11](7v^2,16v^3) \\ &= [4](7v^2,16v^3) \\ &= (37v^2,27v^3)\end{aligned}$$

So XXX is really all we need to do computations in $\mathbb{G}_2[13]$ in this book efficiently.

To summerize the previous steps, we have found two subgroups $\mathbb{G}_1[13]$ as well as $\mathbb{G}_2[13]$ suitable to do Weil pairings on *BLS*6_6 as explained in XXX. Using the logarithmic order XXX of $\mathbb{G}_1[13]$, the logarithmic order XXX of $\mathbb{G}_2[13]$ and the bilinearity

$$e([k_1]g_{BLS6\_6[13]}, [k_2]g_{\mathbb{G}_2[13]}) = e(g_{BLS6\_6[13]}, g_{\mathbb{G}_2[13]})^{k_1 \cdot k_2}$$

we can do Weil pairings on *BLS*6_6 in a pen and paper style, observing that the Weil pairing between our two generators is given by the identity

$$e(g_{BLS6\_6[13]}, g_{\mathbb{G}_2[13]}) = 5v^5 + 16v^4 + 16v^3 + 15v^2 + 3v + 41$$

```
sage: g1 = BLS6([13,15])
sage: g2 = BLS6([7*v^2, 16*v^3])
sage: g1.weil_pairing(g2,13)
5*v^5 + 16*v^4 + 16*v^3 + 15*v^2 + 3*v + 41
```

**Hashing to the pairing groups** We give various constructions to hash into $\mathbb{G}_1$ and $\mathbb{G}_2$.

We start with hashing to the scalar field... TO APPEAR

Non of these techniques work for hashing into $\mathbb{G}_2$. We therefore implement Pederson's Hash for BLS6.

We start with $\mathbb{G}_1$. Our goal is to define an 12-bit bounded hash function

$$H_1 : \{0,1\}^{12} \to \mathbb{G}_1$$

Since $12 = 3\cdot 4$ we "randomly" select 4 uniformly distributed generators $\{(38,15),(35,28),(27,34),(38,28)\}$ from $\mathbb{G}_1$ and use the pseudo-random function from XXX. For every genrator we therefore have to choose a set of 4 randomly generated invertible elements from $\mathbb{F}_{13}$. We choose

$$\begin{array}{lcl} (38,15) & : & \{2,7,5,9\} \\ (35,28) & : & \{11,4,7,7\} \\ (27,34) & : & \{5,3,7,12\} \\ (38,28) & : & \{6,5,1,8\} \end{array}$$

So our hash function is computed like this:

$$\begin{aligned} H_1(x_{11},x_1,\ldots,x_0) = [2\cdot 7^{x_{11}}\cdot 5^{x_{10}}\cdot 9^{x_9}](38,15)+[11\cdot 4^{x_8}\cdot 7^{x_7}\cdot 7^{x_6}](35,28)+ \\ [5\cdot 3^{x_5}\cdot 7^{x_4}\cdot 12^{x_3}](27,34)+[6\cdot 5^{x_2}\cdot 1^{x_1}\cdot 8^{x_0}](38,28) \end{aligned}$$

Note that $a^x = 1$ whe $x = 0$ and hence those terms can be omitted in the computation. In particular the hash of the 12-bit zero string is given by

$$\begin{aligned} WRONG-ORDERING-REDOH_1(0) = [2](38,15)+[11](35,28)+[5](27,34)+[6](38,28) = \\ (27,34)+(26,34)+(35,28)+(26,9) = (33,9)+(13,28) = (38,28) \end{aligned}$$

The hash of 011010101100 is given by

$$\begin{aligned} H_1(011010101100) = WRONG-ORDERING-REDO \\ [2\cdot 7^0\cdot 5^1\cdot 9^1](38,15)+[11\cdot 4^0\cdot 7^1\cdot 7^0](35,28)+[5\cdot 3^1\cdot 7^0\cdot 12^1](27,34)+[6\cdot 5^1\cdot 1^0\cdot 8^0](38,28) = \\ [2\cdot 5\cdot 9](38,15)+[11\cdot 7](35,28)+[5\cdot 3\cdot 12](27,34)+[6\cdot 5](38,28) = \\ [12](38,15)+[12](35,28)+[11](27,34)+[4](38,28) = \\ TO_APPEAR \end{aligned}$$

We can use the same technique to define a 12-bit bounded hash function in $\mathbb{G}_2$:

$$H_2 : \{0,1\}^{12} \to \mathbb{G}_2$$

Again we "randomly" select 4 uniformly distributed generators $\{(7v^2,16v^3),(42v^2,16v^3),(17v^2,15v^3),(10v^2,1$ from $\mathbb{G}_2$ and use the pseudo-random function from XXX. For every genrator we therefore have to choose a set of 4 randomly generated invertible elements from $\mathbb{F}_{13}$. We choose

$$\begin{array}{lcl} (7v^2,16v^3) & : & \{8,4,5,7\} \\ (42v^2,16v^3) & : & \{12,1,3,8\} \\ (17v^2,15v^3) & : & \{2,3,9,11\} \\ (10v^2,15v^3) & : & \{3,6,9,10\} \end{array}$$

So our hash function is computed like this:

$$H_1(x_{11}, x_{10}, \ldots, x_0) = [8 \cdot 4^{x_{11}} \cdot 5^{x_{10}} \cdot 7^{x_9}](7v^2, 16v^3) + [12 \cdot 1^{x_8} \cdot 3^{x_7} \cdot 8^{x_6}](42v^2, 16v^3) + \\ [2 \cdot 3^{x_5} \cdot 9^{x_4} \cdot 11^{x_3}](17v^2, 15v^3) + [3 \cdot 6^{x_2} \cdot 9^{x_1} \cdot 10^{x_0}](10v^2, 15v^3)$$

We extend this to a hash function that maps unbounded bitstring to $\mathbb{G}_2$ by precomposing with an actual haah function like $MD5$ and feet the first 12 bits of its outcome into our previously defined hash function.

$$TinyMD5_{\mathbb{G}_2} : \{0,1\}^* \to \mathbb{G}_2$$

with $TinyMD5_{\mathbb{G}_2}(s) = H_2(MD5(s)_0, \ldots MD5(s)_{11})$. For example, since $MD5(\text{''''}) = 0xd41d8cd98f00b204e98$ and the binary representation of the hexadecimal number $0x27e$ is 001001111110 we compute $TinyMD5_{\mathbb{G}_2}$ of the empty string as $TinyMD5_{\mathbb{G}_2}(\text{''''}) = H_2(MD5(s)_{11}, \ldots MD5(s)_0) = H2(001001111110) =$

# 7 Statements

As we have seen in the informal introduction XXX, a snarks is a short non-interactive argument of knowledge, where the knowledge-proof attests to the correctness of statements like "I know the prime factorization of a given number" or "I know the preimage to a given SHA3 digest value" and similar things. However human readable statements like those are imprecise and not very useful from a formal perspective.

In this chapter we therefore look more closely at ways to formalize statements in mathematically rigorous ways, useful for snark development. We start by introducing formal languages as a way to define statements properly. We will then look at algebraic circuits and rank-1 constraint systems as two particulary useful ways to define statements in certain formal languages.

As in many other parts of the book the emphesis is on an introduction from the developers point of view. Proper statement design should be of high priority in the development of snarks, since unintended true statements can lead to potentially severe and almost undetectable attacks on the applications of snarks.

## 7.1 Formal Languages

Formal languages provide the theoretical backround in which logical statements can be formulated in a logically regious way. Roughly speaking a formal language is nothing but a set of words, that are strings of letters taken from some alphabet and formed according to some defining rules of that language.

To be more precise, let $\Sigma$ be any set and $\Sigma^*$ the set of all finite tupels $(x_1,\ldots,x_n)$ of elements $x_j$ from $\Sigma$ including the empty tupel $(\ ) \in \Sigma^*$. Then a **formal language** $L$ is in its most general definition nothing but a subset of $\Sigma^*$. In this context, the set $\Sigma$ is called the **alphabet** of the language $L$, elements from $\Sigma$ are called letters and elements from $L$ are called **words**. The rules that specify which tupels from $\Sigma^*$ belong to the language and which don't, are called the **grammar** of the language.

**Checking Relations** In the context of snark development it is common to formalize the grammar of a laguage by a so called **checking relation**, which can be interpreted as a map

$$R : \Sigma^* \to \{TRUE, FALSE\} \tag{7.1}$$

that decides if a tupel $x \in \Sigma^*$ is an element of the language or not. In this case the language itself can be written as the set of all tupels that satisfies the grammar, i.e as:

$$L := \{x \in \Sigma^* \mid R(x) = TRUE\} \tag{7.2}$$

In our context a **statement** is then a claim, that language $L$ contains a word $x$, i.e a statement claims $x \in L$ and one way to constructively *proof* a staitment is to provided an actual **instance**, that is an actual word of the language.

Of course formal languages should not be confused with the more intuitive concept of a natural langual. The following examples will provide some intuition about formal languages.

**Example 103** (Alternating Binary strings)**.** *To consider a very basic formal language with an almost trivial grammar consider the set* $\{0,1\}$ *of the two letters* $0$ *and* $1$ *as our alphabet* $\Sigma$ *and imply the rule that a proper word must consist of alternating binary letters of arbitrary length.*

*Then the associated language* $L_{alt}$ *is the set of all finite binary tupels, where a* $1$ *must follow a* $0$ *and vice versa. So for example* $(1,0,1,0,1,0,1,0,1) \in L_{alt}$ *is a proper word as well as* $(0) \in L_{alt}$ *or the empty word* $(\,) \in L_{alt}$*. However the binary tupel* $(1,0,1,0,1,0,1,1,1) \in \{0,1\}^*$ *is not a proper word as it violates the grammer of* $L_{alt}$*. In addition the tupel* $(0,A,0,A,0,A,0)$ *is not a proper word as its letter are not from the proper alphabet.*

*Inside language* $L_{alt}$ *it makes sense to claim the following statement: "There exists an alternating string." One way to proof this statement would be by proving an actual instance, that is finding actual alternating string like* $x = (1,0,1)$*. Constructing a string like* $(1,0,1)$ *therefore proofs that statement "There exists an alternating string."*

*Atempting to write the grammar of this language in a more formal way, we can define the following checking relation:*

$$R : \{0,1\}^* \to \{TRUE, FALSE\}\ ;\ (x1, x_2, \ldots, x_n) \mapsto \begin{cases} TRUE & x_j \neq x_{j+1}\, \textit{for all}\ 1 \leq j < n \\ FALSE & \textit{else} \end{cases}$$

*We can use this relation to check if arbitrary binary tupels are words in* $L_{alt}$ *or not. For example* $R(1,0,1) = TRUE$*,* $R(0) = TRUE$ *and* $R() = TRUE$*, but* $R(1,1) = FALSE$ *and so on.*

**Example 104** (Programing Language)**.** *Programming languages are a very important class of formal languages. In this case the alphabet is usually (a subset) of the ASCII Table and the grammar is defined by the rules of the programming language's compiler. Words are then nothing but properly written computer programms that the compiler accepts. The compiler can therefore be interpreted as the checking relation.*

*To give an unusual example strange enough to highlight the point, consider the programing language Malbolge as defined in XXX. This language was specifically designed to be almost impossible to use and writing programs in this language is a difficult task. An intersting claim is therefore the statement: "There exists a computer program in Malbolge". As it turned out proofing this statement by providing an actual instance was not an easy task as it took two years after the introduction of Malbolge, to write a program that its compiler accepts. So for two years no one was able to proof the statement.*

*To look at this high level description more formally, we write* $L_{Malbolge}$ *for the language, that uses the ASCII table as its alphabet and words are tuples of ASCII letters that the Malbolge compiler accepts. Prooving the statement "There exists a computer program in Malbolge" is the equivalent to the task of finding some word* $x \in L_{Malbolge}$*. The string*

(=<'#9] 6ZY327Uv4−QsqpMn&+Ij"'E%e{Ab w=_:]Kw%o44Uqp0/Q?xNvL:'H%c#DD2∧WV>gY;dts76qKJImZkj

