1

FOUNDATIONS OF CODING

COPYRIGHTED MATERIAL

This first chapter is an introduction to the notion of code. It contains the mathematical background, necessary to manipulate the codes, and the coding operations. Through this chapter, the reader will understand the constraints of the transmission of information, starting from historical examples and eventually learning advanced techniques. The knowledge of some historical aspects is pedagogically useful to work first on simple structures. In cryptology, history is also essential for efficient protection against the whole range of known attacks.

The principle is always to start from examples of codes, showing step by step why some mathematical notions are useful so that the codes are short, safe, and efficient. While the following chapters describe recent, elaborate, and currently used protocols for coding, this chapter provides the foundations of this activity.

Simple objects from probability theory, algebra, or algorithmic are introduced along the lines of this chapter, when they become necessary. For example, block coding calls for the definition of structures in which elements can be added, multiplied, which justifies the introduction of groups and fields. The emphasis is always put on the effective construction of introduced structures. This calls for a section on algorithms, as well as polynomials and primitive roots.

This chapter is organized in four sections. The first three allow to focus on the three important mathematical notions related to coding: algorithms and their complexity, which is at the core of coding theory; probabilities, related to stream cipher; and algebra, related to block coding. Then, the last section of this chapter is devoted to

Foundations of Coding: Compression, Encryption, Error Correction, First Edition.
Jean-Guillaume Dumas, Jean-Louis Roch, Éric Tannier and Sébastien Varrette.
© 2015 John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons, Inc.
http://foundationsofcoding.imag.fr

the many facets of decoding, from ambiguity and information loss to secret code breaking.

We will denote by *code* a transformation method that converts the representation of a piece of information into another one. This definition is wide enough to include several mathematical objects (functions, algorithms, and transforms), which will be used throughout this book.

The word *code* will also apply to the result of these transformations[1], namely the encoded information and its structure. But for now, in order to find our way in these definitions and to understand what is actually a code, here is a simple example mixing technologies and eras.

1.1 FROM JULIUS CAESAR TO TELECOPY

In this section, we introduce all the fundamental properties of codes, using one of them built from real examples – currently or formerly used. It is the occasion to define and use the algorithms and their properties, which are constantly used in the whole book.

Suppose that we wish to send a piece of information by fax while guaranteeing the secrecy of the transmission. We could do it with the following code.

1.1.1 The Source: from an Image to a Sequence of Pixels

Transmission by fax enables us to pass images through a phone channel. We want this code to perform the requested transformations in order to obtain – quickly and secretly – an image similar to the original one at the end, in a format that could pass through the channel.

The first transformation process will be performed by the scanner of the device. It consists in reading the image and transforming it into a sequence of pixels that we can visualize as small squares either black or white. This is a code, according to the definition we have given, and we will write it as an algorithm, in a format we will adopt throughout this book.

Encoding algorithm

Input An image
Output A sequence of pixels

The input of the algorithm is also called the *source*, and the output is called the *code*.

The chosen method will be the following: the width of the source image is split into 1728 equal parts; the length is then split into lines so that the image is divided into

[1]The word *code* in natural languages can have several meanings. In our context, we will see that it applies to transformations and their results, but in computer science, it can also mean a computer program. This apparent confusion actually enables one to figure out the link between various mathematical and computer processes. That is why we will keep the word with its multiple meanings.

squares of the same size, 1728 per line. These squares will be considered as pixels. Each pixel is given either the color black if the zone of the image is dark enough or the color white if the image is bright.

This is the first part of our code. The following sections describe the other transformations; one can use to encode the information: compression, encryption, and error correction.

1.1.2 Message Compression

In order to formalize source messages and codes, we define the language in which they are formulated. We call *alphabet* a finite set $V = \{v_1, \dots, v_k\}$ of elements (called *characters*). The *cardinal number* of a finite set V is the number of elements it contains and is denoted by $|V|$.

A sequence of characters belonging to V is called a *string* . We denote by V^* the set of all the strings over V, and V^+ the set of all the strings whose length is not equal to 0. As the alphabet of the code and the alphabet of the source may differ, we will distinguish the *source alphabet* and the *code alphabet* .

For the example of the fax, the source alphabet is the result of the scanner, that is, $\{$*white pixel*, *black pixel*$\}$ and the code alphabet will be the set of bits $\{0, 1\}$. We could send the sequence of pixels directly as bits, as we have an immediate conversion to 0 (for white pixels) and 1 (for black pixels).

However, we can apply some compression principles to this sequence of 0s and 1s. We can imagine that the datasheet we want to send has only a few lines of text and is mainly white – which is very common with faxes. But is there a better way of sending a sequence of, let us say a 10 000 zeros, than simply using a 10 000 zeros? Surely such a method exists, and we have just used one by evoking that long sequence of bits without writing it explicitly. We have indicated that the code is composed of 10 000 zeros rather than writing them down.

The fax code performs that principle of compression line by line (i.e., over 1728-pixels blocks). For each line, Algorithm 1.1 describes precisely the encoding algorithm.

For example, with this method, a completely white line will not be encoded with a sequence of 1728 zeros, but with the code "11011000000 0" which represents "1728 0" in the binary representation that will be sent through the channel. We have replaced a 1728-bit message by a 12-bit message. This is noticeably shorter.

We will give details of this principle of message *compression* – called Run-Length Encoding (RLE) – in Section 2.3.1.

For a better visibility, we used a space character in our representation (between 1728 and 0) but that character does not belong to the code alphabet. In practice, we will see later in this chapter what constraints that statement implies.

Exercise 1.1 *A pixelized image meant to be sent by fax contains a lot of pairs "01." What do you think of the fax code presented before? Give a more effective code.*

Solution on page 281.

Algorithm 1.1 Simplified Fax Encoding for a Line of Pixels

Input M, a sequence of 1728 pixels (from $M[0]$ to $M[1727]$)
Output C, the compressed message of M

```
1:  n ← 1 // counter of consecutive pixels with the same value
2:  C ←"" // empty string
3:  For i from 1 to 1727 do
4:    If M[i − 1] = M[i] then
5:      n ← n + 1 // increment counter of consecutive pixels with value M[i]
6:    else
7:      Append n and the color of the last pixel M[i − 1] to C
8:      n ← 1 // re-initialize the counter
9:    End If
10: End For
11: Append n and the col-our of the pixel M[1727] to C
12: return C
```

1.1.3 Error Detection

All the readers who have already used a phone channel will not be surprised to learn that its reliability is not infinite. Each message is likely to be distorted, and it is not impossible, assuming that "11011000000 0" was sent on the channel, to receive "11010000000 0" (modification of one bit) or "1101000000 0" (loss of one bit).

Phone channels usually have an error rate between 10^{-4} and 10^{-7}, depending on their nature, which means that they can commit an error every 10,000 bits. This is far from being negligible when you consider long messages, and it can also modify their meaning. For an image, if the value 1728 becomes 1664 because of the loss of one single bit, a shift will occur in the final image and the result will be unusable.

The fax code enables the detection of such transmission errors. If an error is detected on a line, one can ask for a second transmission of the same line to have a confirmation and – as it is unlikely to have the same error twice at the same place – the message will be corrected.

The principle of error detection in the fax code is explained as follows: predetermined sequences are appended at the beginning and at the end of each line. Such flag sequences are used to establish bit and line synchronization (and in particular detect loss of bit). In order to illustrate this principle, let us suppose that "0" (respectively "1729") is added at the beginning (respectively the end) of each line even if these are not the exact values that are used in practice. The receiver can then check that each line is in the format "0 $n_1 b_1 \ \dots \ n_k b_k$ 1729," with n_i an integer that provides the number of consecutive bits and b_i the color of these bits. In particular, the condition $n_1 + \cdots + n_k = 1728$ must be respected and the colors b_i must be alternated. Thus, a modification or the loss of one bit is easily detected as soon as this format is not respected.

All error detection and correction principles, which will be closely studied in Chapter 4, are based on this principle: the addition of information in order to check the consistency of the received message.

1.1.4 Encryption

Now, let us suppose that, after having compressed the message and set up a format that enables error detection, we wish to keep the message secret for everybody but its recipient. The phone channel, like most channels, does not provide secrecy in itself. Every message that is transmitted through it can be easily read by a third party. The setting up of the secret consists in transforming the original message, the *plaintext*, putting it into a nonunderstandable form, the *ciphertext*, and putting it back into its original form at the end.

This is a technique that has been used by men as they started to communicate. In secret codes of Antiquity, the secret resided in the method that was used to produce the ciphertext, which is what we call the encryption algorithm nowadays. For example, historians have discovered messages encrypted by Julius Caesar's secret services. The messages were texts, and the algorithm substituted each letter of the original message M with the letter located three positions later in the alphabet. For the three last letters of the alphabet, the three first letters were used. For example, the word *TROY* became *WURB*. Hence, the text did not have any immediate signification. That is what is called *mono-alphabetic substitution*, as each letter is replaced by another one (always the same) in the message.

If Caesar had wanted to send a fax, he would have adapted his code to the numerical format, which would have given the function $f(x) = x + 3 \mod n$ for every number sent on the channel where the number n is the size of the alphabet. Here it would have been 1730 as no number greater than 1729 would theoretically be used.

These encryption and decryption functions were then extended with a simple key K, an integer chosen secretly between the interlocutors. This is equivalent to the construction of a function $f_K(x) = x + K \mod n$. As for the Spartans, they used a completely different encryption algorithm, called transposition encryption. Their system is based on the Scytale , a stick on which a strip was rolled. The message was written on the strip along the stick rather than along the scroll. This means that the consecutive letters of the message appeared on a circumlocution different from the one of the strip.

Figure 1.1 illustrates the encryption principle. In order to decrypt the message, the recipient would have a stick with the same diameter as the stick used for the encryption.

Other cryptographic systems, more evolved, were created afterward (affine encryption $f_{a,b}(x) = a.x + b \mod n$ – studied in Exercises 1.2 and 3.1; substitution encryption where each letter is replaced by a symbol of another alphabet – such as the Morse code, etc…).

Exercise 1.2 *Mark Antony intercepts a message sent by Caesar, encrypted with an affine encryption. Without knowing the key* (a, b)*, how can he decrypt the message?*

Solution on page 282.

1.1.5 Decryption

In Section 1.1.4, we have seen some methods for encrypting a message and insuring the secrecy and the efficiency of its transmission. This description would not be

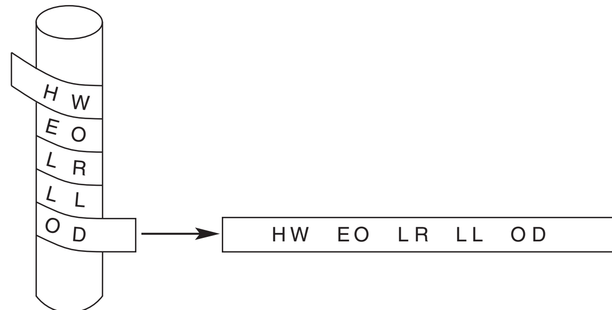

Figure 1.1 Spartan Scytale principle to transmit the text "HELLOWORLD"

complete without a short presentation of the principles of the complementary operation – decryption.

1.1.5.1 Attack and Decryption The encrypted message arrives. It is a matter of decrypting it. If one is not given the key or the encryption method, we talk about an *attack* on the code, or a *codebreaking*. The discipline which handles the development of methods of attacking existing codes or of building more resistant codes is called *cryptanalysis*. For the recipient of the message – who knows the key – we talk about *decryption*.

We will soon talk about attacks. For now, the decryption method is very simple and consists in applying the inverse function of the encryption function, possibly customized with a key, namely $f_K^{-1}(y) = y - K \mod n$. In our case, the recipient of Caesar's fax would have to apply $f_3^{-1}(x) = y - 3 \mod 1730$ for every number received on the channel The message is now decrypted.

Now, we still have to perform the inverse of all the remaining transformations. First of all, the format of each line is checked in order to detect possible errors (in this case, we ask for a second transmission), then the decoding algorithm is performed which – given a sequence of numbers – will return the initial pixel sequence. Algorithm 1.2 formalizes this operation for each line.

Algorithm 1.2 Fax Decoding for a Decrypted and Checked Line

Input A sequence of numbers in the format "0 n_1 b_1 … n_{1728} b_{1728} 1729"
Output A sequence of 1728 corresponding pixels
1: **For** i from 1 to 1728 **do**
2: Draw n_i pixels with color b_i
3: **End For**

1.1.5.2 Decompression and Data Loss When we apply this algorithm to all the lines, we obtain a sequence of pixels, identical to the one we built from the original image. Now, we can recover the original image, or at least a similar image as the

only information we have is the value of the pixels. Thus, the method consists in printing the sequence of corresponding black and white squares on a sheet of paper. The image resulting from this operation will be a damaged version of the initial one, as the original squares were not completely black or white. That is what gives the nonaesthetic aspect of all faxes, but the essential information is preserved.

In this case, initial information is not entirely recovered – and arrives in an approximate form – thus we talk about *encoding with (data-)loss*. One often uses codes which admit a determined level of loss, providing that important information is not distorted. It is often the case with multimedia information (see Section 2.5).

1.1.6 Drawbacks of the Fax Code

We have completely described the encoding – and decoding – processes, while respecting the major constraints we will encounter in coding (compression efficiency, error detection, and secrecy). But for several reasons, the principles used in the fax code are not very usable in practice in common numerical communications.

Compression: the way the RLE principle is applied here has several drawbacks. First of all, if the message is only composed of alternating black and white pixels, the size of the "compressed" message will be greater than that of the original. Thus, there is absolutely no guarantee of the compression of the message. Moreover, including the bits b_i is almost useless as they are alternated and the value of the first one would be enough to determine the value of the others. The fax code removes them. It then becomes a sequence of numbers $n_1 \dots n_k$. But that sequence – encoded as bits (to each number its binary representation) – is more difficult to decode. Indeed, when receiving a bit sequence "1001011010010," how do we know whether it represents a number n_i or several numbers appended ? In order to solve this issue, for example, we can encode every number with the same number of bits but then the compression algorithm is no longer optimal. We are now forced to represent "2" with the sequence "00000000010," which increases the size of the message. We will see how to insure that a message can be decoded later in this chapter. In Chapter 2, we will see how to deduce good compression methods while insuring this decoding ability.

Error detection: this principle implies that one has to ask for a second transmission of information each time an error is detected, whereas we could develop codes that automatically correct the channel errors (Chapter 4). Moreover, there is no theoretical guarantee that this code adapts to phone channel error rates. We do not know the error rate detected by the code, nor if it can accomplish the same performance adding less information in the messages. Efficiency of the transmission may depend on the quality of error detection and correction principles.

Secrecy: Caesar's code is breakable by any beginner in cryptology. It is easy to decode any *mono-alphabetic substitution* principle, even without knowing the key. A simple cryptanalysis consists of studying the frequency of appearance

TABLE 1.1 Letter Distribution in this LaTeX Script

E	13.80%	T	8.25%	I	8.00%	N	7.19%	A	6.63%
O	6.57%	R	6.37%	S	6.28%	C	4.54%	D	4.35%
L	4.34%	U	3.34%	M	3.15%	P	2.92%	H	2.81%
F	2.36%	G	1.86%	B	1.83%	X	1.35%	Y	1.07%
V	0.87%	W	0.69%	K	0.57%	Q	0.55%	Z	0.16%
J	0.14%								

of the letters in the ciphertext and of deducing the correspondence with the letters of the plaintext.

Table 1.1 presents the statistical distribution of the letters in this document written in LaTeX (LaTeX special tags are also considered). As this book is quite long, we can assume that these frequencies are representative of scientific texts in English written in LaTeX. It is obviously possible to obtain such frequency tables for literary English texts, scientific French texts, and so on.

Exercise 1.3 *Scipio discovers a parchment manuscript with the following ciphertext:* `HFJXFW BTZQI MFAJ GJJS UWTZI TK DTZ!` *Help Scipio decrypt this message.* *Solution on page 282.*

1.1.7 Orders of Magnitude and Complexity Bounds for Algorithms

We have described a code and we have presented its advantages and disadvantages. However, there is another critical point we have not yet mentioned: what about encoding and decoding speed? It depends on computer speed, but mostly on the complexity of the algorithms.

Size of numbers. We consider numbers and their size either in decimal digits or in bits. Thus, a number m will be $\lceil \log_{10}(m) \rceil$ digits long and $\lceil \log_2(m) \rceil$ bits long. For instance, 128 bits can be represented with 39 decimal digits, 512 bits with 155 decimal digits, and 1024 bits with 309 decimal digits.

Computer speed. Nowadays, the frequency of any PC is at least 1 GHz. That is to say it can execute 1 billion (10^9) operations per second. By way of comparison, the greatest speed in the universe is the speed of light: 300 000 km/s = 3.10^8 m/s. It takes only a ten thousand millionth of a second for light to cross a room of 3 m long. During this period, your computer has performed 10 operations!!! Hence, we can say that *current computers calculate at the speed of light*.

Comparison with the size and age of the universe. This computing speed is truly astronomical; yet the size of the numbers we manipulate remains huge. Indeed, one has to enumerate 10^{39} numbers just in order to count up to a 39 digit long number. In order to see how huge it is, let us calculate the age of the universe in seconds:

$$\text{Age of the universe} \simeq 15 \text{ billion years} \times 365.25 \times 24 \times 60 \times 60 \approx 5.10^{17} \text{ s.}$$

Thus, a 1 GHz computer would take more than two million times the age of the universe just to count up to a number of “only” 39 digits! As for a 155 digit number (commonly used in cryptography), it is simply the square of the number of electrons in the universe. Indeed, our universe is thought to contain about 3.10^{12} galaxies, each galaxy enclosing roughly 200 billion stars. Knowing that the weight of our sun is approximately 2.10^{30} kg and that the weight of an electron is $0.83.10^{-30}$ kg, we have

$$\text{Universe} = (2.10^{30}/0.83.10^{-30}) * 200.10^{9} * 3.10^{12} \approx 10^{84} \text{ electrons.}$$

Yet such numbers will have to be manipulated. Their size will represent one of the algorithmic challenges we will have to take up, as well as a guarantee of the secrecy of our messages if it takes millions of years of computing – on the billion machines available on Earth, including the fastest ones – to decrypt them without the key. We will see how to build algorithms which can handle such numbers later.

Complexity bounds of the algorithms. There may be many different algorithms for solving the same computational problem. However despite the power of computers, an ill-conceived algorithm could turn out to be unusable. It is therefore of interest to find a way to compare the efficiency of two algorithms. This is done on the basis of a computational model and the *size* of the problem instance, which is a measure of the quantity of input data.

There exists many different computational models. For instance, one can use Turing machines, random-access machines, or Boolean circuits. Yet we will not detail them in this book (see, e.g., [1] for more details). As for the size and in order to be able to compute it in every instance, we assume that the input is a sequence of bits and the size of the input is the length of the sequence. This statement implies that we have to build a code that transforms the input of an algorithm into a sequence of bits. This is often a quite simple operation. For instance, it takes about $\log_2(a)$ bits to code an integer a (we simply write it in base 2). Thus, the size of an integer is equal to its logarithm in base 2. The size of a sequence of black and white pixels is the length of that sequence.

The number of basic computer steps needed by an algorithm expressed as a function of the size of the problem is called the *time complexity*[2] of the algorithm. This helps to define a machine-independent characterization of an algorithm’s efficiency as the naive evaluation of an execution time crucially depends on the processor speed and various architecture-specific criteria.

Indeed, the notion of *basic computer step* is vague as the set of processor’s instruction has a variety of basic primitives (branching, comparing, storing, performing arithmetic operations, etc.). However, all of them can be evaluated in terms of *bit operations*; Therefore, the true complexity of an algorithm is computed using this elementary unit. For the sake of simplicity, one sometimes counts only the number of arithmetical operations requested by the algorithm, typically limited to the four

[2]Analogous definition can be made for *space complexity* to measure the amount of space, or memory required by the algorithm.

classical operations (addition, subtraction, multiplication, and division) and some binary operations such as bit shift. In that case, it should be reminded that the conversion to bit operation depends on the base field.

As it is often impossible to count exactly all the basic computer steps performed during the execution of a program, the complexity of an algorithm is generally given with asymptotic upper bounds and lower bounds using the Big-O notation (also called Landau notation). More formally, if $f(n)$ and $g(n)$ are functions from positive integers to positive reals, then for any n large enough, it is said that

- $f(n) = O(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \times g(n)$. This defines g as an asymptotic upper bound for f. Intuitively, it means that f grows no faster asymptotically than $g(n)$ to within a constant multiple;
- $f(n) = \Omega(g(n))$ if there exists a constant $c > 0$ such that $f(n) \geq c \times g(n)$; This defines g as an asymptotic lower bound for f;
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; and This defines g as an asymptotic tight bound for f;

Therefore, if an algorithm takes for instance, $14n^2 + 2n + 1$ steps on an input of size n, we simply say it runs with a time complexity bound of $O(n^2)$. Table 1.2 summarizes the typically met complexity functions and their associated names. AS Landau notation enables us to give the complexity to within a constant multiple, we can write $O(\log(n))$ without giving the base of the logarithm, knowing that the base multiplies the function by a constant (namely the logarithm of the base).

Exercise 1.4 *What is the complexity of the fax code presented in the previous section?* *Solution on page 282.*

In practice, every algorithm should have a linear complexity $O(n)$ – where n is the size of the source message – to be "real-time" efficient, that is, usable in a transmission when a long latency is not acceptable (telephony, multimedia, etc.).

One of the fields of cryptography relies on the hypothesis that there exists some problems where no algorithm with such complexity is known. For such problems, every known algorithm requests astronomical computing time – which makes their application impossible.

TABLE 1.2 Typical Complexity Bounds Obtained from an Algorithm Analysis

Function	Complexity	Example
$O(1)$	Constant	Table lookup
$O(\log n)$	Logarithmic	Binary search on a sorted vector
$O(n)$	Linear	Traversing of an unsorted vector
$O(n \log n)$	Quasilinear	Quicksort, FFT (Section 1.4.2.4)
$O(n^k), k > 1$	Polynomial	Naive square matrix multiplication – $O(n^3)$
$O(k^n), k > 1$	Exponential	Naive Fibonacci – $O(1.6^n)$

In this book, we consider that a problem is impossible to solve (one often uses the euphemism “difficult”) when there is no known algorithm which solves it in humanly reasonable time, for instance, if its resolution would request more than 10^{50} operations.

1.2 STREAM CIPHERS AND PROBABILITIES

As mentioned in Section 1.1.3, the probability of transmission error in many communication channels is high. To build an efficient code in such a situation, one can consider the message to be transmitted as a *bit stream*, that is, a potentially infinite sequence of bits sent continuously and serially (one at a time). Each character is then transformed bit by bit for the communication over the channel. Such an approach has various advantages:

- the transformation method can change for each symbol;
- there is no error propagation on the channel; and
- it can be implemented on embedded equipment with limited memory as only a few symbols are processed at a time.