*is an example of such a proof as it is excepted by the Malbolge compiler and is compiled to an executable binary that displays "Hello, World." (See XXX)*

**Example 105** (3-Factorization)**.** *As one of our main runing examples in this book, we want to develop a snark that proofs knowledge of the factorization of elements of the finite field* $\mathbb{F}_{13}$ *into exactly three factors.*

*Formalizing such a high level description, we can define* $\Sigma := \mathbb{F}_{13}$ *as the alphabet of our language and then define a language* $L_{3.fac}$ *to consists of those tupels of field elements from* $\mathbb{F}_{13}$*, that contain exactly* $4$ *letters* $w_1, w_2, w_3, w_4$ *which satisfy the equation* $w_1 \cdot w_2 \cdot w_3 = w_4$*.*

*So for example the tuple* $(2,12,4,5)$ *is a word in* $L_{3.fac}$*, while neither* $(2,12,11)$*, nor* $(2,12,4,7)$ *nor* $(2,12,7,168)$ *are words in* $L_{3.fac}$ *as they dont satisfy the grammar or are not define over the proper alphabet.*

*We can describe the language* $L_{3.fac}$ *more formally by introducing a checking relation as described in XXX:*

$$R_{3.fac} : \mathbb{F}_{13}^* \to \{TRUE, FALSE\} \; ; \; (x_1, \ldots, x_n) \mapsto \begin{cases} TRUE & n = 4 \text{ and } x_1 \cdot x_2 \cdot x_3 = x_4 \\ FALSE & else \end{cases}$$

*Having defined the language* $L_{3.fac}$ *it then makes sense to claim the statement like "There is a word in* $L_{3.fac}$*" which is logically equivalent to say "There are four elements* $w_1, w_2, w_3, w_4$ *from the finite field* $\mathbb{F}_{13}$*" such that the equation* $w_1 \cdot w_2 \cdot w_3 = w_4$ *holds. Proofing the correctness of this statement could then be achieved by finding actual field elements like* $x_1 = 2$*,* $x_2 = 12$*,* $x_3 = 4$ *and* $x_4 = 5$ *giving the* $L_{3.fac}$ *instance* $(2,12,4,5)$*, which is a proof for our statement.*

**Instance and Witness** As we have seen statements can formulated as membership claims in formal languages and instances can serve as constructive proofs for those claims. However, in the context of *zero-knowledge* proofs its possible to hide parts of the proofing instance and still be able to proof the statement. In such a context the instance is therefore split into a *public part* which again is called the **instance** and a not publically known part (a private part) called the **wittness**.

To acknowledge for this seperation of a proof instance into a public and a private part, our previous definition of a formal language needs a refinement in the context of zero-knowledge proofs. Instead of a single alphabet we consider two alphabets $\Sigma_I$ and $\Sigma_W$, such that the words of the languages are tupels $(i|w) \in \Sigma_I^* \times \Sigma_W^*$ subject to a checking relation

$$R : \Sigma_I^* \times \Sigma_W^* \to \{TRUE, FALSE\} \tag{7.3}$$

that decides if a tupel $(i|w) \in \Sigma_I^* \times \Sigma_W^*$ is an element of the language or not. Again the language itself can be written as the set of all tupels that satisfies the grammar, i.e as:

$$L := \{(i,w) \in \Sigma_I^* \times \Sigma_W^* \in \Sigma^* \mid R(i|w) = TRUE\} \tag{7.4}$$

In this case we call a public input $i$ an **instance** and a private input $w$ a **wittness** of the relation $R$.

In this context a **statement** is the claim, that given an instance $i \in \Sigma_I^*$ there is a wittness $w \in \Sigma_W^*$, such that language $L$ contains a word $(i|w)$. As in the situation of more general languages one way to constructively *proof* such a statement is to provided an actual pair $(i|w)$ with $R(i|w)$, however the point of zero-knowledge proofing systems, as we will see in XXX, is to proof such a statement, without revealing any knowledge about $w$.

So while statements in the sense of the previous chapter can be seen as membership proofs, statements in this refined definition in combination with publically known instances are knowledge-proofs rather.

**Example 106** (3-factorization)**.** *Consider the language* $L_{3.fac}$ *from example XXX again. Providing instances for the membership statement in this languages is equivalent to providing knowledge of 4 elements* $x_1$*,* $x_2$*,* $x_3$ *and* $x_4$ *from* $\mathbb{F}_{13}$*, such that the modular* 13 *product of the first three elements is equal to the* 4*'th element.*

*Splitting instances into private and public parts, we can reformulate the problem introducing different levels of zero-knowledge into the problem. For example we could reformulate the non-hiding membership statement "there are* 4 *field elements* $x_1$*,* $x_2$*,* $x_3$ *and* $x_4$ *from* $\mathbb{F}_{13}$*, such that*

*$x_1 \cdot x_2 \cdot x_3 = x_4$ into a statement, where all factors $x_1$, $x_2$, $x_3$ of $x_4$ are private and only the product $x_4$ is public.*

*A statement would then be something like "There are three factors of $x_4$" and as we will see in XXX a zero-knowledge proofing system is able to proof this statement without revealing anything about the factors $x_1$, $x_2$, or $x_3$.*

*Formalizing this new language, which we might call $L_{3.fac\_zk}$ we define a checking relation by*

$$R_{3.fac\_zk} : \mathbb{F}_{13}^* \times \mathbb{F}_{13}^* \to \{TRUE, FALSE\} \;;$$
$$((i_1,\ldots,i_n)|(w_1,\ldots,w_m)) \mapsto \begin{cases} TRUE & n = 1,\, m = 3,\, i_1 = w_1 \cdot w_2 \cdot w_3 \\ FALSE & else \end{cases}$$

*and as usual $L_{3.fac\_zk}$ is then defined by all tupels from $\mathbb{F}_{13}^* \times \mathbb{F}_{13}^*$ that are mapped onto $TRUE$ under $R_{3.fac\_zk}$.*

*Since words in $L_{3.fac\_zk}$ are tuples $(i|w)$ consisting of an instance and a wittness, there are different possibilities to formulate statements. The most general one would be equivalent to the one in XXX claiming "there are words in $L_{3.fac\_zk}$" and a proof, could be given by a concrete pair $(i|w) \in L_{3.fac\_zk}$, such as$(5|2,12,4)$.*

*However as explained in XXX, in the context of zero knowlege proofs, statements are rather knowledge-claims like "Given public input i, there is a private input w. So for example in $L_{3.fac\_zk}$ with public input $i = 5$ a proof for the associated statement could be given by $w = (2,12,4)$.*

*As we will see in XXX, zero-knowledge proofing systems provide techniques to statements like this without revealing anything about the wittness.*

*One question that arises in this context might be why we decided the factors $x_1$, $x_2$ and $x_3$ to be the wittness and the product $x_4$ to be the instance. This of course was just an arbitrary choice and we could have decided on any other constellation. For example nothing stops us from declaring all variables as private or just $x_1$ or whatever. The actual choice is determined by the application only.*

**Example 107** (SHA256 – Knowlege of Preimage)**.** *A standard example to show the power of zero knowlege proofs is proving the knowledge of some preimage of a cryptographic hash function like the SHA256 function, without actually revealing it.*

*To understand this example in detail, lets start with introducing a language well suited to build a snark for this problem. Since SHA256 is a function*

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

*that maps binary string of arbitrary length onto binary strings of length 256 and we want to proof knowledge of preimages, we have to consider binary strings of size 256 as instances and binary strings of arbitrary length as witnesses.*

*An appropriate alphabet for both the set of all witnesses and the set of all instances is therefore the set $\{0,1\}$ of the two binary letters. We then define a checking relation by*

$$R_{SHA256} : \{0,1\}^* \times \{0,1\}^* \to \{TRUE, FALSE\} \;;$$
$$(i|w) \mapsto \begin{cases} TRUE & i.len() = 256,\, i = SHA256(w) \\ FALSE & else \end{cases}$$

*and we write $L_{SHA256}$ for the associated language that consists of instance, witness pairs $(i|w) \in L_{SHA256}$ where the instance i is the SHA256 image of the witness w.*

*Given an instance $i \in \{0,1\}^{256}$ a statements in $L_{SHA256}$ then is the claim, that there is a wittness $w \in \{0,1\}^*$, such that i is the image of SHA256 of w. One way to proof such a statement would therefore be to actually provide some data that hashes onto i.*

**Example 108** (Knowlege of Private Key)**.** *In this example we want to proof knowledge of an elliptic curve secrete key, provided a public key is given as instance. Of course the most common way to proof knowledge of this key, would be to sign something with it. However the purpose of this example is to shift view on the problem a bit.*

*As an example curve consider the tiny-jubjub curve from example XXX in its Edwards form .*

**Modularity and Gadgets** From a developers perspective it is often useful to construct complex statements and their representing languages from simple ones. In the context of zero knowledge proofs those simple building blocks are often called *gadgets* and mature proofing systems usually contain libraries that contain many useful gadgets, like representations of basic types as booleans, unit32, preimage proofs for Hash functions, elliptic curve cryptography and so on. Implementers can then combine these fundamental building blocks to write complex real world applications.

To understand what this means on the level of formal languages defined by checking relations as explained in XXX, we need to look at the *intersection* of two formal languages, which can be constructed whenever both languages are defined over the same alphabet. In this case the intersection language consists of words that are contained in both languages. To be more precise, let $L_1$ and $L_2$ be two formal languages defined over the instance and witness alphabets $\Sigma_I$ and $\Sigma_W$. Then

$$L_1 \cap L_2 := \{x \mid x \in L_1 \text{ and } x \in L_2\} \tag{7.5}$$

If both languages are defined by checking functions $R_1$ and $R_2$ as explained in XXX, the following function is a checking function for the intersection language $L_1 \cap L_2$:

$$R_{L_1 \cap L_2} : \Sigma_I^* \times \Sigma_W^* \to \{TRUE, FALSE\} \; ; \; (i,w) \mapsto R_1(i,w) \text{ and } R_2(i,w) \tag{7.6}$$

This is an important fact from an implementations point of view as it allows to construct complex checking relations and their statements from simple building blocks. Given a publically known instance $i \in \Sigma_I^*$ a statement in an intersection language then claims knowledge of a witness that satisfies all relations simultaniously.

## 7.2 Statement Representatuons

As we have seen in the previous section, formal languages and their definition by checking relations are a powerful tool to describe statements in a formaly regurous manner.

However from the perspective of existing zero knowledge proofing systems not all ways to actually represent checking relations are equally useful. Depending on the proofing system ad hand some are more suitable then others. In this section will therefore describe the most common ways to represent checking relation and their statements.

### 7.2.1 Circuit Satisfyability

**Definition 7.2.1.1** (Circuits)**.** *Let $\Sigma_I$ and $\Sigma_W$ be two alphabets. Then a directed, acyclic graph $C$ is called a **circuit** over $\Sigma_I \times \Sigma_W$, if the graph has an ordering and every node has a label in the following way:*

- *Every source node (called input) has a letter from $\Sigma_I \times \Sigma_W$ as label.*
- *Every sink node (called output) has a letter from $\Sigma_I \times \Sigma_W$ as label.*
- *Every other node (called gate) with $j$ incoming edges has a label that consist of a function $f : (\Sigma_I \times \Sigma_W)^j \to \Sigma_I \times \Sigma_W$.*

**Remark 1** (Circuit-SAT)**.** *Every circuit with $n$ input nodes and $m$ output nodes can be seen a function that transforms strings of size $n$ from $\Sigma_I \times \Sigma_W$ into strings of size $m$ over the same alphabet. The transformation is done by sending the strings from a node along the outgoing edges to other nodes. If those nodes are gates, then the string is transformed according to the label.*

*By executing the previous transformation, every node of a circuit has an associated letter from $\Sigma_I \times \Sigma_W$ and this defines a checking relation over $\Sigma_I^* \times \Sigma_W^*$. To be more precise, let $C$ be a circuit with $n$ nodes and $(i,w) \in \Sigma_I^j \times \Sigma_W^k$ a string. Then $R_C(i,w)$ iff THE CIRCUIT IS SATISFIED WHEN ALL LABELS ARE ASSOCIATED TO ALL NODES IN THE CIRCUIT.... BUT MORE PRECISE*

*MODULO ERRORS. TO BE CONTINUED.....*

*An Assignment associates field elements to all edges (indices) in an algebraic circuit. An Assignment is valid, if the field element arise from executing the circuit. Every other assignment is invalid.*

*The checking relation for circuit-SAT then is satidfied if valid asignment (TODO: THE WITNESS/INSTANCE SPLITTING)*

*Valid assignments are proofs for proper circuit execution.*

So to summarize, algebraic circuits (over a field $\mathbb{F}$) are directed acyclic graphs, that express arbitrary, but bounded computation. Vertices with only outgoing edges (leafs, sources) represent inputs to the computation, vertices with only ingoing edges (roots, sinks) represent outputs from the computation and internal vertices represent field operations (Either addition or multiplication). It should be noted however that there are many circuits that can represent the same laguage...