This technique is notably used in cryptography in the so-called *stream cipher*. Under some conditions (Section 1.2.1), such systems can produce *unconditionally secure* messages: for such messages, the knowledge of the ciphertext does not give any information on the plaintext. This property is also called *perfect secrecy*. Thus, the only possible attack on a perfect encryption scheme given a ciphertext is the exhaustive research of the secret key (such a strategy is also called brute force attack). We also use the stream cipher model to build error detection and correction principles (see convolutional codes in Chapter 4). The *one-time-pad* (OTP) encryption scheme is an example of a cryptographic stream cipher whose unconditional security can be proved, using probability and information theory.

Some notions of probabilities are necessary for this section and also for the rest of the book. They are introduced in Section 1.2.2.

1.2.1 The Vernam Cipher and the One-Time-Pad Cryptosystem

In 1917, during the first World War, the American company AT&T asked the scientist Gilbert Vernam to invent a cipher method the Germans would not be able to break. He came with the idea to combine the characters typed on a teleprinter with the one of a previously prepared key kept on a paper tape. In the 1920s, American secret service captain Joseph Mauborgne suggested that the key should be generated randomly and used only once. The combination of both ideas led to the *OTP* encryption scheme, which is up to now the only cipher to be mathematically proved as unconditionally secure.

The Vernam cipher derived from the method introduced by Gilbert Vernam. It belongs to the class of secret key cryptographic system, which means that the

secret lies in one parameter of the encryption and decryption functions, which is only known by the sender and the recipient. It is also the case of Caesar's code mentioned in Section 1.1.4, in which the secret parameter is the size of the shift of the letters (or the numbers).

Mathematically speaking, the Vernam encryption scheme is as follows: for any plaintext message M and any secret key K of the same size, the ciphertext C is given by

$$C = M \oplus K,$$

where $\oplus$ (also denoted by xor) is the bitwise logical "exclusive or" operation. Actually, it consists in an addition modulo 2 of each bit. This usage of the "exclusive or" was patented by Vernam in 1919. Decryption is performed using the same scheme due to the following property:

$$C \oplus K = (M \oplus K) \oplus K = M.$$

If K is truly random, used only once and of the same size as M i.e., $|K| = |M|$), then a third party does not obtain any information on the plaintext by intercepting the associated ciphertext (except the size of M). Vernam cipher used with those three assumptions is referred to as a OTP scheme. It is again unconditionally secure, as proved by Claude Shannon (Section 1.2.5).

Exercise 1.5 *Why is it necessary to throw the key away after using it, that is, why do we have to change the key for each new message?* *Solution on page 282.*

Exercise 1.6 *Build a protocol, based on the One-time-pad system, allowing a user to connect to a secured server on Internet from any computer. The password has to be encrypted to be safely transmitted through Internet and to avoid being captured by the machine of the user.* *Solution on page 282.*

Obviously, it remains to formalize the notion of random number generation, give the means to perform such generation and finally, in order to prove that the system is secure, make precise what "obtaining some information" on the plaintext means. For this, the fundamental principles of information theory – which are also the basis of message compression – are now introduced.

1.2.2 Some Probability

In a cryptographic system, if one uses a key generated randomly, any discrepancy with "true" randomness represents an angle of attack for the cryptanalysts. Randomness is also important in compression methods as any visible order, any redundancy, or organization in the message can be used not only by code breakers but also by code inventors who will see a means of expressing the message in a more dense form. But what do we call discrepancy with true randomness, and more simply what do we call randomness ? For instance, if the numbers "1 2 3 4 5 6" are drawn at lotto, one will

doubt that they were really randomly generated, although *stricto sensu*, this combination has exactly the same probability of appearing as any other. We do not go deeply into the philosophy of randomness, but this section provides mathematical material to address randomness and its effects and to create something close to randomness.

1.2.2.1 Events and Probability Measure An *event* is a possible result of a random experiment. For example, if one throws a six face die, getting the number 6 is an event. The set operators ($\cup$, $\cap$, $\setminus$) are used for relations between events (they stand for *or*, *and*, and *except*). We denote by Ω the set of all possible events for a given experiment. A probability measure P is a map of Ω onto $[0, 1]$ satisfying:

1. $P(\Omega) = 1$ and $P(\emptyset) = 0$;
2. For all A, B mutually exclusive events ($A \cap B = \emptyset$), $P(A \cup B) = P(A) + P(B)$.

If the set of events is a discrete set or a finite set, we talk about *discrete probability*. For example, if the random experiment is the throw of a fair six face die, the set of events is $\Omega = \{1, 2, 3, 4, 5, 6\}$ and the probability of occurrence of each one is $1/6$.

The probability function is called the *probability distribution* or the *law of probability*. A distribution is said to be *uniform* if all events have the same probability of occurrence.

Gibbs' lemma is a result on discrete distributions that will be useful several times in this book.

Lemma 1 (Gibbs' Lemma) *Let $(p_1, \dots, p_n)$, $(q_1, \dots, q_n)$ be two discrete probability laws with $p_i > 0$ and $q_i > 0$ for all i. Then*

$$\sum_{i=1}^{n} p_i * \log_2 \frac{q_i}{p_i} \leq 0$$

with equality if and only if $p_i = q_i$ for all i.

Proof. As $\log_2(x) = \frac{\ln x}{\ln 2}$ and $\ln(2) > 0$, it is sufficient to prove the statement using the neperian logarithm. Indeed, $\forall x \in \mathbb{R}_*^+$, $\ln(x) \leq x - 1$ with equality if and only if $x = 1$. Hence $\sum_{i=1}^{n} p_i * \ln \frac{q_i}{p_i} \leq \sum_{i=1}^{n} p_i * (\frac{q_i}{p_i} - 1)$.

Having $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i = 1$, we can deduce that $\sum_{i=1}^{n} p_i * (\frac{q_i}{p_i} - 1) = \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i = 0$. As a result, $\sum_{i=1}^{n} p_i * \ln \frac{q_i}{p_i} \leq 0$. The equality holds if $q_i = p_i$ for all i so that the approximation $\ln(\frac{q_i}{p_i}) \leq \frac{q_i}{p_i} - 1$ is exact. □

1.2.2.2 Conditional Probabilities and Bayes' Formula Two events are said to be *independent* if $P(A \cap B) = P(A)P(B)$.

The *conditional probability* of an event A with respect to an event B is the probability of appearance of A, knowing that B has already appeared. This probability is denoted by $P(A|B)$ and defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

By recurrence, it is easy to show that for a set of events $A_1, \dots, A_n$,

$$P(A_1 \cap \dots \cap A_n) = P(A_1|A_2 \cap \dots \cap A_n)P(A_2|A_3 \cap \dots \cap A_n) \dots P(A_{n-1}|A_n)P(A_n).$$

Bayes' formula enables us to compute – for a set of events $A_1, \dots, A_n, B$ – the probabilities $P(A_k|B)$ as functions of the probabilities $P(B|A_k)$.

$$P(A_k|B) = \frac{P(A_k \cap B)}{P(B)} = \frac{P(B|A_k)P(A_k)}{\sum_i P(B|A_i)P(A_i)}$$

Exercise 1.7 *One proposes the following secret code, which encodes two characters a and b with three different keys k_1, k_2, and k_3: if k_1 is used, then $a \to 1$ and $b \to 2$; if k_2 is used, then $a \to 2$ and $b \to 3$. Otherwise, $a \to 3$ and $b \to 4$.*
Moreover, some a priori *knowledge concerning the message M and the key K is assumed: $P(M = a) = 1/4$, $P(M = b) = 3/4$, $P(K = k_1) = 1/2$, and $P(K = k_2) = P(K = k_3) = 1/4$.*
What are the probabilities of having the ciphertexts 1, 2, and 3? What are the conditional probabilities that the message is a or b knowing the ciphertext? Intuitively, can we say that this code has a "perfect secrecy"? *Solution on page 282.*

Having revised the basic theory of random events, now let us present what randomness means for computers.

1.2.3 Entropy

1.2.3.1 Source of Information Let S be the alphabet of the source message. Thus, a message is an element of S^+. For each message, we can compute the frequencies of appearance of each element of the alphabet and build a probability distribution over S.

A *source of information* is constituted by a couple $\mathcal{S} = (S, \mathcal{P})$ where $S = (s_1, \dots, s_n)$ is the source alphabet and $\mathcal{P} = (p_1, \dots, p_n)$ is a probability distribution over S, namely p_i is the probability of occurrence of s_i in an emission. We can create a source of information with any message by building the probability distribution from the frequencies of the characters in the message.

The source of information $\mathcal{S} = (S, \mathcal{P})$ is said *without memory* when the events (occurrences of a symbol in an emission) are independent and when their probabilities remain stable during the emission (the source is *stable*).

The source is said to follow a *Markov model* if the probabilities of the occurrence of the characters depend on the one issued previously. In the case of dependence on one predecessor, we have the probability set $\mathcal{P} = \{p_{ij}\}$, where p_{ij} is the probability of appearance of s_i knowing that s_j has just been issued. Hence, we have $p_i = \sum_j p_{ij}$.

For instance, a text in English is a source whose alphabet is the set of Latin letters. The probabilities of occurrence are the frequencies of appearance of each character. As the probabilities strongly depend on the characters that have just been issued (a U is much more probable after a Q than after another U), the Markov model will be more adapted.

An image also induces a source. The characters of the alphabet are the color levels. A sound is a source whose alphabet is a set of frequencies and intensities.

A source $\mathcal{S}$ is *without redundancy* if its probability distribution is uniform. This is obviously not the case for common messages, in which some letters and words are much more frequent than others. This will be the angle of attack of compression methods and also pirates trying to read a message without being authorized.

1.2.3.2 Entropy of a Source Entropy is a fundamental notion for the manipulation of a code. Indeed, it is a measure of both the amount of information we can allocate to a source (this will be useful for the compression of messages) and the level of order and redundancy of a message, the crucial information in cryptography.

The *entropy* of a source $\mathcal{S} = (S, \mathcal{P})$, $S = (s_1, \dots, s_n)$, $\mathcal{P} = (p_1, \dots, p_n)$ is

$$H(\mathcal{S}) = H(p_1, \dots, p_n) = -\sum_{i=1}^{n} p_i \log_2(p_i) = \sum_{i=1}^{n} p_i \log_2\left(\frac{1}{p_i}\right).$$

By extension, one calls entropy of a message the entropy of the source induced by this message, the probability distribution being computed from the frequencies of appearance of the characters in the message.

Property 1 *Let $\mathcal{S} = (S, \mathcal{P})$ be a source:*

$$0 \le H(\mathcal{S}) \le \log_2 n.$$

Proof. We apply Gibbs' lemma to the distribution $(q_1, \dots, q_n) = (\frac{1}{n}, \dots, \frac{1}{n})$, we have $H(\mathcal{S}) \le \log_2 n \sum_{i=1}^{n} p_i \le \log_2 n$ for any source S. Finally, positivity is obvious noticing that the probabilities p_i are less than 1. □

We can notice that for a uniform distribution, entropy reaches its maximum. It decreases when the distribution differs from the uniform distribution. This is why it is called a "measure of disorder," assuming that the greatest disorder is reached by a uniform distribution.

Exercise 1.8 *What is the entropy of a source where the character* 1 *has probability* 0.1 *and the character* 0 *has probability* 0.9*? Why will a small entropy be good for compression?* *Solution on page 282.*

1.2.3.3 Joint Entropies, Conditional Entropies The definition of entropy can be easily extended to several sources. Let $\mathcal{S}_1 = (\mathcal{S}_1, P_1)$ and $\mathcal{S}_2 = (\mathcal{S}_2, P_2)$ be two sources without memory, whose events are not necessarily independent. Let $\mathcal{S}_1 = (s_{11}, \dots, s_{1n})$, $P_1 = (p_i)_{i=1\dots n}$, $\mathcal{S}_2 = (s_{21}, \dots, s_{2m})$ and $P_2 = (p_j)_{j=1\dots m}$; then $p_{i,j} = P(\mathcal{S}_1 = s_{1i} \cap \mathcal{S}_2 = s_{2j})$ is the probability of joint occurrence of s_{1i} and s_{2j} and $p_{i|j} = P(\mathcal{S}_1 = s_{1i} | \mathcal{S}_2 = s_{2j})$ is the probability of the conditional occurrence of s_{1i} and s_{2j}.

We call *joint entropy* of $\mathcal{S}_1$ and $\mathcal{S}_2$ the quantity

$$H(\mathcal{S}_1, \mathcal{S}_2) = -\sum_{i=1}^{n}\sum_{j=1}^{m} p_{i,j}\log_2(p_{i,j}).$$

For example, if the sources $\mathcal{S}_1$ and $\mathcal{S}_2$ are independent, then $p_{i,j} = p_i p_j$ for all i, j. In this case, we can easily show that $H(\mathcal{S}_1, \mathcal{S}_2) = H(\mathcal{S}_1) + H(\mathcal{S}_2)$.

On the contrary, if the events of $\mathcal{S}_1$ and $\mathcal{S}_2$ are not independent, we might want to know the amount of information in one source, knowing one event of the other source. Thus, we compute the *conditional entropy* of $\mathcal{S}_1$ relative to the value of $\mathcal{S}_2$, given by

$$H(\mathcal{S}_1|\mathcal{S}_2 = s_{2j}) = -\sum_{i=1}^{n} p_{i|j}\log_2(p_{i|j}).$$

Finally, we extend this notion to a conditional entropy of $\mathcal{S}_1$ knowing $\mathcal{S}_2$, which is the amount of information remaining in $\mathcal{S}_1$ if the law of $\mathcal{S}_2$ is known:

$$H(\mathcal{S}_1|\mathcal{S}_2) = \sum_{j=1}^{m} p_j H(\mathcal{S}_1|\mathcal{S}_2 = s_{2j}) = \sum_{i,j} p_{i,j}\log_2\left(\frac{p_j}{p_{i,j}}\right).$$

This notion is crucial in cryptography. Indeed, it is very important for all ciphertexts to have a strong entropy in order to prevent the signs of organization in a message from giving some information on the way it was encrypted. Moreover, it is also important for entropy to remain strong even if a third party manages to obtain some important information concerning the plaintext. For instance, if some mails are transmitted, then they share common patterns in Headers; yet this knowledge should typically not provide information on the secret key used to encrypt the mails.

Then, we have the simple – but important – following relations:

$$H(\mathcal{S}_1) \geq H(\mathcal{S}_1|\mathcal{S}_2)$$

with equality if and only if $\mathcal{S}_1$ and $\mathcal{S}_2$ are independent; and also

$$H(\mathcal{S}_1, \mathcal{S}_2) = H(\mathcal{S}_2) + H(\mathcal{S}_1|\mathcal{S}_2).$$

However, the entropy of a source without memory does not capture all the order or the disorder included in a message on its own. For example, the messages "1 2 3 4 5 6 1 2 3 4 5 6" and "3 1 4 6 4 6 2 1 3 5 2 5" have the same entropy; yet the first one is sufficiently ordered to be described by a formula, such as "1…6 1…6," which is probably not the case for the second one. To consider this kind of organization, we make use of the extensions of a source.

1.2.3.4 Extension of a Source Let $\mathcal{S}$ be a Source without memory. The *kth extension* $\mathcal{S}^k$ of $\mathcal{S}$ is the couple $(S^k, \mathcal{P}^k)$, where S^k is the set of all words of length k over S

and $\mathcal{P}^k$ the probability distribution defined as follows: for a word $s = s_{i_1} \ldots s_{i_k} \in S^k$, then $P^k(s) = P(s_{i_1} \ldots s_{i_k}) = p_{i_1} \cdots p_{i_k}$.

Example 1.1 If $S = (s_1, s_2)$, $\mathcal{P} = (\frac{1}{4}, \frac{3}{4})$, then $S^2 = (s_1s_1, s_1s_2, s_2s_1, s_2s_2)$ and $P^2 = (\frac{1}{16}, \frac{3}{16}, \frac{3}{16}, \frac{9}{16})$.

If $\mathcal{S}$ is a Markov source, we define S^k in the same way, and for a word $s = s_{i_1} \ldots s_{i_k} \in S^k$, then $P^k(s) = p_{i_1} p_{i_2 i_1} \cdots p_{i_k i_{k-1}}$.

Property 2 *Let $\mathcal{S}$ be a source, and $\mathcal{S}^k$ its kth extension, then*

$$H(\mathcal{S}^k) = kH(\mathcal{S})\,.$$

In other words this property stresses the fact that the amount of information of a source extended to k characters is exactly k times the amount of information of the original source. This seems completely natural.

However, this does not apply to the amount of information of a *message* (a file for instance) "extended" to blocks of k characters. More precisely, it is possible to enumerate the occurrences of the characters in a message to compute their distribution and then the entropy of a source that would have the same probabilistic characteristics. Indeed, this is used to compress files as it will be seen in Chapter 2. Now if the message is "extended" in the sense that groups of k characters are formed and the occurrences of each group is computed to get their distribution, then the entropy of this "message extension" is necessarily lower than k times the entropy corresponding to the original message as shown in the following.

Property 3 *Let $\mathcal{M}$ be a message of size n, $\mathcal{S}_{\mathcal{M}^k}$ be the source whose probabilities correspond to the occurrences of the successive k-tuples of $\mathcal{M}$ and $\mathcal{S}^k_{\mathcal{M}}$ be the kth extension of the induced source $\mathcal{S}_{\mathcal{M}}$. Then*

$$H(\mathcal{S}_{\mathcal{M}^k}) \leq H(\mathcal{S}^k_{\mathcal{M}})\,.$$

Proof. We give some details for $k = 2$. Let (q_i) be the probabilities of the induced source $\mathcal{S}_{\mathcal{M}}$, $(q_{i,j} = q_i \cdot q_j)$ those of the second extension $\mathcal{S}^2_{\mathcal{M}}$ and $(p_{i,j})$ those of an source induced by the successive pairs of elements of M, $\mathcal{S}_{\mathcal{M}^2}$. Gibbs lemma, page 17, applied to $p_{i,j}$ and $q_{i,j}$ shows that $\sum_{i,j} p_{i,j} \log_2 \left(\frac{q_i q_j}{p_{i,j}}\right) \leq 0$. This is also

$$H(\mathcal{S}_{\mathcal{M}^2}) \leq -\sum_{i,j} p_{i,j} \log_2(q_i q_j). \tag{1.1}$$

Now, denote by n_i the number of occurrences of the i symbol in $\mathcal{M}$. With $n = |\mathcal{M}|$, we have $q_i = n_i/n$. Also denote by $n_{i,j}$ the number of occurrences of the pair (i, j) in $\mathcal{M}^2$. Then $|\mathcal{M}^2| = n/2$ and $p_{i,j} = 2n_{i,j}/n$. We also have $n_i = \sum_j n_{i,j} + \sum_k n_{k,i}$ so that $2q_i = \sum_j p_{i,j} + \sum_k p_{k,i}$.

Therefore, the right-hand side of Inequation (1.1) can be rewritten as follows:

$$\begin{aligned}
\sum_{i,j} p_{i,j}\log_2(q_i q_j) &= \sum_{i,j} p_{i,j}\log_2(q_i) + \sum_{i,j} p_{i,j}\log_2(q_j)\\
&= \sum_i \log_2(q_i)\left(\sum_j p_{i,j}\right) + \sum_j \log_2(q_j)\left(\sum_i p_{i,j}\right)\\
&= \sum_i \log_2(q_i)\left(\sum_j p_{i,j} + \sum_k p_{k,i}\right) \qquad (1.2)\\
&= \sum_i \log_2(q_i)\left(2q_i\right)\\
&= -2H(\mathcal{S}_{\mathcal{M}})\\
&= -H(\mathcal{S}_{\mathcal{M}}^2).
\end{aligned}$$

This proves that Inequation (1.1) is actually

$$H(\mathcal{S}_{\mathcal{M}^2}) \le H(\mathcal{S}_{\mathcal{M}}^2).$$

This construction generalizes to any k, in the same manner, by enumeration of all the k-tuples. □

The following example illustrates both situations.

Example 1.2 The messages "1 2 3 4 5 6 1 2 3 4 5 6" and "3 1 4 6 4 6 2 1 3 5 2 5" have the same entropy taking the first extension of the source: six characters of probability one $\frac{1}{6}$ each, giving an entropy of $\sum_{i=1}^{6}\frac{1}{6}\log_2(6) = \log_2(6) \approx 2.585$. With the second extension of the source $(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6})$, we obtain 36 possible groups of probability $\frac{1}{36}$ each and the entropy conforms to Property 2: $\log_2(36) = \log_2(6^2) = 2\log_2(6)$. However, for example, when regrouping the messages in blocks of two characters, we have

- (12)(34)(56)(12)(34)(56): three different couples of probability $\frac{1}{3}$ each corresponding to an entropy of $\log_2(3) \approx 1.585$.
- All the same, the sequence (31)(46)(46)(21)(35)(25) gives five different couples and is of entropy $\frac{1}{6}\log_2(6) + \frac{2}{6}\log_2\left(\frac{6}{2}\right) + \frac{1}{6}\log_2(6) + \frac{1}{6}\log_2(6) + \frac{1}{6}\log_2(6) \approx 2.252$.

In both cases, the entropy obtained is definitely lower than twice the entropy of the original message. We will make this statement precise in Property 4.

Property 4 *Let $\mathcal{M}$ be a message of size n and let $\mathcal{S}_{\mathcal{M}^k}$ be the source whose probabilities correspond to the occurrences of the successive k-tuples of $\mathcal{M}$. Then*

$$H(\mathcal{S}_{\mathcal{M}^k}) \le log_2\left(\left\lceil \frac{n}{k} \right\rceil\right).$$

Proof. There are $\left\lceil \frac{n}{k} \right\rceil$ k-tuples in the message $\mathcal{M}$. Besides, entropy is maximal for the greatest number of distinct possible k-tuples with the same number of occurrences. In this case, the corresponding source would contain at most $\left\lceil \frac{n}{k} \right\rceil$ distinct characters of probability of occurrence $\frac{1}{\left\lceil \frac{n}{k} \right\rceil}$. Thus, the entropy is $log_2 \left(\left\lceil \frac{n}{k} \right\rceil \right)$. □

This leads us to the problem of randomness and its generation. A sequence of numbers randomly generated should meet harsh criteria – in particular, it should have a strong entropy. The sequence "1 2 3 4 5 6 1 2 3 4 5 6" would not be acceptable as one can easily notice some kind of organization. The sequence "3 1 4 6 4 6 2 1 3 5 2 5" would be more satisfying – having a higher entropy when considering successive pairs of characters. We detail the random number generators in Section 1.3.7.

1.2.4 Steganography and Watermarking

Entropy is a powerful tool to model the information in a code. For instance, it can be used to detect steganography by a study of the quantity of information contained in a device.

Steganography is the art of covering information. The knowledge of the mere existence of some covert information could then be sufficient to discover this information.

Steganographic devices include invisible ink, microdot in images, Digital Right Management (DRM), information encoding in white spaces of a plaintext, and so on.

Nowadays, steganography is quite often combined with cryptography in order to not only conceal the existence of information but also keep its secrecy even if its existence is revealed. It is also combined with error-correcting codes. Indeed, even if the resulting *stegotext* is modified and some parts of the information are altered, the information remains accessible if sufficiently many bits remain unchanged by the media modification.

Digital watermarking is a variant of steganography, which is used to conceal some information into digital media such as images, audio, or video.