Circuits have a notion of execution, where input values are send from leafs along edges, through internal vertices to roots.

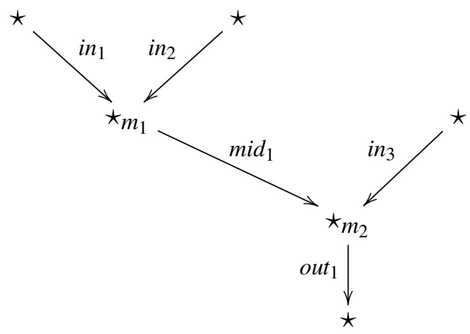
**Remark 2.** *Algebraic circuits are usually derived by Compilers, that transform higher languages to circuits. An example of such a compiler is XXX. Note: Different Compiler give very different circuit representations and Compiler optimization is important.*

**Example 109** (Generalized factorization snark)**.** *Consider our generalized factorization example 115 with associated language* **??**.

*To write this example in circuit-SAT, consider the following function*

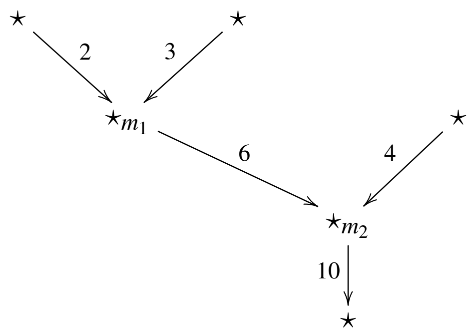
$$f : \mathbb{F}_{13} \times \mathbb{F}_{13} \times \mathbb{F}_{13} \to \mathbb{F}_{13}; (x_1, x_2, x_3) \mapsto (x_1 \cdot x_2) \cdot x_3$$

*A valid circuit for $f : \mathbb{F}_{11} \times \mathbb{F}_{11} \times \mathbb{F}_{11} \to \mathbb{F}_{11}; (x_1, x_2, x_3) \mapsto (x_1 \cdot x_2) \cdot x_3$ is given by:*



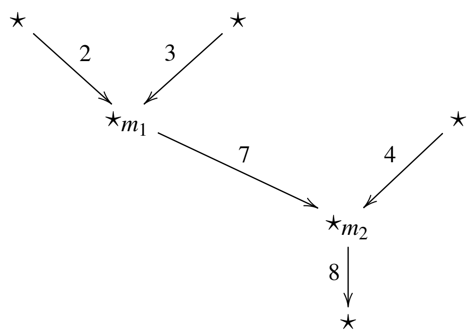

*with edge-index set* $I := \{in_1, in_2, in_3, mid_1, out_1\}$.

*To given a valid assignment, consider the set* $I_{valid} := \{in_1, in_2, in_3, mid_1, out_1\} = \{2, 3, 4, 6, 10\}$




*Appears from multiplying the input values at* $m_1$*,* $m_2$ *in* $\mathbb{F}_{13}$*, hence by executing the circuit.*

*Non valid assignment:* $I_{err} := \{in_1, in_2, in_3, mid_1, out_1\} = \{2, 3, 4, 7, 8\}$




*Can not appear from multiplying the input values at* $m_1$*,* $m_2$ *in* $\mathbb{F}_{13}$

*To match the requirements of the inital task 115, we have to split the statement into instance and witness. So given index set* $I := \{in_1, in_2, in_3, mid_1, out_1\}$*, we assume that every step in the computation other then* $in_3$ *and* $out_1$ *are part of the witness. So we choose:*

- *Instance* $S = \{in_3, out_1\}$.
- *Witness* $W = \{in_1, in_2, mid_1\}$.

**Example 110** (Baby JubJub for BLS6-6)**.**

**Example 111** (ECDH as a circuit)**.** *over BLS6*

**Example 112** (BLS Signature)**.** *example of one layer recursion over MNT4 and MNT6*

**Example 113** (Boolean Circuits)**.**

**Example 114** (Algebraic (Aithmetic) Circuits)**.**

Any program can be reduced to an arithmetic circuit (a circuit that contains only addition and multiplication gates). A particular reduction can be found for example in [BSCG+13]

## 7.3 Computational Models

Proofs are the evidence of correctness of the assertions, and people can verify the cor-rectness by reading the proof. However, we obtain much more than the correctness itself:After you read one proof of an assertion, you know not only the correctness, but also why itis correct. Is it possible to solely show the correctness of an assertion without revealing theknowledge of proofs? It turns out that it is indeed possible, and this is the topic of today'slecture: Zero Knowledge Systems.

**Example 115** (Generalized factorization snark)**.** *As one of our major running examples we want to derive a zk-SNARK for the following generalized factorization problem:*

*Given two numbers $a, b \in \mathbb{F}_{13}$, find two additional numbers $x, y \in \mathbb{F}_{13}$, such that*

$$(x \cdot y) \cdot a = b$$

*and proof knowledge of those numbers, without actually revealing them.*

*Of course this example reduces to the classic factorization problem (over $\mathbb{F}_{13}$ by setting $y = 1$)*

*This zero knowledge system deals with the following situation: "Given two publicly known numbers $a, b \in \mathbb{F}_{13}$ a proofer can show that they know two additional numbers $x, y \in \mathbb{F}_{13}$, such that $(x \cdot y) \cdot a = b$, without actually revealing x or y."*

*Of course our choice of what information to hide and what to reveal was completely arbitrary. Every other split would also be possible, but eventually gives a different problem.*

*For example the task could be to not hide any of the variables. Such a system has no zero knowledge and deals with verifiable computations: "A worker can proof that they multiplied three publicly known numbers $a, b, x \in \mathbb{F}_{13}$ and that the result is $z \in \mathbb{F}_{13}$, in such a way that no verifier has to repeat the computation."*

### 7.3.1 Rank-1 Constraint Systems

The idea of R1CS is that it keeps track of the values that each variable assumes during the computation, and binds the relationships among all those variables that are implied by the computation itself. This way doing a computation properly is enforced by enforcing relations (constraints) between everyconsecutive steps in the computation.

**Definition 7.3.1.1** (Rank-1 Constraint system)**.** *Let $\mathbb{F}$ be a Galois field, $i, j, k$ three numbers and A, B and C three $(i + j + 1) \times k$ matrices with coefficients in $\mathbb{F}$. Then any vector $x = (1, \phi, w) \in \mathbb{F}^{1+i+j}$ that satisfies the* ***rank-1 constraint system*** *(R1CS)*

$$Ax \odot Bx = Cx$$

*(where $\odot$ is the Hadamard/Schur product) is called a* ***statement*** *of that system, with* ***instance*** *$\phi$ and* ***witness*** *$w$.*

*We call $k$ the* ***number of constraints****, $i$ the* ***instance*** *size and $j$ the* ***witness*** *size.*

**Remark 3.** *Any Rank-1 constraint system defines a formal language in the following way: Consider the alphabets $\Sigma_I := \mathbb{F}$ and $\Sigma_W : \mathbb{F}$. Then a checking relation $R_{R1CS} \subset \Sigma_I^i \times \Sigma_W^j \subset \Sigma_I^* \times \Sigma_W^*$ is defined by*

$$R_{R1CS}(i,w) \Leftrightarrow (i,w) \text{ satisfies the R1CS}$$

*As shown in XXX such a checking relation defines a formal language. We call this language* ***R1CS satisfiability****.*

**Example 116** (Generalized factorization snark)**.** *Defining the 5-dimensional affine vector $w = (1, in_1, in_2, in_3, m_1, out_1)$ for $in_1, in_2, in_3, m_1, out_1 \in \mathbb{F}_{13}$ and the $6\times?$-matrices*

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

*We can instantiate the general R1CS equation $Aw \odot Bw = Cw$ as*

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix} \odot \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ in_1 \\ in_2 \\ in_3 \\ m_1 \\ out_1 \end{pmatrix}$$

*So evaluating all three matrix products and the Hadarmat prodoct we get two constraint equations*

$$\begin{aligned} in_1 \cdot in_2 &= m_1 \\ m_1 \cdot in_3 &= out_1 \end{aligned}$$

So from the way this R1CS is constructed, we know that whatever the underlying field $\mathbb{F}$ is, the only solutions to this equations are

$$\{(0,0,0), (0,1,0), (1,0,0), (1,1,1)\}$$

**Gadgets** Rank 1 contraints systems can become very large ....

### Boolean Algebra

Sometimes it is necessary to assume that a statement describes boolean variables. However by definition the alphabet of a statement is a finite field, which is often the scalar field of a large prime order cyclic group. So developers need a way to simulate boolean algebra inside other finite fields.

The most common way to do this, is to interpret the additive and multiplicate neutral element $\{0,1\} \subset F$ as boolean values. This is convinient because they are defined in any field.

In what follows we will define a few of the most basic R1CS to check boolean expressions in R1CS satidfyability. We will leave other basic constructions as exercises to the reader.

We start with actually constraining field elements to boolean values then Once field elements are boolean constraint, we need constraints that are able to enforce boolean algebra on them. We therefore give constraints for the functionally complete set of Boolean operators give by $AND$ and $NOT$. As all other boolean operations can be constructed from $AND$ and $NOT$, this sufficies. However in actual implementations it is of high importance to limit the number of constraints as much as possible. In reality it is therefor advantageous to implement all logic operators in constraints.

**Boolean Constraint** So when a developer needs boolean variables as part of their statement, a R1CS is required on those variables, that enforces the variable to be either 1 or 0. So to "constrain a field element $x \in \mathbb{F}$ to be 1 or 0 what we need is a system of equation $(A_i x)\cdot(B_i x) = C_i x$ for some $A_i, B_i, C_i \in \mathbb{F}$, such that the only possible solutions for $x$ are 0 or 1. As it turns out such a system can be realized by a single equation $x \cdot (1-x) = 0$ We see that indeed 0 and 1 are the only solutions here, since for the right side to be zero, at least one factor on the left side needs to be zero and this only happens for 0 and 1.

So now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}1\\x\end{pmatrix}\odot\begin{pmatrix}1 & -1\end{pmatrix}\begin{pmatrix}1\\x\end{pmatrix}=\begin{pmatrix}0 & 0\end{pmatrix}\begin{pmatrix}1\\x\end{pmatrix}$$

So we get the following statement $\phi = (1,i,w) = (1,x)$, with instance (public input) $i = x$ and now witness (private input) $w$. In addition we get the matrices $A = \begin{pmatrix}0 & 1\end{pmatrix}$, $B = \begin{pmatrix}1 & -1\end{pmatrix}$ and $C = \begin{pmatrix}0 & 0\end{pmatrix}$.

To make those constraints easily accesable for R1CS developers, a gadget is convinient:

**AND-constraints** Given three field elements $x,y,z \in \mathbb{F}$ that represent boolean variables, we want to find a R1CS, such that $w = (1,x,y,z)$ satisfies the constraint system if and only if $x\,AND\,y = z$.

So first we have to constrain $x$, $y$ and $z$ to be boolean as explained in XXX. The next thin is we need to find a R1CS that enforces the *AND* logic. We can simply choose $x \cdot y = z$, since (for boolean constraint values) $x \cdot y$ equals 1 if and only if both $x$ and $y$ are 1.