We distinguish two major kinds of watermarking:

- Fragile watermarking is very close to classical steganography, it is usually invisible and used to detect any modification of the stream. For instance, secure paper money often encloses fragile watermarks that disappear after photocopy.
- Robust watermarking might be visible and should at least partially resist to simple modifications of the source as lossy compression. This is what is required for instance for Digital Right Management.

It is difficult to hide a large quantity of information into a media without begin detectable. Indeed, consider a covering media M, some information X and a stegotext S where the information is embedded into the covertext. As the stegotext should not be very different from the covertext to be undetected, the quantity of information in the stegotext should be very close to that of the covertext added to that of the

embedded information: $H(S) \approx H(M) + H(X)$. Therefore, a classical steganalysis is to compute the entropy of a suspected media and compare it to classical values of unmarked ones. If the obtained entropy is significantly higher than the average, then it can mean that some additional information if carried by this media. In other words, to remain undetected steganography must use a small quantity of information into a large media.

Exercise 1.9 `We  have hidden  a  number  in the spacing of this text.  Can you find  it?` *Solution on page 283.*

We will see an example of watermarking of digital images using the JPEG format in Section 2.5.3.

1.2.5 Perfect Secrecy

Using the concept of entropy, it is also now possible to make precise the concept of unconditional security (also called perfect secrecy).

An encryption scheme is said unconditionally secure or perfect if the ciphertext C does not give any information on the initial plaintext M; hence, the entropy of M knowing C is equal to the entropy of M:

$$H(M|C) = H(M)$$

In other words, unconditional security means that M and C are statistically independent.

Exercise 1.10 (Perfect Secrecy of the One-Time-Pad Encryption Scheme)

1. *We consider a secret code wherein the key K is generated randomly for each message M. Prove that $H(M|K) = H(M)$.*
2. *Using conditional entropies and the definition of a Vernam code, prove that in a Vernam code ($C = M \oplus K$), we always have $H(M,K,C) = H(M,C) = H(M,K) = H(C,K)$; deduce the relations between the conditional entropies of M, C, and K.*
3. *Prove that the OTP system is a perfect encryption scheme.*

Solution on page 283.

1.2.6 Perfect Secrecy in Practice and Kerckhoffs' Principles

Now, we have an encryption method (the one-time-pad) and the proof of its security. It is the only known method to be proved unconditionally secure. Yet in order to use this method in practice, we should be able to generate random keys of the same size of the message, which is far from being trivial. One solution is to rely on PRNG (Section 1.3.7) to create the successive bits of the key that are combined with the bit stream of the plaintext. It leads to a bitwise encryption (stream cipher) as illustrated in Figure 1.2.

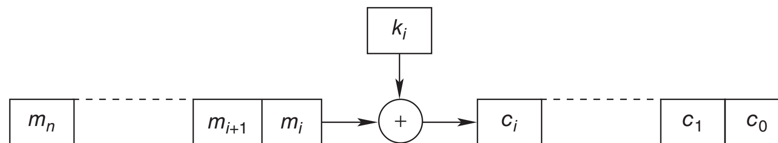

Figure 1.2 Bitwise encryption (Stream cipher)

However, key exchange protocols between the sender and the recipient remain problematic as the keys cannot be used twice and are generally big (as big as the message).

Exercise 1.11 (Imperfection Mesure and Unicity Distance) *We consider messages M written in English, randomly chosen keys K and ciphertexts C, produced by an imperfect code over the* 26 *letters alphabet. A length l string is denoted by* $X_1 \dots X_l$ *and we suppose that any key is chosen uniformly from the space K. The* unicity distance *of a code is the average minimal length d (number of letters) of a ciphertext required to find the key. In other words d satisfies* $H(K|C_1 \dots C_d) = 0$*: if you know d letters of a ciphertext, you have, at least theoretically, enough information to recover the used key.*

1. *For a given symmetric cipher, knowing the key and the cleartext is equivalent to knowing the key and the ciphertext:* $H(K, M) = H(K, C)$*. From this and the choice of the key, deduce a relation between* $H(K)$, $H(M_1 \dots M_d)$, $H(C_1 \dots C_d)$ *with the unicity distance d.*
2. *Using Table 1.1, the entropy of this book is roughly* 4.19 *per character. What is therefore the entropy of a similar plaintext of d letters?*
3. *What is the entropy of a randomly chosen string over the alphabet with* 26 *letters ? Use these entropies to bound* $H(C_1 \dots C_d)$ *for a good cipher.*
4. *Overall, deduce a lower bound for the unicity distance depending on the information contained in the key.*
5. *If the cipher consists in choosing a single permutation of the alphabet, deduce the average minimal number of letters required to decipher a message encoded with this permutation.* *Solution on page 283.*

In general, coding schemes are not perfect, and theoretical proofs of security are rare. Empirical principles often state the informal properties we can expect from a cryptosystem. Auguste Kerckhoffs formalized the first and most famous ones in 1883 in his article “La cryptographie militaire,” which was published in the “Journal des Sciences Militaires.” Actually, his article contains six principles, which are now known as “*Kerckhoffs’ principles*.” We will summarize three of them, the most useful nowadays:

1. Security depends more on the secrecy of the key than on the secrecy of the algorithm.

For a long time, the security of a cryptosystem concerned the algorithms that were used in this system. For instance, this was the case of Caesar encryption (Section 1.1) or of the ADFVGX code, used by German forces during World War I. Security is illusory because, sooner or later, details of the algorithm are going to be known and its potential weaknesses will be exploited. In addition, it is easier to change a key if it is compromised than to change the whole cryptosystem. Moreover, you can believe in the resistance of public cryptosystems as they are continuously attacked: selection is rough, therefore if a cryptosystem, whose internal mechanisms are freely available, still resists the continuous attacks performed by many cryptanalysts, then there are more chances that the system is really secure.

2. Decryption without the key must be impossible (in reasonable time);
3. Finding the key from a plaintext and its ciphertext must be impossible (in reasonable time).

Therefore, **one must always assume that the attacker knows all the details of the cryptosystem**. Although these principles have been known for a long time, many companies continue to ignore them (voluntarily or not). Among the most media-related recent examples, one may notice the A5/0 and A5/1 encryption algorithms that are used in Global System for Mobile Communications (GSM) and most of all Content Scrambling System (CSS) software for the protection against DVD copying. The latter algorithms were introduced in 1996 and hacked within weeks, despite the secrecy surrounding the encryption algorithm.

Following the Kerckhoffs principles, it is possible to devise codes that are not perfect but tend toward this property. In Section 1.3, we, for instance, see how to build codes which use a single (short) key and make the exchange protocol less time consuming than the OTP. The idea is to split the message into blocks such that each of them is encrypted separately.

1.3 BLOCK CIPHERS, ALGEBRA, AND ARITHMETIC

In this section, we introduce most of the mathematical background used in coding theory and applications. It contains the bases of algebra, and efficient ways to compute in algebraic structures. Some arithmetics is also useful to construct large finite structures. All these notions are widely used in the book and become necessary as soon as block coding methods are envisaged.

Today, the Vernam cipher is the only symmetric cryptographic algorithm that has been proved unconditionally secure. Thus, all other known systems are theoretically breakable.

For these systems, we use *almost secure* encryption schemes: the knowledge of the ciphertext (or the knowledge of some couples plaintext/ciphertext) does not enable to recover the secret key or the plaintext *in humanly reasonable time* (see the orders of magnitude and computing limits in Section 1.1.7).

For instance, we can decide to choose a unique key and to reuse it in order to avoid too frequent key exchange protocols. This implies that we have to split the source messages into blocks of some size, depending on the size of the key. Block cipher is also a standard, which is widely used in error detection and correction.

This is also the principle of one of the most famous codes – the ASCII code ("*American Standard Code for Information Interchange*") – which is a numerical representation of the letters and signs of the Latin alphabet. In ASCII code, the source alphabet is the Latin alphabet, and the code alphabet is $V = \{0, 1\}$. The set of *codewords* is the set of all the words of size 8 over V:

$$C = \{00000000,\ 00000001,\ \dots,\ 11111111\}.$$

Each one of the $2^8 = 256$ characters (uppercases, lowercases, special characters, and control characters) is represented with a word of size 8 over V according to an encoding function. The following Table 1.3 gives an extract of this function.

For example, the ASCII code of the message: *A KEY*, is the string: 0100000100100000010010110100010101011001.

1.3.1 Blocks and Chaining Modes from CBC to CTR

It is possible to encode independently each block of a message with the same algorithm. This is called Electronic Code Book (ECB) cipher mode. More generally, the independence of encryption between the blocks is not required and the several ways of combining the blocks are called *encryption modes*.

ECB mode: Electronic Code Book. In this mode, the message M is split into blocks m_i of constant size. Each block is encrypted separately with a function E_k (the key k being a parameter of the function) as illustrated in Figure 1.3.

$$c_i = E_k(m_i) \tag{1.3}$$

Thus, a given block of the message m_i will always be encrypted in the same way. This encryption mode is the easiest one but does not provide any security and is normally never used in cryptography.

TABLE 1.3 Extract of the ASCII Code

A	01000001	J	01001010	S	01010011
B	01000010	K	01001011	T	01010100
C	01000011	L	01001100	U	01010101
D	01000100	M	01001101	V	01010110
E	01000101	N	01001110	W	01010111
F	01000110	O	01001111	X	01011000
G	01000111	P	01010000	Y	01011001
H	01001000	Q	01010001	Z	01011010
I	01001001	R	01010010	Space	00100000

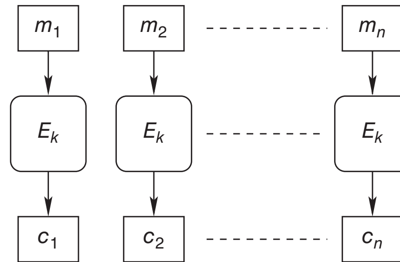

Figure 1.3 Block ciphers: ECB mode

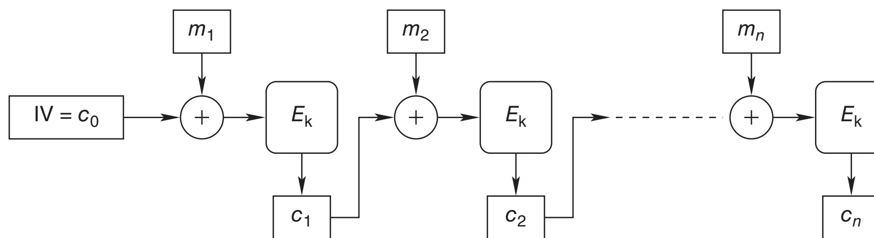

Figure 1.4 Block ciphers: CBC mode

CBC mode: Cipher Block Chaining. The CBC mode was introduced to avoid encrypting a block in the same way in two different messages. We add some initial value $IV = c_0$, possibly generated randomly. Each block m_i is encoded by an *XOR* operation with the previous cipherblock c_{i-1} before being encrypted. Figure 1.4 illustrates this mode.

$$c_i = E_k(m_i \oplus c_{i-1}) \tag{1.4}$$

This is the most widely used encryption mode. Decryption uses the inverse of the encoding function $D_k = E_k^{-1}$: $m_i = c_{i-1} \oplus D_k(c_i)$.

CFB mode: Cipher FeedBack. To avoid using the inverse function for decryption, it is possible to perform an *XOR* after the encryption. This is the principle of the CFB mode, which is illustrated in Figure 1.5.

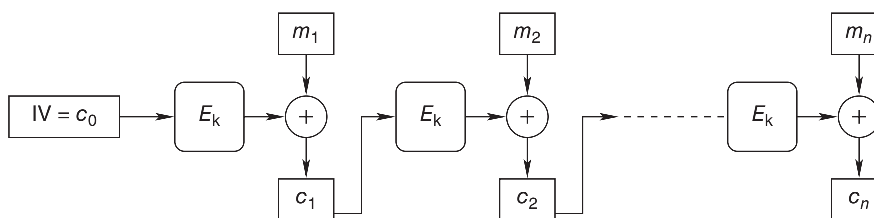

Figure 1.5 Block ciphers: CFB mode

$$c_i = m_i \oplus E_k(c_{i-1}) \tag{1.5}$$

The benefit of this mode is to avoid implementing the function D_k for decryption: $m_i = c_i \oplus E_k(c_{i-1})$. Thus, this mode is less secure than the CBC mode and is used in network encryption for example.

OFB mode: Output FeedBack. OFB is a variant of the previous mode and it provides symmetric encryption and decryption. Figure 1.6 illustrates this scheme.

$$z_0 = c_0 \; ; \; z_i = E_k(z_{i-1}) \; ; \; c_i = m_i \oplus z_i \tag{1.6}$$

Decryption is performed by $z_i = E_k(z_{i-1}) \; ; \; m_i = c_i \oplus z_i$. This mode is useful when one needs to minimize the number of embedded circuits, especially for communications in spacecrafts.

CTR mode: Counter Mode Encryption. The CTR mode is also completely symmetric, but encryption can be perfomed in parallel. It uses the encryption of a counter of initial value T (Figure 1.7):

$$c_i = m_i \oplus E_k(T + i) \tag{1.7}$$

Decryption is performed in the same way: $m_i = c_i \oplus E_k(T + i)$. The advantage of such a mode is that the several computations are independent, as in the ECB mode, but a block is also never encrypted twice in the same way *a priori*.

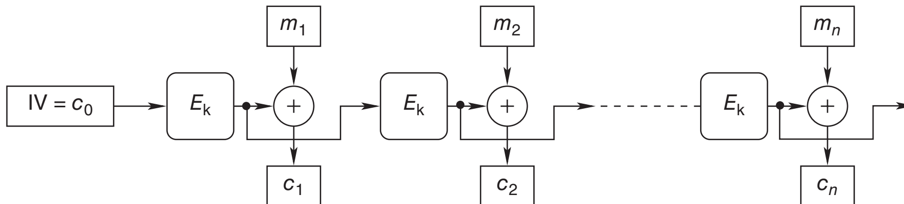

Figure 1.6 Block ciphers: OFB mode

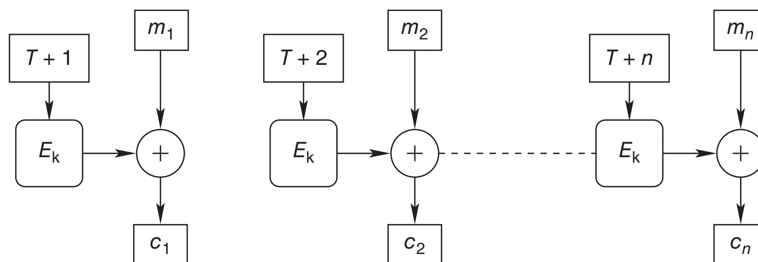

Figure 1.7 Block ciphers: CTR mode

Exercise 1.12 *A message M is split into n blocks $M_1, \ldots, M_n$ that are encrypted into $C = C_1, \ldots C_n$ using an encryption mode. Bob receives the blocks C_i, but unfortunately, he does not know that one (and only one) of the blocks was incorrectly transmitted (for example, some 0s became 1s and some 1s became 0s during the transmission of block C_1). Show that the number of miss-decrypted blocks is equal to 1 if ECB, OFB or CTR modes were used and equal to 2 if CBC, or CFB modes were used.*
Solution on page 283.

1.3.2 Algebraic Structure of Codewords

Developing block ciphers implies that we have to be able to perform operations and computations over blocks. For example, the $\oplus$ operation over a block of bits is a bitwise addition modulo 2 of two vectors. Furthermore, as encoding functions have to be reversible, we need structures for which we can easily compute the inverse of a block. In order to perform these computations using solid algebraic bases, let us recall some fundamental structures. All along the book, $\mathbb{Z}$ denotes the set of integers, and $\mathbb{N}$ denotes the set of nonnegative integers.

1.3.2.1 Groups A *group* $(G, *)$ is a set equipped with an internal binary operator satisfying the following properties:

1. $*$ is associative: for all $a, b, c \in G$, $a * (b * c) = (a * b) * c$.
2. There exists a neutral element $e \in G$, such that for all $a \in G$, one has $a * e = e * a = a$.
3. Each element has an inverse: for all $a \in G$, there exists $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = e$.

Moreover, if the law is commutative ($a * b = b * a$ for all $a, b \in G$), G is called *abelian*.

A subset H of G is called a *subgroup* of G if the operations of G restricted to H give a group structure to H. For an element a of a group G, we denote by a^n the repetition of the law $*$, $a * \cdots * a$, performed on n terms equal to a for all $n \in \mathbb{N}^*$. Moreover, one has $a^0 = e$ and $a^{-n} = (a^{-1})^n$ for all $n \in \mathbb{N}^*$.

If an element $g \in G$ is such that for all $a \in G$, there exists $i \in \mathbb{Z}$, such that $a = g^i$, then g is a *generator* of the group $(G, *)$ or a *primitive root*. A group is called *cyclic* if it has generator.

For example, for a given n fixed, the set of integers $\mathbb{Z}/n\mathbb{Z} = \{0, 1, \ldots, n-1\}$, equipped with the law of addition modulo n is a cyclic group generated by 1; if $n = 7$ and if we choose the multiplication modulo 7 as a law of composition, the set $\{1, \ldots, 6\}$ is a cyclic group generated by 3. Indeed $1 = 3^0$, $2 = 3^2 = 9$, $3 = 3^1$, $4 = 3^4$, $5 = 3^5$, $6 = 3^3$.

Let $(G, *)$ be a group and $a \in G$. The set $\{a^i, i \in \mathbb{Z}\}$ is a subgroup of G, denoted by $< a >$ or G_a. If this subgroup is finite, its cardinal number is the *order* of a.

Property 5 (Lagrange) *If G is a finite group, the cardinal number of any subgroup of G divides the cardinal number of G. In particular, the order of any elements divides the cardinal number of G.*

Proof. Let H be a subgroup of a finite group G. $|H|$ is the cardinal number of H and consider the sets $aH = \{ah, h \in H\}$ for any $a \in G$. First of all, all the sets aH have the same cardinal number: if $ah_1 = ah_2$, then since a is invertible $h_1 = h_2$, so that $|aH| = |H|$. Then these sets form a partition of G: indeed take aH and bH with $a \neq b$ and suppose that there exist an element x in their intersection, that is, $x = ah_1 = bh_2$. Then for any element $u \in aH$, $u = ah_u = b(h_2 h_1^{-1} h_u)$. But as H is a subgroup, $h_2 h_1^{-1} h_u \in H$ and thus $u \in bH$. This proves that aH is included in bH. With the reverse argument, one can prove also that bH is included in aH. Therefore, two sets aH and bH are either equal or disjoint. Finally, any element x in G is in xH. Now, as the sets aH form a partition of G and they are all of cardinal number $|H|$, the cardinal order of G is a multiple of $|H|$. □

Theorem 1 (Lagrange) *In a finite abelian group $(G, *)$ of cardinal number n, for all $a \in G : a^n = e$.*

Proof. Let a be any element in G. The set of the multiples of a, $G_a = \{y = ax$ for $x \in G\}$ is equal to G. Indeed, as a is invertible, for all $y \in G$, we can define $x = a^{-1}y$. Reciprocally, if a and x are elements of G a group, then so is their product ($(ax)^{-1} = x^{-1}a^{-1}$). Hence, the two sets being equal, the products of all their respective elements are also equal:

$$\prod_{x \in G} x = \prod_{y \in G_a} y = \prod_{x \in G} ax.$$

Yet, as multiplication is commutative in an abelian group, we can then extract a from the product. Moreover, there are n elements in G and we thus obtain the following formula:

$$\prod_{x \in G} x = a^n \prod_{x \in G} x.$$

In order to conclude, we use the fact that – all elements in G being invertible – so is their product and we can simplify the previous formula: $e = a^n$. □

1.3.2.2 Rings A *ring* $(A, +, \times)$ is a set equipped with two internal binary operators satisfying the following properties:

1. $(A, +)$ is an abelian group.
2. $\times$ is associative: for all $a, b, c \in A$, $a \times (b \times c) = (a \times b) \times c$.
3. $\times$ is distributive over $+$: for all $a, b, c \in A$, $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$.

Moreover, if there exists a neutral element for $\times$ in A, A is called *unitary*. This neutral element is noted 1_A, or simply 1 if it is not ambiguous from the context. If $\times$

is commutative, A is called *commutative*. All elements in A have an *opposite*, namely their inverse for the law $+$. However, they do not necessarily have an inverse for the law $\times$. The *set of invertible elements* for the law $\times$ is denoted by A^*.

For an element a in a ring A, one denotes by $n \cdot a$ (or more simply na) the sum $a + \cdots + a$ of n terms equal to a for all $n \in \mathbb{N}^*$.

If the set $\{k \in \mathbb{N}^* \,:\, k \cdot 1 = 0\}$ is not empty, the smallest element of this set is called the *characteristic* of the ring. Otherwise, the ring is said to be of characteristic 0. For example, $(\mathbb{Z}, +, \times)$ is a unitary, commutative ring of characteristic 0.

Two rings $(A, +_A, \times_A)$ and $(B, +_B, \times_B)$ are *isomorphic* if there exists a bijection $f \,:\, A \longrightarrow B$ satisfying for all x and y in A:

$$f(x +_A y) = f(x) +_B f(y) \quad \text{and} \quad f(x \times_A y) = f(x) \times_B f(y). \tag{1.8}$$

If E is any set and $(A, +, \times)$ is a ring such that there exists a bijection f from E to A, then E can be equipped with a ring structure:

$$x +_E y = f^{-1}(f(x) + f(y)) \quad \text{et} \quad x \times_E y = f^{-1}(f(x) \times f(y)). \tag{1.9}$$

The ring $(E, +_E, \times_E)$ defined as such is obviously isomorphic to A. If two rings are isomorphic, one ring can be identified with the other.

An *ideal* I is a subgroup of a ring A for the law $+$ and "absorbing" for the law $\times$: for $g \in I$, the product $a \times g$ remains in I for any element a of the ring A. For all $x \in A$, the set $Ax = \{ax;\ a \in A\}$ is an ideal of A, which is *generated* by x. An ideal I of A is called *principal* if there exists a generator x (such that $I = Ax$). A ring A is *principal* if an only if any ideal of A is principal.

A *Euclidean function* ν is a mapping of all nonzero elements of a unitary ring to $\mathbb{N}$. A unitary ring with a Euclidean function is *Euclidean* if it has the property that for every couple of elements a and b of the ring, there exist q and r such that $a = bq + r$ and $\nu(r) < \nu(b)$. This operation is the *Euclidean division*, and the numbers q and r are, respectively, called the Euclidean quotient and the remainder

and are denoted by $q = a$ div b and $r = a \mod b$ (for *a modulo b*). Any Euclidean ring is principal. This implies that there exists a Greatest Common Divisor (GCD) for all couples of elements (a, b). Any generator of the ideal $Aa + Ab$ is a *gcd*.

For example, the ring $\mathbb{Z}$ with the absolute value as Euclidean function, it is a Euclidean ring. The following famous theorem of Bézout extends the properties of Euclidean rings. It is true for any Euclidean ring, and its proof in $\mathbb{Z}$ will follow from the Euclidean algorithm page 37.