Now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}\odot\begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}=\begin{pmatrix}0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}$$

Combining this R1CS with the required fthree boolean constraints for $x$, $y$ and $z$ we get

$$\begin{pmatrix}0 & 1 & 0 & 0\\ \hline 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}\odot\begin{pmatrix}0 & 0 & 1 & 0\\ \hline 1 & -1 & 0 & 0\\ 1 & 0 & -1 & 0\\ 1 & 0 & 0 & -1\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}=\begin{pmatrix}0 & 0 & 0 & 1\\ \hline 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\\z\end{pmatrix}$$

So from the way this R1CS is constructed, we know that whatever the underlying field $\mathbb{F}$ is, the only solutions to this equations are

$$\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}$$

which is the set of all $(x,y,z) \in \{0,1\}^3$ such that $x\,AND\,y = z$.

**NOT constraint** Given two field elements $x,y \in \mathbb{F}$ that represent boolean variables, we want to find a R1CS, such that $w = (1,x,y)$ satisfies the constraint system if and only if $x = \neg y$.

So again we have to constrain $x$ and $y$ to be boolean as explained in XXX. The next think is we need to find a R1CS that enforces the *NOT* logic. We can simply choose $(1-x) = y$, since (for boolean constraint values) this enforces that $y$ is always the boolean opposite of $x$.

Now that we have found a correct equation for a boolean constrain, we have to translate it into the associated R1CS format, which is given by

$$\begin{pmatrix}1 & -1 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}\odot\begin{pmatrix}1 & 0 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}=\begin{pmatrix}0 & 0 & 1\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}$$

So actually we wrote the linear equation $1-x=y$ like $(1-x)\cdot 1=y$ and translated that into the matrix equation.

Combining this R1CS with the required fthree boolean constraints for $x$, $y$ and $z$ we get

$$\begin{pmatrix}1 & -1 & 0\\ \hline 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}\odot\begin{pmatrix}1 & 0 & 0\\ \hline 1 & -1 & 0\\ 1 & 0 & -1\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}=\begin{pmatrix}0 & 0 & 1\\ \hline 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\begin{pmatrix}1\\x\\y\end{pmatrix}$$

So from the way this R1CS is constructed, we know that whatever the underlying field $\mathbb{F}$ is, the only solutions to this equations are

$$\{(0,1),(1,0)\}$$

which is the set of all $(x,y)\in\{0,1\}^2$ such that $x=\neg y$.

EXERCISE: DO OR; XOR; NAND

More complicated logical constraints can then be optained by combining all sub-R1CS together. For example if the task is to enforce $(in_1\ AND \neg in_2) AND in_3 = out_1$ we first apply the FLATTENING technique from XXX, which gives is

$$\begin{array}{lcl}\neg in_2 & = & mid_1\\ in_1\ AND\ mid_1 & = & mid_2\\ mid_2\ AND\ in_3 & = & out_1\end{array}$$

So we have the statement $w=(1,in_1,in_2,in_3,mid_1,mid_2,out_1)$, 6 boolean constraints for the variables, 2 constraints for the 2 *AND* operations and 1 constraint for the *NOT* operation.

### Binary representations

In circuit computations its is often necessary to use the binary representation of a prime field element. Binary representations of prime field elements work execactly like binary representations of ordinary unsigned integers. Only the algebraic operations are different. To compute the binary representation of some number $x\in\mathbb{F}_p$ we need to know the number of bits in the binary representation of $p$ first. We write this as $m=|p_{bin}|$.

Then a bitstring $(b_0,\ldots,b_m)\in\{0,1\}^m$ is the binary representation of the field element $x$, if and only if

$$x=b_0\cdot 2^0+b_1\cdot 2^1+\ldots+b_m\cdot 2^m$$

Note that, since $p$ is a prime number that has a leading bit 1 at position $m$. Moreover every prime number $p>2$ is odd and hence has least significat bit set to 1. Hence all numbers $2^j$ for $0\le j\le m$ are elements of $\mathbb{F}_p$ and the equation is well defined. We can therefore enforce this equation as a R1CS, by flattening the equation:

$$\begin{array}{lcl}b_0\cdot 1 & = & mid_0\\ b_1\cdot 2 & = & mid_1\\ \cdots & = & \cdots\\ b_m\cdot 2^m & = & mid_m\\ (mid_0+mid_1+\ldots+mid_m)\cdot 1 & = & x\end{array}$$

So we have the statement $w = (1, x, b_0, \ldots, b_m, mid_0, \ldots, mid_m)$ and we need $(m+1)$ constraints to enforce the binary representation in addition to the $m$ constraints that enforce booleanness.

At this point we see, that writing more complex R1CS becomes clumbsy and in actual implementations people therefore use languages to makes the constraint system more readable. In this example we could write for example something like this:

keeping in mind that this is a meta level algorith to **generate** the R1CS, not the R1CS itself, as constructs like for loops have not direct meening on the level of the R1CS itself.

**Example 117.** *Considering the prime field $\mathbb{F}_{13}$, we want to enforce the binary representation of $7 \in \mathbb{F}_{13}$. To find the number of bits that we need to consider in our R1Cs, we start with the binary representation of 13, which is $(1,0,1,1)$ since $13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$. So $m = 4$ and we have to enforce a 4-bit representation for 7, which is $(1,1,1,0)$, since $7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$.*

*A valid statement is then given by $w = (1,7,1,1,1,0,1,2,4,0)$ and indeed we satify the 9 required constraints*

$$\begin{array}{lcll} 1 \cdot (1-1) & = & 0 & \textit{// boolean constraints} \\ 1 \cdot (1-1) & = & 0 & \\ 1 \cdot (1-1) & = & 0 & \\ 0 \cdot (1-0) & = & 0 & \\ & & & \\ 1 \cdot 1 & = & 1 & \\ 1 \cdot 2 & = & 2 & \\ 1 \cdot 4 & = & 4 & \\ 0 \cdot 8 & = & 0 & \\ (1+2+4+0) \cdot 1 & = & 7 & \end{array}$$

### Conditional (ternary) operator

It is often required to implement the terniary conditional operator ? : as a R1CS. In general this operator takes three arguments, a boolean value $b$ and two expressions $if\_true$ and $if\_false$, usually written as $b\ ?\ c\ :\ d$ and executes $c$ and $d$ according to the value of $b$.

If we assume all three arguments to be values from a finite field, such that $b$ is boolean constraint (XXX), we can enforce a field element $x$ to be the result of the conditional operator as

$$x = b \cdot c + (1-b) \cdot d$$

Flattening the code gives

$$\begin{array}{lcl} b \cdot c & = & mid_0 \\ (1-b) \cdot d & = & mid_1 \\ (mid_0 + mid_1) \cdot 1 & = & x \end{array}$$

So we have the statement $w = (1, x, b, c, d, mid_0, mid_1)$ and we need 3 constraints to enforce the conditional operator in addition to 1 constraint that enforces booleanness of $b$.

NOTE: THERE WAS THIS PODCAST WITH ANNA AND THE GUY JAN TALKE TO WHERE HE SAID; CONDITIONALS CAN BE IMPLEMENTED SUCH THAT NOT BOTH BRANCHES ARE EXCUTED: LOOK THAT UP

### Range Proofs

$x > 5$...

### UintN

STUFF ABOUT HOW UINTN COMPUTATIONS ARE NOT STANDARDIZED AND THAT THERE ARE IMPLEMENTATIONS OTHER THEN MOD-N.... WE FIX ON MOD-N. WHAT DO ZEXE CIRCOM ECT FIX ON?

As we know circuits are not defined over integers but over finite fields instead. We therefore have no notation of integers in circuits. However on computers we also not use integers natively but Uint's instead.

As we know a UintN type is a representation of integers in the range of $0\ldots2^N$ with the exception that algebraic operations like addition and multiplication deviate from actual integers, whenever the result exceeds the largest representable number $2^N-1$.

In circuit design it is therefore important to distinguish between various things tht might look like integers, but are actually not. For example Haskells type NAT is an actual implementation of natural numbers. In particular this means ....

**Example 118** (Uint8)**.** *What is $0xFFF0+0xFFF0$ and so on...*

**Bit constraints** In prime fields, addition and multiplication behaves exactly like addition and multiplication with integers as long as the result does not exceed the modulus.

This makes the representation of UintNs in a prime field $\mathbb{F}_p$ potentially ambigious, as there are two possible representations, whenever $2^N-1<p$. In that case any element of $UintN$ could be interpreted as an element of $\mathbb{F}_p$. This however is dangerous as the algebraic laws like addition and multiplication behave very different in general.

It is therefore common to represent UintN types in circuits as binary constraints strings of field elements of length $N$.

**Example 119.** *Consider the Uint4 type over the prime field $\mathbb{F}_{17}$. Since $2^4=16$, Uint4 can represent the numbers $0,\ldots,15$ and it would be possible to interpret them as elements in $\mathbb{F}_{17}$. However addition*

### Twisted Edwards curves

Sometimes it required to do elliptic curve cryptography "inside of a circuit". This means that we have to implement the algebraic operations (addition, scalar multiplication) of an elliptic curve as a R1CS. To do this efficiently the curve that we want to implement must be defined over the same base field as the field that is used in the R1CS.

**Example 120.** *So for example when we consider an R1CS over the field $\mathbb{F}_{13}$ as we did in example XXX, then we need a curve that is also defined over $\mathbb{F}_{13}$. Moreover it is advantegous to use a (twisted) Edwards curve inside a circuit, as the addition law contains no branching (See XXX). As we have seen in XXX our Baby-Jubjub curve is an Edwards curve defined over $\mathbb{F}_{13}$. So it is well suited for elliptic curve cryptography in our pend and paper examples*

**Twisted Edwards curves constraints** As we have seen in XXX, an Edwards curve over a finite field $F$ is the set of all pairs of points $(x,y)\in\mathbb{F}\times\mathbb{F}$, such that $x$ and $y$ satisfy the equation $a\cdot x^2+y^2=1+d\cdot x^2y^2$.

We can interpret this equation as a constraint on $x$ and $y$ and rewrite it as a R1CS by applying the flattenin technique from XXX.

$$\begin{array}{lcr} x\cdot x & = & x\_sq \\ y\cdot y & = & y\_sq \\ x\_sq\cdot y\_sq & = & xy\_sq \\ (a\cdot x\_sq + y\_sq)\cdot 1 & = & 1+d\cdot xy\_sq \end{array}$$

So we have the statement $w=(1,x,y,x\_sq,y\_sq,xy\_sq)$ and we need 4 constraints to enforce that $x$ and $y$ are points on the Edwards curve $x^2+y^2=1+d\cdot x^2y^2$. Writing the constraint system in matrix form, we get:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & a & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix} \odot \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & d \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ x\_sq \\ y\_sq \\ xy\_sq \end{pmatrix}$$

EXERCISE: WRITE THE R1CS FOR WEIERSTRASS CURVE POINTS

**Example 121** (Baby-JubJub)**.** *Considering our pen and paper Baby JubJub curve over from XXX, we know that the curve is defined over $\mathbb{F}_{13}$ and that $(11,9)$ is a curve point, while $(2,3)$ is not a curve point.*

*Starting with $(11,9)$, we can compute the statement $w=(1,11,9,4,3,12)$. Substituting this into the constraints we get*

$$\begin{array}{lcr} 11\cdot 11 & = & 4 \\ 9\cdot 9 & = & 3 \\ 4\cdot 3 & = & 12 \\ (1\cdot 4+3)\cdot 1 & = & 1+7\cdot 12 \end{array}$$

*which is true in $\mathbb{F}_{13}$. So our statement is indeed a valid assignment to the twisted Edwards curve constraining system.*

*Now considering the non valid point $(2,3)$, we can still come up with some kind of statement $w$ that will satisfy some of the constraints. But fixing $x=2$ and $y=3$, we can never satisfy all constraints. For example $w=(1,2,3,4,9,10)$ will satisfy the first three constraints, but the last constrain can not be satisfied. Or $w=(1,2,3,4,3,12)$ will satisfy the first and the last constrain, but not the others.*

**Twisted Edwards curves addition** As we have seen in XXX one the major advantages of working with (twisted) Edwards curves is the existence of an addition law, that contains no branching and is valid for all curve points. Moreover the neutral element is not "at infinity" but the actual curve poin $(0,1)$.

As we know from XXX, give two points $(x_1,y_1)$ and $(x_2,y_2)$ on a twisted Edwards curve their sum is given by

$$(x_3,y_3)=\left(\frac{x_1y_2+y_1x_2}{1+d\cdot x_1x_2y_1y_2}, \frac{y_1y_2-a\cdot x_1x_2}{1-d\cdot x_1x_2y_1y_2}\right)$$

We can realize this equation as a R1CS as follows: First not that we can rewrite the addition law as

$$\begin{array}{lcl} x_1 \cdot x_2 & = & x_{12} \\ y_1 \cdot y_2 & = & y_{12} \\ x_1 \cdot y_2 & = & xy_{12} \\ y_1 \cdot x_2 & = & yx_{12} \\ x_{12} \cdot y_{12} & = & xy_{1212} \\ x_3 \cdot (1 + d \cdot xy_{1212}) & = & xy_{12} + yx_{12} \\ y_3 \cdot (1 - d \cdot xy_{1212}) & = & y_{12} - a \cdot x_{12} \end{array}$$

So we have the statement $w = (1, x_1, y_1, x_2, y_2, x_3, y_3, x_{12}, y_{12}, xy_{12}, yx_{12}, xy_{1212})$ and we need 7 constraints to enforce that $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

**Example 122** (Baby-JubJub)**.** *Considering our pen and paper Baby JubJub curve over from XXX. We recall from XXX that* $(11,9)$ *is a generator for the large prime order subgroup. We therefor already know from XXX that* $(11,9)+(7,8) = (11,9)+[3](11,9) = [4](11,9) = (2,9)$. *So we compute a valid statement as* $w = (1,11,9,7,8,2,9,12,7,10,11,6)$. *Indeed*

$$\begin{array}{lcl} 11 \cdot 7 & = & 12 \\ 9 \cdot 8 & = & 7 \\ 11 \cdot 8 & = & 10 \\ 9 \cdot 7 & = & 11 \\ 10 \cdot 11 & = & 6 \\ 2 \cdot (1 + 7 \cdot 6) & = & 10 + 11 \\ 9 \cdot (1 - 7 \cdot 6) & = & 7 - 1 \cdot 12 \end{array}$$

There are optimizations for this using only 6 constraints, available:

**Twisted Edwards curves inversion** Similar to elliptic curves in Weierstrass form, inversion is cheap on Edwards curve as the negative of a curve point $-(x,y)$ is given by $(-x,y)$. So a curve point $(x_2,y_2)$ is the additive inverse of another curve point $(x_1,y_1)$ precisely if the equation $(x_1,y_1) = (-x_2,y_2)$ holds. We can write this as

$$\begin{array}{lcl} x_1 \cdot 1 & = & -x_2 \\ y_1 \cdot 1 & = & y_2 \end{array}$$

We therefor have a statement of the form $w = (1, x_1, y_1, x_2, y_2)$ and can write the constraints into a matrix equation as

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{pmatrix}$$

In addition we need the following constraints:

$$\begin{array}{lcl} x_1 \cdot 1 & = & -x_2 \\ y_1 \cdot 1 & = & y_2 \end{array}$$

**Twisted Edwards curves scalar multiplication** Although there are highly optimzed R1CS implementations for scal multiplication on elliptic curves, the basic idea is somewhat simple: Given an elliptic curve $E/\mathbb{F}_r$, a scalar $x \in \mathbb{F}_r$ with binary representation $(b_0, \ldots, b_m)$ and a curve point $P \in E/\mathbb{F}_r$, the scalar multiplication $[x]P$ can be written as

$$[x]P = [b_0]P + [b_1]([2]P) + [b_2]([4]P) + \ldots + [b_m]([2^m]P)$$

and since $b_j$ is either 0 or 1, $[b_j](kP)$ is either the neutral element of the curve or $[2^j]P$. However $[2^j]P$ can be computed inductively by curve point doubling, since $[2^j]P = [2]([2^{j-1}]P)$.

So scalar multiplication can be reduced to a loop of length $m$, where the original curve point is repeadedly douled and added to the result, whenever the appropriate bit in the scalar is equal to one.

So to enforce that a curve point $(x_2, y_2)$ is the scalar product $[k](x_1, y_1)$ of a scalar $x \in F_r$ and a curve point $(x_1, y_1)$, we need an R1CS the defines point doubling on the curve (XXX) and an R1CS that enforces the binary representation of $x$ (XXX).

In case of twisted Edwards curve, we can use ordinary addition for doubling, as the constraints works for both cases (doublin is addition, where both arguments are equal). Moreover $[b](x,y) = (b \cdot x, b \cdot y)$ for boolean $b$. Hence flattening equation XXX gives

$$\begin{aligned} b_0 \cdot x_1 &= x_{0,1} \quad //[b_0]P \\ b_0 \cdot y_1 &= y_{0,1} \end{aligned}$$

In addition we need to constrain $(b_0, \ldots, b_N)$ to be the binary representation of $x$ and we need to constrain each $b_j$ to be boolean.

As we can see a R1CS for scalar multiplication utilizes many R1CS that we have introduced before. For efficiency and readability it is therefore useful to apply the concept of a gadget (XXX). A pseudocode method to derive the associated R1CS could look like this:

**Curve Cycles** A particulary interesting case with far reaching implication is the situation when we have two curve $E_1$ and $E_2$, such that the scalar field of curve $E_1$ is the base field of curve $E_2$ and vice versa. In that case it is possible to implement the group laws of one curve in circuits defined over the scalar field of the other curve.

#### The RAM Model

FROM THE PODCAST WITH ANNA R. AND THE GUY FROM JAN....

#### Generalizations

many circuits can be found here:

### 7.3.2 Quadratic Arithmetic Programs

As shown by [Pinocchio] rank-1 constraint systems can be transformed into so called quadratic arithmetic programs assuming $\mathbb{F}$.

taken from the pinocchio paper. For proving arithmetic circuit-sat. Given a R1CS QAPs transform potential solution vectors into two polynomials $p$ and $t$, such that $p$ is divisible by $t$ if and only if the vector is a solution to the R1CS.

They are major building blocks for **succinct** proofs, since with high probability, the divisibility check can be performed in a single point of those polynomials. So computationally expensive polynomial division check is reduced TO WHAT? (IN FIELDS THERE IS ALWAYS DIVISIBILITY)

**Definition 7.3.2.1** (Quadratic Arithmetic Program)**.** *Assume we have a Galois field $\mathbb{F}$, three numbers $i, j, k$ as well as three $(i+j+1)\times k$ matrices $A$, $B$ and $C$ with coefficients in $\mathbb{F}$ that define the R1CS $Ax\odot Bx = Cx$ for some statement $x=(1,i,w)$ and let $m_1,\ldots,m_k \in \mathbb{F}$ be arbitrary field elements.*

*Then a **quadratic arithmetic program** of the R1CS is the following set of polynomials over $\mathbb{F}$*

$$QAP = \left\{ t \in \mathbb{F}[x], \{a_h, b_h, c_h \in \mathbb{F}[x]\}_{h=1}^{i+j+1} \right\}$$

*where $t(x) := \Pi_{l=1}^{k}(x-m_l)$ is a polynomial f degree $k$, called the **target polynomial** of the QAP and $a_h(x)$, $b_h(x)$ as well as $c_h(x)$ are the unique degree $k-1$ polynomials that are defined by the equations*

$$a_h(m_l) = A_{h,l} \quad b_h(m_l) = B_{h,l} \quad c_h(m_l) = C_{h,l} \quad h = 1,\ldots,i+j+1, l = 1,\ldots,k$$

The major point is that R1CS-sat can be reformulated into the divisibility of a polynomials defined by any QAP.

**Theorem 7.3.2.2.** *Assume that an R1CS and an associated QAP as defined in XXX are given. Then the affine vector $y = (1,i,w)$ is a solution to the R1CS, if and only if the polynomial*

$$p(x) = \left(\sum y_h \cdot a_h(x)\right)\cdot\left(\sum y_h \cdot b_h(x)\right) - \sum y_h \cdot c_h(x)$$

*is divisible by the target polynomial $t$.*

The polynomials $a_h$, $b_h$ and $c_h$ are uniquely defined by the equations in XXX. However to actually compute them we need some algorithm like the Langrange XXX from XXX.

**Example 123** (Generalized factorization snark)**.** *In this example we want to transform the R1CS from example **??** into an associated QAP.*

*We start by choosing an arbitrary field element for every constraint in the R1CS, since we have 2 constraints we choose $m_1 = 5$ and $m_2 = 7$*

*With this choice we get the target polynomial $t(x) = (x-m_1)(x-m_2) = (x-5)(x-7) = (x+8)(x+6) = x^2+x+9$.*

*Since our statement has structure $w = (1, in_1, in_2, in_3, m_1, out_1)$ we have to compute the following degree 1 polynomials*

$\{a_c, a_{in_1}, a_{in_2}, a_{in_3}, a_{mid_1}, a_{out}\}$ $\{b_c, b_{in_1}, b_{in_2}, b_{in_3}, b_{mid_1}, b_{out}\}$ $\{c_c, c_{in_1}, c_{in_2}, c_{in_3}, c_{mid_1}, c_{out}\}$

*Apply QAP rule XXX to the $a_{k\in I}$ polynomials gives*

$$\begin{array}{llllll}
a_c(5) = 0, & a_{in_1}(5) = 1, & a_{in_2}(5) = 0, & a_{in_3}(5) = 0, & a_{mid_1}(5) = 0, & a_{out}(5) = 0 \\
a_c(7) = 0, & a_{in_1}(7) = 0, & a_{in_2}(7) = 0, & a_{in_3}(7) = 0, & a_{mid_1}(7) = 1, & a_{out}(7) = 0 \\
\\
b_c(5) = 0, & b_{in_1}(5) = 0, & b_{in_2}(5) = 1, & b_{in_3}(5) = 0, & b_{mid_1}(5) = 0, & b_{out}(5) = 0 \\
b_c(7) = 0, & b_{in_1}(7) = 0, & b_{in_2}(7) = 0, & b_{in_3}(7) = 1, & b_{mid_1}(7) = 0, & b_{out}(7) = 0 \\
\\
c_c(5) = 0, & c_{in_1}(5) = 0, & c_{in_2}(5) = 0, & c_{in_3}(5) = 0, & c_{mid_1}(5) = 1, & c_{out}(5) = 0 \\
c_c(7) = 0, & c_{in_1}(7) = 0, & c_{in_2}(7) = 0, & c_{in_3}(7) = 0, & c_{mid_1}(7) = 0, & c_{out}(7) = 1
\end{array}$$

*Since our polynomials are of degree* 1 *only we don't have to invoke Langrange method but can deduce the solutions right away.*

*Polynomials are defined on the two values* 5 *and* 7 *here. Linear Polynomial* $f(x) = m \cdot x + b$ *is fully determined by this. Derive the general equation:*

- $5m + b = f(5)$ *and* $7m + b = f(7)$
- $b = f(5) - 5m$ *and* $b = f(7) - 7m$
- $b = f(5) + 8m$ *and* $b = f(7) + 6m$
- $f(5) + 8m = f(7) + 6m$
- $8m - 6m = f(7) - f(5)$
- $2m = f(7) + 12f(5)$
- $7 \cdot 2m = 7(f(7) + 12f(5))$
- $m = 7(f(7) + 12f(5))$
- 
- $b = f(5) + 8m$
- $b = f(5) + 8 \cdot (7(f(7) + 12f(5)))$
- $b = f(5) + 4(f(7) + 12f(5))$
- $b = f(5) + 4f(7) + 9f(5)$
- $b = 10f(5) + 4f(7)$