Theorem 2 (Bézout) *Let a and b be two nonzero elements of a Euclidean ring A, and let d be their GCD. There exist two elements x and y, called the Bézout's numbers, such that $\nu(x) \leq \nu(b)$ and $\nu(y) \leq \nu(a)$ satisfying*

$$ax + by = d.$$

The modulo operation allows to define a ring on $\mathbb{Z}/n\mathbb{Z}$, the set of nonnegative integers strictly inferior to n, for $n \in \mathbb{N} \setminus \{0, 1\}$. The set $\mathbb{Z}/n\mathbb{Z}$ equipped with the

addition and the multiplication modulo n [that is, $a +_{\mathbb{Z}/n\mathbb{Z}} b = (a +_{\mathbb{Z}} b) \mod n$ and $a \times_{\mathbb{Z}/n\mathbb{Z}} b = (a \times_{\mathbb{Z}} b) \mod n$] is a (finite) ring. It is widely used in coding.

Exercise 1.13 *Bézout's theorem is very useful to prove properties in number theory. In particular, use it to prove the famous Gauss's lemma stated as follows:*

Lemma 2 (Gauss) *If an integer number a divides the product of two integers b and c, and if a and b are coprime, then a divides c.*

Solution on page 284.

1.3.2.3 Fields A *field* $(A, +, \times)$ is a set equipped with two internal binary operators satisfying the following properties:

1. $(A, +, \times)$ is an unitary ring.
2. $(A \setminus \{0\}, \times)$ is a group.

If $(A, +, \times)$ is commutative, the field is *commutative*; the inverse (or opposite) of x with regard to the law $+$ is denoted by $-x$; the inverse of x with regard to the law $\times$ is denoted by x^{-1}. The *characteristic* of a field is its characteristic when considered as a ring.

For instance, $(\mathbb{Q}, +, \times)$ is a commutative field of characteristic 0.

As all the rings and fields we are dealing with are commutative, thereafter the word ring (respectively field) will refer to a unitary commutative ring (respectively a commutative field).

Two fields are *isomorphic* if they are isomorphic when considered as rings.

A subset W of a field V is called a *subfield* of V if the operations of V restricted to W give a field structure to W.

The standard notation is used for classical fields in this book: $\mathbb{Q}$ denotes the field of rational numbers, and $\mathbb{R}$ denotes the field of real numbers.

If p is a prime number, the ring $\mathbb{Z}/p\mathbb{Z}$ is a field of characteristic p.

Indeed, with Bézout's theorem (see page 32), we have for all couples of integers a and b, there exists two integers x and y such that

$$ax + by = \gcd(a, b).$$

If p is prime and a is a nonzero element of $\mathbb{Z}/p\mathbb{Z}$, this identity applied to a and p gives $ax + bp = 1$, hence $ax = 1 \mod p$. Thus, a is invertible and x is its inverse. This field is denoted by $\mathbb{F}_p$.

The field of rational numbers $\mathbb{Q}$ and the fields $\mathbb{F}_p$ are called *prime fields*.

1.3.2.4 Vector Spaces and Linear Algebra Rudiments of linear algebra are necessary for the reading of the major part of this book. Without any explicative pretention, here we define useful concepts and we introduce the main notation. A set $\mathbb{E}$ is a *vector space* over a field V if it has one internal law of composition $+$ and an external law of composition ".," such that

1. $(\mathbb{E}, +)$ is a commutative group.
2. For all $u \in \mathbb{E}$, $1_V.u = u$.
3. For all $\lambda, \mu \in V$, and $u \in \mathbb{E}$, $\lambda.(\mu.u) = (\lambda \times \mu).u$.
4. For all $\lambda, \mu \in V$, and $u \in \mathbb{E}$, $\lambda.u + \mu.u = (\lambda + \mu).u$.
5. For all $\lambda \in V$ and $u, v \in \mathbb{E}$, $\lambda.(u + v) = \lambda.u + \lambda.v$.

An element of a vector space is called a *vector*, and the elements of the field V are called *scalars*. The set $\{0, 1\}$ equipped with the addition $\oplus$ and the multiplication $\wedge$ is a commutative field denoted by $\mathbb{F}_2$. Thus, the set of bit tables of size n can be equipped with a vector space structure. The set of codewords is then $\mathbb{F}_2^n$.

A set of vectors $x_1, \dots, x_n$ is an *independent* set if for all scalars $\lambda_1, \dots, \lambda_n$, $\sum_{i=1}^{n} \lambda_i x_i = 0$ implies $\lambda_1 = \cdots = \lambda_n = 0$.

The *dimension* of a vector space V, denoted by $\dim(V)$, is the cardinal number of the greatest set of independent vectors of V.

For example, if V is a field, V^n is a space of dimension n because the vectors $(1, 0, \dots, 0)$, $(0, 1, 0, \dots, 0)$, …, $(0, \dots, 0, 1)$ are independent.

A *linear mapping* is an application from one vector space to another, which preserves the laws of composition. Namely, if A and B are two vectors spaces, an application f is said to be linear if for all x, y in A, $f(x +_A y) = f(x) \times_B f(y)$ and for all $\lambda \in V$ and $x \in A$, $f(\lambda \cdot_A x) = \lambda \cdot_B f(x)$.

The *image* of a linear mapping f from a vector space $\mathbb{E}$ to a vector space $\mathbb{F}$, denoted by $\mathrm{Im}(f)$, is the set of vectors $y \in \mathbb{F}$ such that there exists $x \in \mathbb{E}$ with $f(x) = y$.

The *kernel* of a linear application f from a vector space $\mathbb{E}$ to a vector space $\mathbb{F}$, denoted by $\mathrm{Ker}(f)$, is the set of vectors $x \in \mathbb{E}$ such that $f(x) = 0$.

It is easy to verify that $\mathrm{Ker}(f)$ and $\mathrm{Im}(f)$ are vector spaces.

If the dimension of $\mathrm{Im}(f)$ is finite, this quantity is called the *rank* of the linear mapping and is denoted by $\mathrm{Rk}(f) = \dim(\mathrm{Im}(f))$. Moreover, if the dimension of $\mathbb{E}$ is also finite, then we have the following result: $\dim(\mathrm{Ker}(f)) + \dim(\mathrm{Im}(f)) = \dim(\mathbb{E})$.

A *matrix* M of size (m, n) is an element of the vector space $V^{m \times n}$, represented by a table of m horizontal lines (rows) of size n or n vertical lines (columns) of size m. The element that lies in the ith row and the jth column of the matrix is written as $M_{i,j}$. Multiplication of a vector x of size n with such as matrix gives a vector y of size m satisfying $y_i = \sum_{k=1}^{n} M_{i,k} x_k$ for all i from 1 to m; multiplication is written as $y = Mx$.

Each matrix M is associated to a linear application f from V^m to V^n, defined by $f(x) = Mx$. Reciprocally, any linear application can be written using a matrix. The coding processes studied throughout this book – mainly in Chapter 4 – often use linear applications, which enable us to illustrate the properties of these functions.

Depending on the chosen structure, codewords can be manipulated with additions and multiplications over integers or vectors. All these structures are very general and common in algebra. Codes are particular as they are finite sets. Finite groups and finite fields have some additional properties, which will be widely used throughout this work.

1.3.3 Bijective Encoding of a Block

Now that we have some structures, we are able to perform additions, multiplications, and Euclidean divisions on blocks. We can also compute their inverses. Here, we give some fundamental examples of computations that can be performed on sets with good algebraic structure. As blocks are of finite size, we will manipulate finite sets in this section.

1.3.3.1 Modular Inverse: Euclidean Algorithm Bézout's theorem (see page 32) guarantees the existence of Bézout numbers and thus the existence of the inverse of a number modulo a prime number in $\mathbb{Z}$. The Euclidean algorithm makes it possible to compute these coefficients efficiently.

In its fundamental version, the Euclidean algorithm computes the Greatest Common Divisor (GCD) of two integers according to the following principle: assuming that $a \geq b$,

$$\gcd(a, b) = \gcd(a - b, b) = \gcd(a - 2b, b) = \cdots = \gcd(a \mod b, b),$$

where $a \mod b$ are the remainder of the Euclidean division of a by b. Indeed, if a and b have a common divisor d, then $a - b, a - 2b, \ldots$ are also divisible by d.

The recursive principle appears:

Algorithm 1.3 GCD: Euclidean Algorithm

Input Two integers a and b, $a \geq b$.
Output $\gcd(a, b)$
1: **If** $b = 0$ **then**
2: **return** a;
3: **else**
4: Compute recursively $\gcd(b, a \mod b)$ and return the result
5: **End If**

Example 1.3 (The gcd **of 522 and 453)**
We compute

$$\begin{aligned}\gcd(522, 453) &= \gcd(453, 522 \mod 453 = 69) &&= \gcd(69, 453 \mod 69 = 39)\\ &= \gcd(39, 69 \mod 39 = 30) &&= \gcd(30, 39 \mod 30 = 9)\\ &= \gcd(9, 30 \mod 9 = 3) &&= \gcd(3, 9 \mod 3 = 0)\\ &= 3.\end{aligned}$$

The gcd of 522 and 453 is equal to 3.

Extended Euclidean algorithm. The "extended" version of the Euclidean algorithm – the one we will use most of the time in this book – enables us to compute the gcd of two numbers and a pair of Bézout numbers.

This algorithm is also extended because it is meant to be more general: It can be not only applied to number sets but also applied to any Euclidean ring. This is the case for polynomials, as we see in the following sections.

The principle of the algorithm is to iterate with the following function G:

$$G \ : \ \begin{bmatrix} a \\ b \end{bmatrix} \longmapsto \begin{bmatrix} 0 & 1 \\ 1 & -(a \text{ div } b) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}.$$

Example 1.4 We wish to find x and y such that $x \times 522 + y \times 453 = \gcd(522, 453)$. We write the matrices corresponding to the iterations with the function G starting from

$$\begin{bmatrix} 522 \\ 453 \end{bmatrix}$$

One first computes $-(a \text{ div } b)$ with $a = 522$ and $b = 453$ and one gets the matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}.$$

Then, one iterates the algorithm with $\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 522 \\ 453 \end{bmatrix}$. Thus, at the end, one gets

$$\begin{aligned} \begin{bmatrix} 3 \\ 0 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -6 \end{bmatrix} \\ &\quad \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 522 \\ 453 \end{bmatrix} \\ &= \begin{bmatrix} 46 & -53 \\ -151 & 174 \end{bmatrix} \begin{bmatrix} 522 \\ 453 \end{bmatrix} \end{aligned}.$$

Hence, we have $46 \times 522 - 53 \times 453 = 3$. Thus, $d = 3$, and the Bézout's numbers are $x = 46$ and $y = -53$.

Here is a version of the extended Euclidean algorithm that performs this computation while storing only the first line of G. It modifies the variables x, y, and d in order to verify at the end of each iteration that $d = \gcd(a, b)$ and $ax + by = d$.

In order to resolve it "by hand," we can calculate recursively the following equations (E_i) (we stop when $r_{i+1} = 0$):

$$\begin{array}{llcccl} (E_0) : & 1 \times a & + & 0 \times b & = & a \\ (E_1) : & 0 \times a & + & 1 \times b & = & b \\ (E_{i+1}) = (E_{i-1}) - q_i(E_i) & u_i \times a & + & v_i \times b & = & r_i \end{array}.$$

The following exercise gives examples of resolution using this method.

Exercise 1.14 (Extended Euclidean Algorithm) *Find pairs of Bézout numbers for the following integers:*

- $(a, b) = (50, 17)$
- $(a, b) = (280, 11)$
- $(a, b) = (50, 35)$

Solution on page 284.

Now let us prove that this algorithm is correct if the Euclidean ring is the set $\mathbb{Z}$ of integers. This will also provide a constructive proof of Bézout's theorem (see page 32) in $\mathbb{Z}$.