*Gives the general equation:* $f(x) = 7(f(7) + 12f(5))x + 10f(5) + 4f(7)$

*For* $a_{in_1}$ *the computation looks like this:*

- $a_{in_1}(x) = 7(a_{in_1}(7) + 12a_{in_1}(5))x + 10a_{in_1}(5) + 4a_{in_1}(7) =$
- $7(0 + 12 \cdot 1)x + 10 \cdot 1 + 4 \cdot 0 =$
- $7 \cdot 12x + 10 =$
- $6x + 10$
- $a_{mid_1}(x) = 7(a_{mid_1}(7) + 12a_{mid_1}(5))x + 10a_{mid_1}(5) + 4a_{mid_1}(7) =$
- $7(1 + 12 \cdot 0)x + 10 \cdot 0 + 4 \cdot 1 =$
- $7 \cdot 1x + 4 =$
- $7x + 4$

| | | |
|---|---|---|
| $a_c(x) = 0$ | $b_c(x) = 0$ | $c_c(x) = 0$ |
| $a_{in_1}(x) = 6x+10$ | $b_{in_1}(x) = 0$ | $c_{in_1}(x) = 0$ |
| $a_{in_2}(x) = 0$ | $b_{in_2}(x) = 6x+10$ | $c_{in_2}(x) = 0$ |
| $a_{in_3}(x) = 0$ | $b_{in_3}(x) = 7x+4$ | $c_{in_3}(x) = 0$ |
| $a_{mid_1}(x) = 7x+4$ | $b_{mid_1}(x) = 0$ | $c_{mid_1}(x) = 6x+10$ |
| $a_{out}(x) = 0$ | $b_{out}(x) = 0$ | $c_{out}(x) = 7x+4$ |

*This gives the quadratic arithmetic program for our generalized factorization snark as*

$QAP = \{x^2+x+9, \{0,6x+10,0,0,7x+4,0\}, \{0,0,6x+10,7x+4,0,0\}, \{0,0,0,0,6x+10,7x+4\}\}$

*Now as we recall, the main point for using QAPs in snarks is the fact, that solutions to R1CS are in 1:1 correspondence to the divisibility of a polynomial p, constructed from a R1CS solution and the polynomials of the QAP and the target polynomial.*

*So lets see this in our example. We already know from example XXX, that* $I = \{1,2,3,4,6,11\}$ *is a solution to the R1CS XXX of our problem. To see how this translates to polyinomial divisibility we compute the polynomial* $p_I$ *by*

$$
\begin{aligned}
p_I(x) = &(\sum_{h\in|I|} I_h \cdot a_h(x)) \cdot (\sum_{h\in|I|} I_h \cdot b_h(x)) - (\sum_{h\in|I|} I_h \cdot c_h(x)) \\
=&(2(6x+10)+6(7x+4)) \cdot (3(6x+10)+4(7x+4)) - (6(6x+10)+11(7x+4)) \\
=&((12x+7)+(3x+11)) \cdot ((5x+4)+(2x+3)) - ((10x+8)+(12x+5)) \\
=&(2x+5) \cdot (7x+7) - (9x) \\
=&(x^2+2\cdot 7x+5\cdot 7x+5\cdot 7) - (9x) \\
=&(x^2+x+9x+9) - (9x) \\
=&x^2+x+9
\end{aligned}
$$

*And as we can see in this particular example* $p_I(x)$ *is equal to the target polynomial* $t(x)$ *and hence it is divisible by t with* $p/t = 1$.

*To give a counter example we already know from XXX that* $I = \{1,2,3,4,8,2\}$ *is not a solution to our R1CS. To see how this translates to polyinomial divisibility we compute the polynomial* $p_I$ *by*

$$
\begin{aligned}
p_I(x) = &(\sum_{h\in|I|} I_h \cdot a_h(x)) \cdot (\sum_{h\in|I|} I_h \cdot b_h(x)) - (\sum_{h\in|I|} I_h \cdot c_h(x)) \\
=&(2(6x+10)+6(7x+4)) \cdot (3(6x+10)+4(7x+4)) - (6(6x+10)+11(7x+4)) \\
=&8x^2+11x+3
\end{aligned}
$$

*This polynomial is not divisible by the target polynomial t since Not divisible by t:* $(8x^2+11x+3)/(x^2+x+9) = 8 + \frac{3x+8}{x^2+x+9}$

### 7.3.3 Quadratic span programs

## 7.4 proof system

Now a *proof system* is nothing but a game between two parties, where one parties task is to convince the other party, that a given string over some alphabet is a statement is some agreed on language. To be more precise. Such a system is more over *zero knowledge* if this possible without revealing any information about the (parts of) that string.

**Definition 7.4.0.1** ((Interactive) Proofing System)**.** *Let $L$ be some formal language over an alphabet $\Sigma$. Then an **interactive proof system** for $L$ is a pair $(P,V)$ of two probabilistic interactive algorithms, where $P$ is called the **prover** and $V$ is called the **verifier**.*

*Both algorithms are able to send messages to one another. Each algorithm only sees its own state, some shared initial state and the communication messages.*

*The verifier is bounded to a number of steps which is polynomial in the size of the shared initial state, after which it stops in an accept state or in a reject state. We impose no restrictions on the local computation conducted by the prover.*

*We require that, whenever the verifier is executed the following two conditions hold:*

- *(Completeness) If a string $x \in \Sigma^*$ is a member of language $L$, that is $x \in L$ and both prover and verifier follow the protocol; the verifier will accept.*
- *(Soundness) If a string $x \in \Sigma^*$ is not a member of language $L$, that is $x \notin L$ and the verifier follows the protocol; the verifier will not be convinced.*
- *(Zero-knowledge) If a string $x \in \Sigma^*$ is a member of language $L$, that is $x \in L$ and the prover follows the protocol; the verifier will not learn anything about $x$ but $x \in L$.*

In the context of zero knowledge proving systems definition XXX gets a slight adaptation:

- Instance: Input commonly known to both prover (P) and verifier (V), and used to support the statement of what needs to be proven. This common input may either be local to the prover-verifier interaction, or public in the sense of being known by external parties (Some scientific articles use "instance" and "statement" interchangeably, but we distinguish between the two.).
- Witness: Private input to the prover. Others may or may not know something about the witness.
- Relation: Specification of relationship between instances and witness. A relation can be viewed as a set of permissible pairs (instance, witness).
- Language: Set of statements that appear as a permissible pair in the given relation.
- Statement:Defined by instance and relation. Claims the instance has a witness in the relation(which is either true or false).

The following subsections define ways to describe checking relations that are particularly useful in the context of zero knowledge proofing systems

## 7.4.1 Succinct NIZK

Preprocessing style: trusted setup, multi party ceremony

Blum, Feldman and Micali extended the notion tonon-interactivezero-knowledge(NIZK) proofs in the common reference string model. NIZK proofs are useful in theconstruction of non-interactive cryptographic schemes, e.g., digital signatures and CCA-secure public key encryption.

**Definition 7.4.1.1.** *Let $\mathscr{R}$ be a relation generator that given a security parameter $\lambda$ in unary returns a polynomial time decidable binary relation $R$. For pairs $(i,w) \in R$ we call $i$ the instance*[1]

[1] Note that in Groth16 this is called the statement. We think the term instance is more consistent with SOMETHING.

*and w the witness. We define $R_\lambda$ to be the set of possible relations R the relation generator may output given $1^\lambda$. We will in the following for notational simplicity assume $\lambda$ can be deduced from the description of R. The relation generator may also output some side information, an auxiliary input z, which will be given to the adversary. An efficient prover publicly verifiable non-interactive argument for R is a quadruple of probabilistic polynomial algorithms* $(\textsc{Setup}, \textsc{Prove}, \textsc{Vfy}, \textsc{Sim})$ *such*

- *Setup:* $(CRS, \tau) \to Setup(R)$*: The setup produces a common reference string CRS and a simulation trapdoor $\tau$ for the relation R.*
- *Proof:* $\pi \to Prove(R, CRS, i, w)$*: The prover algorithm takes as input a common reference string CRS and a statement $(i, w) \in R$ and returns an argument $\pi$.*
- *Verify:* $0/1 \to Vfy(R, CRS, i, \pi)$*: The verification algorithm takes as input a common reference string CRS, an instance i and an argument $\pi$ and returns 0 (reject) or 1 (accept).*
- $\pi \to Sim(R, \tau, i)$*: The simulator takes as input a simulation trapdoor $\tau$ and instance i and returns an argument $\pi$.*

**Common Reference String Generation** Also called trusted setup phase. The field elements needed in this step are called toxic waste ...

**Trusted third party** The most simple approach to generate a common reference string is a so called *trusted third party*. By assumption the entire systems trusts this party to generate the common reference string exactly accoring to the rules and the party will delete all traces of the toxic waste after CRS generation.

**Player exchangeable Multi Party Ceremonies** Achive soundness if only a single party is honest and correctly deletes toxix waste. Is always zero knowledge.

State of the art works in the random beacon model.

A random beacon produces publicly available and verifyable random values at fixed intervals. The difference between random beacons and random oracles, is that random beacons are not available until certain time slots. Random beacons can be instanciated for example by evaluation of say $2^{40}$ iterations of SHA256 on some high entropy, publically available data like the closing value of the stock market on a certain date, the output of a selected set of national lotteries and so on.

The assumption is that any given random beacon value contains large amounts of entropy that is independent from the influence of an adversary in previous time slots.

### Groth16

Groth's constant size NIZK argument is based on constructing a set of polynomial equations and using pairings to efficiently verify these equations. Gennaro, Gentry,Parno and Raykova [Pinocchio] found an insightful construction of polynomial equations based on Lagrange interpolation polynomials yielding a pairing-based NIZK argumentwith a common reference string size proportional to the size of the statement and wit-ness.

It constructs a snark for arithmetic circuit satisfiability, where a proof consists of only 3 group elements. In addition to being small, the proof is also easy to verify. The verifier just needs to compute a number of exponentiations proportional to the instance size and check a single pairing product equation, which only has 3 pairings.

The construction can be instantiated with any type of pairings including Type III pairings, which are the most efficient pairings. The argument has perfect completeness and perfect zero-knowledge. For soundness ??

In the common reference string model.

Setup:

- random elements $\alpha, \beta, \gamma, \delta, s \in \mathbb{F}_{scalar}$
- Common reference string $CRS_{QAP}$, specific to the $QAP$ and the choice of statement and witness $CRS_{QAP} = (CRS_{\mathbb{G}_1}, CRS_{\mathbb{G}_2})$, with $n = deg(t)$:

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{c} [\alpha]g, [\beta]g, [\delta]g, \left\{[s^k]g\right\}_{k=0}^{n-1}, \left\{[\frac{\beta a_k(s)+\alpha b_k(s)+c_k(s)}{\gamma}]g\right\}_{k\in I} \\ \left\{[\frac{\beta a_k(s)+\alpha b_k(s)+c_k(s)}{\delta}]g\right\}_{k\in W}, \left\{[\frac{s^k t(s)}{\delta}]g\right\}_{k=0}^{n-2} \end{array} \right\}$$

$$CRS_{\mathbb{G}_2} = \left\{[\beta]h, [\gamma]h, [\delta]h, \left\{[s^k]h\right\}_{k=0}^{n-1}\right\}$$

- Toxic waste: Must delete random elements after $CRS_{QAP}$ generation.

**Example 124** (Generalized factorization snark)**.** *In this example we want to compile our main example in Groth16. Input is the R1CS from example 116. We choose the following global parameters*

$$curve = BLS6\text{-}6 \quad \mathbb{G}_1 = BLS6\text{-}6(13) \quad g = (13,15) \quad \mathbb{G}_2 = \quad h = (7v^2, 16v^3) \text{ and } \mathbb{G}_T = \mathbb{F}_{43^6}^*.$$

**Example 125** (Trusted third party for the factorization snark)**.** *We consider ourself as a trusted third part to generate the common reference string for our generalized factorization snark. We therefore choose the following secret field elements $\alpha = 6$, $\beta = 5$, $\gamma = 4$, $\delta = 3$, $s = 2$ from $\mathbb{F}_{13}$ and are very careful to hide them from anyone how hasn't read this book. From those values we can then instantiate the common reference string XXX:*

$$CRS_{\mathbb{G}_1} = \left\{ \begin{array}{c} [6](13,15), [5](13,15), [3](13,15), \left\{[s^k](13,15)\right\}_{k=0}^{1}, \left\{[\frac{5a_k(2)+6b_k(2)+c_k(2)}{4}](13,15)\right\}_{k\in S} \\ \left\{[\frac{5a_k(2)+6b_k(2)+c_k(2)}{3}](13,15)\right\}_{k\in W}, \left\{[\frac{s^k t(2)}{3}](13,15)\right\}_{k=0}^{0} \end{array} \right\}$$

*Since we have instance indices $I = \{1, in_1, in_2\}$ and witness indices $W = \{in_3, mid_1, out_1\}$ we have The instance parts.*

$$\left[\frac{5a_c(2)+6b_c(2)+c_c(2)}{4}\right](13,15) = \left[\frac{5\cdot 0+6\cdot 0+0}{4}\right](13,15) = [0](13,15) = \mathcal{O}$$

$$\left[\frac{5a_{in_3}(2)+6b_{in_3}(2)+c_{in_3}(2)}{4}\right](13,15) = [(5\cdot 0+6\cdot(7\cdot 2+4)+0)\cdot 10](13,15) = \\ [(6\cdot 5)\cdot 10](13,15) = [1](13,15) = (13,15)$$

$$\left[\frac{5a_{out}(2)+6b_{out}(2)+c_{out}(2)}{4}\right](13,15) = [(5\cdot 0+6\cdot 0+(7\cdot 2+4))\cdot 10](13,15) =$$
$$[5\cdot 10](13,15) = [11](13,15) = (33,9)$$

*Witness part:*

$$\left[\frac{5a_{in_1}(2)+6b_{in_1}(2)+c_{in_1}(2)}{3}\right](13,15) = [(5\cdot(6\cdot 2+10)+6\cdot 0+0)\cdot 9](13,15) =$$
$$[(5\cdot 9)\cdot 9](13,15) = [2](13,15) = (33,34)$$

$$\left[\frac{5a_{in_2}(2)+6b_{in_2}(2)+c_{in_2}(2)}{3}\right](13,15) = [(5\cdot 0+6\cdot(6\cdot 2+10)+0)\cdot 9](13,15) =$$
$$[(6\cdot 9)\cdot 9](13,15) = [5](13,15) = (26,34)$$

$$\left[\frac{5a_{mid_1}(2)+6b_{mid_1}(2)+c_{mid_1}(2)}{3}\right](13,15) = [(5\cdot(7\cdot 2+4)+6\cdot 0+0)\cdot 9](13,15) =$$
$$[(5\cdot 5)\cdot 9](13,15) = [4](13,15) = (35,28)$$

*For* $\left\{\left[\frac{s^k t(2)}{3}\right](13,15)\right\}_{k=0}^{0}$ *we get*

$$\left[\frac{2^0 t(2)}{3}\right](13,15) = [t(2)\cdot 9](13,15) = [(2^2+2+9)\cdot 9](13,15) = [5](13,15) = (26,34)$$

*All together, the* $\mathbb{G}_1$ *part of the CRS is:*

$$CRS_{\mathbb{G}_1} = \left\{\begin{array}{c} (27,34),(26,34),(38,15),\{(13,15),(33,34)\},\{\mathcal{O},(13,15),(33,9)\} \\ \{(33,34),(26,34),(35,28)\},\{(26,34)\} \end{array}\right\}$$

*To compute the* $\mathbb{G}_2$ *part*

$$CRS_{\mathbb{G}_2} = \left\{[5](7v^2,16v^3),[4](7v^2,16v^3),[3](7v^2,16v^3),\left\{[2^k](7v^2,16v^3)\right\}_{k=0}^{1}\right\}$$

$$CRS_{\mathbb{G}_2} = \left\{[5](7v^2,16v^3),[4](7v^2,16v^3),[3](7v^2,16v^3),\left\{[1](7v^2,16v^3),[2](7v^2,16v^3)\right\}\right\}$$
$$CRS_{\mathbb{G}_2} = \left\{(16v^2,28v^3),(37v^2,27v^3),(42v^2,16v^3),\left\{(7v^2,16v^3),(10v^2,28v^3)\right\}\right\}$$

*So alltogether our common reference string is*

$$\left(\begin{array}{c}\left\{\begin{array}{c} (27,34),(26,34),(38,15),\{(13,15),(33,34)\},\{\mathcal{O},(13,15),(33,9)\} \\ \{(33,34),(26,34),(35,28)\},\{(26,34)\} \end{array}\right\} \\ \left\{(16v^2,28v^3),(37v^2,27v^3),(42v^2,16v^3),\left\{(7v^2,16v^3),(10v^2,28v^3)\right\}\right\}\end{array}\right)$$

**Example 126** (Player exchangeable multi party ceremony for the factorization snark)**.** *In this example we want to simulate a real world player exchangeable multi party ceremony for our factorization snark XXX as explained in XXX.*

*We use our TinyMD5 hash function XXX to hash to* $\mathbb{G}_2$*.*

*We assume that we have a coordinator Alice together with three parties Bob, Carol and Dave that want to contribute their randomness to the protocol. Since the degree n of the target polynomial is* 2*, we need to compute the common reference string*

$$CRS = \{\}$$

*For contributer* $j > 0$ *in phase l to compute the proof of knowledge XXX, we need to define the* $transcript_{l,j-1}$ *of the previous round. We define it as sha256 of* $MPC_{l,j-1}$*. To be more precise we define*

$$transcript_{1,j-1} = MD5('[s]g_1[s]g_2[s^2]g_1[\alpha]g_1[\alpha\cdot s]g_1[\beta]g_1[\beta]g_2[\beta\cdot s]g_1')$$

*The only thing actually important about the transcript, is that it is publically available data that is not accesable for anyone before the MPC-data of round* $j-1$ *in phase l exists.*

*We start with the first round usually called the 'powers of tau' EXPLAIN THAT TERM... The computation is initialized With* $s = 1$, $\alpha = 1$, $\beta = 1$. *Hence the computation starts with the following data*

$$MPC_{1,0} = \left\{\begin{array}{lcl} ([s]g_1,[s]g_2) & = & ((13,15),(7v^2,16v^3)) \\ [s^2]g_1 & = & (13,15) \\ [\alpha]g_1 & = & (13,15) \\ [\alpha\cdot s]g_1 & = & (13,15) \\ ([\beta]g_1,[\beta]g_2) & = & ((13,15),(7v^2,16v^3)) \\ [\beta\cdot s]g_1 & = & (13,15) \end{array}\right\}$$

*Then*

$$\begin{aligned} &transcript_{1,0} = \\ &\quad MD5('(13,15)(7v^2,16v^3)(13,15)(13,15)(13,15)(13,15)(7v^2,16v^3)(13,15)') = \\ &\qquad f2baea4d3dba5eef5c63bb210920e7d9 \end{aligned}$$

*We obtain that hash by computing*

$printf'\%s'"(13,15)(7v\hat{}2,16v\hat{}3)(13,15)(13,15)(13,15)(13,15)(7v\hat{}2,16v\hat{}3)(13,15)"|md5sum$

*Everyone agreed, that the MPC starts on the 21.03.2020 and everyone can contribute for exactly a year until the 20.03.2021.*

*It then proceeds in a round robin style, starting with Bob, who optains that data in* $MPC_{1,0}$ *and then computes his contribution. Lets assume that Bob is honest and that bought a 13-sided dice (PICTURE OF 13-SIDED DICE) to randomly find three secret field values from our prime field* $\mathbb{F}_{13}$*. He though the dice and got* $\alpha = 4$, $\beta = 8$ *and* $s = 2$*. He then updates* $MPC_{1,0}$*:*

$$MPC_{1,1} = \left\{\begin{array}{lclcl} ([s]g_1,[s]g_2) & = & ([2](13,15),[2](7v^2,16v^3)) & = & ((33,34),(10v^2,28v^3)) \\ [s^2]g_1 & = & [4](13,15) & = & (35,28) \\ [\alpha]g_1 & = & [4](13,15) & = & (35,28) \\ [\alpha\cdot s]g_1 & = & [8](13,15) & = & (26,9) \\ ([\beta]g_1,[\beta]g_2) & = & ([8](13,15),[8](7v^2,16v^3)) & = & ((26,9),(16v^2,15v^3)) \\ [\beta\cdot s]g_1 & = & [3](13,15) & = & (38,15) \end{array}\right\}$$

*In addition he compute*

$$POK_{1,1}\left\{\begin{array}{lcl} y_s & = & POK(2,f2baea4d3dba5eef5c63bb210920e7d9) = ((33,34),(16v^2,28v^3)) \\ y_\alpha & = & POK(4,f2baea4d3dba5eef5c63bb210920e7d9) = ((35,28),(10v^2,15v^3)) \\ y_\beta & = & POK(8,f2baea4d3dba5eef5c63bb210920e7d9) = ((26,9),(16v^2,28v^3)) \end{array}\right\}$$

*since* $[s]g_1 = (33,34)$, $[\alpha]g_1 = (35,28)$ *and* $[\beta]g_1 = (26,9)$. *as well as*

$$\begin{aligned}
&TinyMD5_2('(33,34)f2baea4d3dba5eef5c63bb210920e7d9') = \\
&H_2(MD5('(33,34)f2baea4d3dba5eef5c63bb210920e7d9').trunc(3)) = \\
&H_2(2066b3b6b6d97c46c3ac6ee2ccd23ad9.trunc(3)) = H_2(ad9) = \\
&H_2(101011011001) = \\
&[8\cdot 4^1\cdot 5^0\cdot 7^1](7v^2,16v^3)+[12\cdot 1^0\cdot 3^1\cdot 8^1](42v^2,16v^3)+ \\
&[2\cdot 3^0\cdot 9^1\cdot 11^1](17v^2,15v^3)+[3\cdot 6^0\cdot 9^0\cdot 10^1](10v^2,15v^3) = \\
&[8\cdot 4\cdot 7](7v^2,16v^3)+[12\cdot 3\cdot 8](42v^2,16v^3)+[2\cdot 9\cdot 11](17v^2,15v^3)+[3\cdot 10](10v^2,15v^3) = \\
&[8\cdot 4\cdot 7](7v^2,16v^3)+[12\cdot 3\cdot 8](42v^2,16v^3)+[2\cdot 9\cdot 11](17v^2,15v^3)+[3\cdot 10](10v^2,15v^3) = \\
&[3](7v^2,16v^3)+[2](42v^2,16v^3)+[3](17v^2,15v^3)+[4](10v^2,15v^3) = \\
&[3](7v^2,16v^3)+[2*3](7v^2,16v^3)+[3*7](7v^2,16v^3)+[4*11](7v^2,16v^3) = \\
&(42v^2,16v^3)+(17v^2,28v^3)+(16v^2,15v^3)+(16v^2,28v^3) = \\
&[3](7v^2,16v^3)+[6](7v^2,16v^3)+[8](7v^2,16v^3)+[5](7v^2,16v^3) = \\
&[3+6+8+5](7v^2,16v^3) = (37v^2,16v^3)
\end{aligned}$$

*So we get* $[2](37v^2,16v^3) = (16v^2,28v^3)$

===================

$$\begin{aligned}
&TinyMD5_2('(35,28)f2baea4d3dba5eef5c63bb210920e7d9') = \\
&H_2(MD5('(35,28)f2baea4d3dba5eef5c63bb210920e7d9').trunc(3)) = \\
&H_2(ad54fa3674f6a84fab9208d7a94c9163.trunc(3)) = H_2(163) = \\
&H_2(000101100011) = \\
&[8\cdot 4^0\cdot 5^0\cdot 7^0](7v^2,16v^3)+[12\cdot 1^1\cdot 3^0\cdot 8^1](42v^2,16v^3)+ \\
&[2\cdot 3^1\cdot 9^0\cdot 11^0](17v^2,15v^3)+[3\cdot 6^0\cdot 9^1\cdot 10^1](10v^2,15v^3) = \\
&[8](7v^2,16v^3)+[12\cdot 8](42v^2,16v^3)+[2\cdot 3](17v^2,15v^3)+[3\cdot 9\cdot 10](10v^2,15v^3) = \\
&[8](7v^2,16v^3)+[5](42v^2,16v^3)+[6](17v^2,15v^3)+[10](10v^2,15v^3) = \\
&[8](7v^2,16v^3)+[5*3](7v^2,16v^3)+[6*7](7v^2,16v^3)+[10*11](7v^2,16v^3) = \\
&(16v^2,15v^3)+(10v^2,28v^3)+(42v^2,16v^3)+(17v^2,28v^3) = \\
&[8](7v^2,16v^3)+[2](7v^2,16v^3)+[3](7v^2,16v^3)+[6](7v^2,16v^3) = \\
&[8+2+3+6](7v^2,16v^3) = (17v^2,28v^3)
\end{aligned}$$

*So we get* $[4](17v^2,28v^3) = (10v^2,15v^3)$

$$
\begin{aligned}
TinyMD5_2('(26,9)f2baea4d3dba5eef5c63bb210920e7d9') = \\
H_2(MD5('(26,9)f2baea4d3dba5eef5c63bb210920e7d9').trunc(3)) = \\
H_2(b87b632f7027ad78cadc2452beb30e9a.trunc(3)) = H_2(e9a) = \\
H_2(111010011010) = \\
[8\cdot 4^1\cdot 5^1\cdot 7^1](7v^2,16v^3)+[12\cdot 1^0\cdot 3^1\cdot 8^0](42v^2,16v^3)+ \\
[2\cdot 3^0\cdot 9^1\cdot 11^1](17v^2,15v^3)+[3\cdot 6^0\cdot 9^1\cdot 10^0](10v^2,15v^3) = \\
[8\cdot 4\cdot 5\cdot 7](7v^2,16v^3)+[12\cdot 3](42v^2,16v^3)+[2\cdot 9\cdot 11](17v^2,15v^3)+[3\cdot 9](10v^2,15v^3) = \\
[2](7v^2,16v^3)+[10](42v^2,16v^3)+[3](17v^2,15v^3)+[1](10v^2,15v^3) = \\
[2](7v^2,16v^3)+[10*3](7v^2,16v^3)+[3*7](7v^2,16v^3)+[1*11](7v^2,16v^3) = \\
(10v^2,28v^3)+(37v^2,27v^3)+(16v^2,15v^3)+(10v^2,15v^3) = \\
[2](7v^2,16v^3)+[4](7v^2,16v^3)+[8](7v^2,16v^3)+[11](7v^2,16v^3) = \\
[2+4+8+11](7v^2,16v^3) = (7v^2,27v^3)
\end{aligned}
$$

*So we get* $[8](17v^2,28v^3) = (16v^2,28v^3)$

*So Bob publishes* $MPC_{1,1}$ *as well as* $POK_{1,1}$ *and after that its Carols turn. Lets also assume that Carrol is honest. So Carol looks at Bobs data and compute the transcript according to our rules*

$$
\begin{aligned}
&transcript_{1,1} = \\
&\quad MD5('(33,34)(10v^2,28v^3)(35,28)(35,28)(26,9)(26,9)(16v^2,15v^3)(38,15)') = \\
&\qquad fe72e18b90014062682a77136944e362
\end{aligned}
$$

*We obtain that hash by computing*

$printf'\%s'"(33,34)(10v^2,28v^3)(35,28)(35,28)(26,9)(26,9)(16v^2,15v^3)(38,15)"|md5sum$

*Carol then computes here contribution. Since she is honest she chooses randomly three secret field values from our prime field* $\mathbb{F}_{13}$*, by invoking her compter. She found* $\alpha = 3$*,* $\beta = 4$ *and* $s = 9$ *and updates* $MPC_{1,1}$*:*

$$
MPC_{1,2} = \left\{
\begin{array}{lclcl}
([s]g_1,[s]g_2) & = & ([9](33,34),[9](10v^2,28v^3)) & = & ((26,34),(16v^2,28v^3)) \\
{[s^2]g_1} & = & [9\cdot 9](35,28) & = & (13,28) \\
{[\alpha]g_1} & = & [3](35,28) & = & (13,28) \\
{[\alpha\cdot s]g_1} & = & [3\cdot 9](26,9) & = & (26,9) \\
([\beta]g_1,[\beta]g_2) & = & ([4](26,9),[4](16v^2,15v^3)) & = & ((27,34),(17v^2,28v^3)) \\
{[\beta\cdot s]g_1} & = & [4\cdot 9](38,15) & = & (35,28)
\end{array}
\right\}
$$

*In addition he compute*

$$
POK_{1,2}\left\{
\begin{array}{lcl}
y_s & = & POK(9,fe72e18b90014062682a77136944e362) = ((35,15),(17v^2,28v^3)) \\
y_\alpha & = & POK(3,fe72e18b90014062682a77136944e362) = ((38,15),(17v^2,15v^3)) \\
y_\beta & = & POK(4,fe72e18b90014062682a77136944e362) = ((35,28),(42v^2,27v^3))
\end{array}
\right\}
$$

$$
\begin{aligned}
TinyMD5_2('(35,15)fe72e18b90014062682a77136944e362') &= \\
H_2(MD5('(35,15)fe72e18b90014062682a77136944e362').trunc(3)) &= \\
H_2(115f145ceffdda73e916dc5ba8ae7354.trunc(3)) = H_2(354) &= \\
H_2(001101010100) &= \\
[8\cdot 4^0\cdot 5^0\cdot 7^1](7v^2,16v^3)+[12\cdot 1^1\cdot 3^0\cdot 8^1](42v^2,16v^3)+& \\
[2\cdot 3^0\cdot 9^1\cdot 11^0](17v^2,15v^3)+[3\cdot 6^1\cdot 9^0\cdot 10^0](10v^2,15v^3) &= \\
[8\cdot 7](7v^2,16v^3)+[12\cdot 8](42v^2,16v^3)+[2\cdot 9](17v^2,15v^3)+[3\cdot 6](10v^2,15v^3) &= \\
[4](7v^2,16v^3)+[5](42v^2,16v^3)+[5](17v^2,15v^3)+[5](10v^2,15v^3) &= \\
[4](7v^2,16v^3)+[5*3](7v^2,16v^3)+[5*7](7v^2,16v^3)+[5*11](7v^2,16v^3) &= \\
(37v^2,27v^3)+(10v^2,28v^3)+(37v^2,16v^3)+(42v^2,16v^3) &= \\
[4](7v^2,16v^3)+[2](7v^2,16v^3)+[9](7v^2,16v^3)+[3](7v^2,16v^3) &= \\
[4+2+9+3](7v^2,16v^3) = (16v^2,28v^3)&
\end{aligned}
$$

*So we get* $[9](16v^2,28v^3) = (17v^2,28v^3)$

$$
\begin{aligned}
TinyMD5_2('(38,15)fe72e18b90014062682a77136944e362') &= \\
H_2(MD5('(38,15)fe72e18b90014062682a77136944e362').trunc(3)) &= \\
H_2(cc4da0c02c4c1b15e72d6cc6430206ab.trunc(3)) = H_2(6ab) &= \\
H_2(011010101011) &= \\
[8\cdot 4^0\cdot 5^1\cdot 7^1](7v^2,16v^3)+[12\cdot 1^0\cdot 3^1\cdot 8^0](42v^2,16v^3)+& \\
[2\cdot 3^1\cdot 9^0\cdot 11^1](17v^2,15v^3)+[3\cdot 6^0\cdot 9^1\cdot 10^1](10v^2,15v^3) &= \\
[8\cdot 5\cdot 7](7v^2,16v^3)+[12\cdot 3](42v^2,16v^3)+[2\cdot 3\cdot 11](17v^2,15v^3)+[3\cdot 9\cdot 10](10v^2,15v^3) &= \\
[7](7v^2,16v^3)+[10](42v^2,16v^3)+[1](17v^2,15v^3)+[10](10v^2,15v^3) &= \\
[7](7v^2,16v^3)+[10*3](7v^2,16v^3)+[1*7](7v^2,16v^3)+[10*11](7v^2,16v^3) &= \\
(17v^2,15v^3)+(17v^2,28v^3)+(17v^2,15v^3)+(17v^2,28v^3) &= \\
[7](7v^2,16v^3)+[4](7v^2,16v^3)+[7](7v^2,16v^3)+[6](7v^2,16v^3) &= \\
[7+4+7+6](7v^2,16v^3) = (10v^2,15v^3)&
\end{aligned}
$$

*So we get* $[3](10v^2,15v^3) = (17v^2,15v^3)$

$$
\begin{aligned}
TinyMD5_2('(35,28)fe72e18b90014062682a77136944e362') =\\
H_2(MD5('(35,28)fe72e18b90014062682a77136944e362').trunc(3)) =\\
H_2(502323bc55c75f7189fad7999c9f1708.trunc(3)) = H_2(708) =\\
H_2(011100001000) =\\
[8\cdot 4^0\cdot 5^1\cdot 7^1](7v^2,16v^3) + [12\cdot 1^1\cdot 3^0\cdot 8^0](42v^2,16v^3)+\\
[2\cdot 3^0\cdot 9^0\cdot 11^1](17v^2,15v^3) + [3\cdot 6^0\cdot 9^0\cdot 10^0](10v^2,15v^3) =\\
[8\cdot 5\cdot 7](7v^2,16v^3) + [12](42v^2,16v^3) + [2\cdot 11](17v^2,15v^3) + [3](10v^2,15v^3) =\\
[7](7v^2,16v^3) + [12](42v^2,16v^3) + [9](17v^2,15v^3) + [3](10v^2,15v^3) =\\
[7](7v^2,16v^3) + [12*3](7v^2,16v^3) + [9*7](7v^2,16v^3) + [3*11](7v^2,16v^3) =\\
(17v^2,15v^3) + (42v^2,27v^3) + (10v^2,15v^3) + (17v^2,15v^3) =\\
[7](7v^2,16v^3) + [10](7v^2,16v^3) + [11](7v^2,16v^3) + [7](7v^2,16v^3) =\\
[7+10+11+7](7v^2,16v^3) = (37v^2,16v^3)
\end{aligned}
$$

*So we get* $[4](37v^2,16v^3) = (42v^2,27v^3)$

*Dave thinks he can outsmart the syste, Since he is the last to contribute, he just makes up an entirely new MPC, that does not contain any randomness from the previous contributers. He thinks he can do that because, no one can distinguish his* $MPC_{1,3}$ *from a correct one. If this is done in a smart way, he will even be able to compute the correct POKs.*

*So Dave choses* $s = 12$, $\alpha = 11$ *and* $\beta = 10$ *and he will keep those values, hoping to be able to use them later to forge false proofs in the factorization snark. He then compute*

$$
MPC_{1,3} = \left\{
\begin{array}{lcl}
([s]g_1,[s]g_2) & = & ((13,28),(7v^2,27v^3))\\
[s^2]g_1 & = & (13,15)\\
[\alpha]g_1 & = & (33,9)\\
[\alpha\cdot s]g_1 & = & (33,34)\\
([\beta]g_1,[\beta]g_2) & = & ((38,28),(42v^2,27v^3))\\
[\beta\cdot s]g_1 & = & (38,15)
\end{array}
\right\}
$$

*Dave does not delete* $s$, $\alpha$ *and* $\beta$, *because if this is accepted as phase one of the common reference string computation, Dave controls already* $3/4$*-th of the cheating key to forge proofs. So Dave is careful to get the proofs of knowledge right. He computes the transcript of Carols contribution as*

$$
\begin{aligned}
&transcript_{1,2} =\\
&\quad MD5('(26,34)(16v^2,28v^3)(13,28)(13,28)(26,9)(27,34)(17v^2,28v^3)(35,28)') =\\
&\qquad c8e6308fffd47009f5f65e773ae4b499
\end{aligned}
$$

*We obtain that hash by computing*
$printf'\%s'"(26,34)(16v^2,28v^3)(13,28)(13,28)(26,9)(27,34)(17v^2,28v^3)(35,28)"|md5sum$

# 8 Exercises and Solutions

TODO: All exercises we provided should have a solution, which we give here in all detail.