Theorem 3 *The extended Euclidean algorithm is correct in $\mathbb{Z}$.*

Proof. First of all, let us show that the sequence of remainders is always divisible by $d = \gcd(a, b)$: recursively, if $r_{j-2} = kd$ and $r_{j-1} = hd$, then $r_j = r_{j-2} - q_j r_{j-1} = d(k - q_j h)$ and thus $\gcd(a, b) = \gcd(r_{j-1}, r_j)$. Moreover, the sequence of positive remainders r_j is monotonically decreasing and is bounded below by 0. Hence, the algorithm ends.

Besides, after a finite number of steps, one has $r_j = 0$. Thus, there exists an index i such that $r_{i-1} = q_{i+1} r_i + 0$. In that case, $\gcd(r_{i-1}, r_i) = r_i$ and the previous remark indicates that r_i is the gcd we are looking for.

Finally, we need to prove that the Bézout numbers are correct. Let us do it recursively. Obviously, the initial case $a \mod b = 0$ is correct and so is the algorithm. Then, let us denote $r = a \mod b$ and $q = a$ div b. Hence $a = bq + r$. Recursively, and using the notation introduced in Algorithm 1.4, we have $d = xb + yr$ with $|x| \leq b$ and $|y| \leq r$. This relation implies that $d = ya + (x - qy)b$ with $|y| \leq r \leq b$ and $|x - qy| \leq |x| + q|y| \leq b + qr = a$. Thus, the algorithm is correct. □

Algorithm 1.4 GCD: Extended Euclidean Algorithm

Input Two elements a and b of an Euclidean ring.
Output $d = \gcd(a, b)$ and x,y such that $ax + by = d$, and $\nu(x) \leq \nu(b)$, $\nu(y) \leq \nu(a)$.

1: **If** $b = 0$ **then**
2: Return $d \leftarrow a$, $x \leftarrow 1$, $y \leftarrow 1$
3: **else**
4: Recursive call of extended Euclidean algorithm on b, $a \mod b$
5: Let d, u, v be the elements of the result.
6: Compute $y \leftarrow u - (a \text{ div } b) * v$
7: **return** $d, x \leftarrow v, y$
8: **End If**

Exercise 1.15 (Modular Computation) *The extended Euclidean algorithm also enables one to solve linear modular equations in $\mathbb{Z}$.*

Give a method to solve the following equations:

1. $17x = 10 \mod 50$

2. $35x = 10 \mod 50$
3. $35y = 11 \mod 50$

Solution on page 284.

Complexity of the Euclidean algorithm. At each step, the greatest number is, at least, divided by two, hence its size decreases by 1 bit, at least. Thus, the number of steps is bounded by $O(\log_2(a) + \log_2(b))$. At each step, we also compute the remainder of the Euclidean division. The algorithms for Euclidean division we learned in elementary school have a cost $O(\log_2^2(a))$. Finally, the overall cost is bounded by $O(\log_2^3(a)) = O(n^3)$ if n is the size of the data. However, a closer study of the algorithm can make this complexity more precise. Indeed, the cost of the Euclidean algorithm is rather of the order $O(\log_2^2(a))$!

The proof is technical, but the idea is rather simple: either there are actually about $O(\log_2(a))$ steps, thus each quotient is very small and then each division and multiplication can be performed with $O(\log_2(a))$ operations, or the quotients are large and thus each division and multiplication has to be performed with $O(\log_2^2(a))$ operations (but then the number of steps is small).

Exercise 1.16 *Implement the extended Euclidean algorithm on the set of integer numbers with your favorite programming language (solution coded in C++).*

Solution on page 285.

1.3.3.2 Euler's Totient Function and Fermat's theorem Let $n \geq 2$ be an integer. We denote by $\mathbb{Z}/n\mathbb{Z}^*$ the set of positive integers lower than n and coprime with n:

$$\mathbb{Z}/n\mathbb{Z}^* = \{x \in \mathbb{N} : 1 \leq x < n \text{ and } \gcd(x, n) = 1\}.$$

The cardinal number of $\mathbb{Z}/n\mathbb{Z}^*$ is denoted by $\varphi(n)$. The function φ is called *Euler's totient function*. For example, $\varphi(8) = 4$. Moreover, if p is prime, $\varphi(p) = p - 1$. Exercise 1.19 focuses on a more general formula.

Each element in $\mathbb{Z}/n\mathbb{Z}^*$ has an inverse: indeed, as x is coprime with n, Bézout's identity guarantees the existence of two integers u and v of opposite sign ($|u| \leq n$ and $|v| \leq x$), such that

$$u.x + v.n = 1.$$

Thus, $u.x = 1 \mod n$, that is, $u = x^{-1} \mod n$. The integer u is nonzero and not equal to n because of the relation $u.x = 1 \mod n$ and $n > 1$. So it is an element of $\mathbb{Z}/n\mathbb{Z}^*$ and is called the inverse of x modulo n. Computation of u is performed with the extended Euclidean algorithm.

Theorem 4 (Euler) *Let a be any element in $\mathbb{Z}/n\mathbb{Z}^*$. One has $a^{\varphi(n)} = 1 \mod n$.*

Proof. $\mathbb{Z}/n\mathbb{Z}^*$ is a finite multiplicative and abelian group, with neutral element 1 and of cardinal number $\varphi(n)$. Therefore, Lagrange Theorem 1, page 31, applies directly to give $a^{\varphi(n)} = 1 \mod n$. □

Fermat's theorem can be immediately deduced from Euler's theorem when n is prime.

Theorem 5 (Fermat) *If p is prime, then any $a \in \mathbb{Z}/p\mathbb{Z}$ satisfies the following result: $a^p = a \mod p$.*

Proof. If a is invertible, then Euler's theorem gives $a^{p-1} = 1 \mod p$. We multiply the equation by a in order to obtain the relation. The only noninvertible element in $\mathbb{Z}/p\mathbb{Z}$ (if p is prime) is 0. In that case, we have obviously $0^p = 0 \mod p$. □

The Chinese Remainder Theorem – which was first formulated by Chinese mathematician Qin Jiu-Shao during the *XIII*th century – enables one to combine several congruences modulo pairwise coprime numbers in order to obtain a congruence modulo the product of these numbers.

Theorem 6 (Chinese Remainder Theorem) *Let $n_1, \ldots, n_k$ be positive pairwise coprime numbers and $N = \prod n_i$. Then, for all set of integers $a_1, \ldots, a_k$, there exists a unique solution $0 \leq x < N$ to the modular equation system $\{x = a_i \mod n_i$, for $i = 1...k\}$. If we call $N_i = \frac{N}{n_i}$, this unique solution is given by*

$$x = \sum_{i=1}^{k} a_i N_i N_i^{-1 \mod n_i} \mod N,$$

where $N_i^{-1 \mod n_i}$ is the inverse of N_i modulo n_i.

Proof. Let us proceed in two steps: first, we prove the existence of x, and then we prove the uniqueness of x. As n_i are pairwise coprime, N_i and n_i are coprime. Bézout's theorem gives us the existence of the inverse of N_i modulo n_i, which is written $y_i = N_i^{-1 \mod n_i}$. Let $x = \sum_{i=1}^{n} a_i y_i N_i \mod N$. It is easy to check that x is a solution of the system of congruences ! Indeed, for all i, as n_i divides all N_j (with $j \neq i$), $x = a_i y_i N_i \mod n_i$. According to the definition of y_i, we have $x = a_i.1 = a_i \mod n_i$.

Now let us prove the uniqueness of x. Let us suppose that there exist two solutions x_1 and x_2. Then $x_2 - x_1 = 0 \mod n_1$ and $x_2 - x_1 = 0 \mod n_2$. Thus, $x_2 - x_1 = k_1 n_1 = k_2 n_2$ for some k_1 and k_2. Hence, n_1 divides $k_2 n_2$. Yet, n_1 and n_2 are coprime, thus n_1 also divides k_2; hence $x_2 - x_1 = 0 \mod n_1 n_2$. Processing recursively, as n_{i+1} is coprime with the product $n_1 n_2 \ldots n_i$, we can deduce that $x_2 - x_1 = 0 \mod N$, or $x_2 = x_1 \mod N$, which gives us the uniqueness of the solution. □

Exercise 1.17 *Find all integers x such that $x = 4 \mod 5$ and $x = 5 \mod 11$. Deduce the inverse of 49 modulo 55.* *Solution on page 285.*

Exercise 1.18 *Find all integers x whose remainders after division by 2, 3, 4, 5, 6 are, Respectively, 1, 2, 3, 4, 5.* *Solution on page 285.*

Exercise 1.19 (A formula for Euler's Totient Function)

1. *Let p be a prime number,* $\varphi(p) = p - 1$. *Compute* $\varphi(n)$ *with* $n = p^k$ *and* $k \in \mathbb{N}^*$.
2. *Show that* φ *is* multiplicative*, that is, if m and n are coprime, then* $\varphi(mn) = \varphi(m)\varphi(n)$.
3. *Deduce a general formula for Euler's previous theorem, using the prime factor decomposition.*

Solution on page 286.

Exercise 1.20 (The Chinese Remainder Theorem) *Let* $(n_1, \dots, n_k)$ *be pairwise coprime integers and* $N = \prod_{i=1}^{k} n_i$. *We consider the following application:*

$$\begin{array}{cccccc} \Psi : & \mathbb{Z}/N\mathbb{Z} & \longrightarrow & \mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z} & & \\ & a & \longrightarrow & (a_1, \dots, a_k) & & \forall i \in [1, k],\ a = a_i \mod n_i \end{array}$$

1. *Prove that* Ψ *is a ring isomorphism.*
2. *Characterize the function* Ψ^{-1}
   Hint*: Use* $N_i = \frac{N}{n_i}$ *and notice that* $\gcd(N_i, n_i) = 1$.
3. *Give the unique solution modulo N of the system:*

$$\begin{cases} x = a_1 \mod n_1 \\ \vdots \\ x = a_k \mod n_k \end{cases}$$

Solution on page 286.

Exercise 1.21 (Application: The Chinese Remainder Problem) *This exercise is a typical application of the Chinese Remainder Theorem. A group of 17 pirates has laid hands on a booty composed of golden coins of equal value. They decide to share it equally and to give the rest to the Chinese cook. The latter would receive three coins. However, the pirates quarrel and six of them are killed. The cook would receive four coins. Unfortunately, the boat sinks and only six pirates, the treasure and the cook are saved. The sharing would give five golden coins to the cook. What is the minimal fortune the latter can expect if he decides to poison the rest of the pirates ?*

Note*: one may use the following equalities:*

- $17 \times 11 \times 6 = 1122$ *and* $66 = 3 \times 17 + 15$
- $8 \times 66 \times 3 = 1584$ *and* $16 \times 102 = 1632$
- $4151 = 3 \times 1122 + 785$

Solution on page 287.

1.3.3.3 Modular Exponentiation and Discrete Logarithm Modular exponentiation is a form of coding widely used in modern encryption methods. Let a be a generator of $\mathbb{Z}/n\mathbb{Z}$. Consider the function

$$\begin{array}{cccc} E_a : & \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}/n\mathbb{Z} \\ & b & \longrightarrow & a^b \mod n. \end{array}$$

It is associated with a decoding function. As a is a generator, every element c in $\mathbb{Z}/n\mathbb{Z}$ may be written as a power of a. The lowest positive integer b such that $a^b = c \mod n$ is called the *discrete logarithm* (or the *index*) in base a of b modulo n. We denote $b = \log_a c \mod n$.

$$\begin{array}{cccc} D_a : & \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}/n\mathbb{Z} \\ & c & \longrightarrow & \log_a c \mod n. \end{array}$$

The coding function is easy to compute. The method is called the *exponentiation by squaring* (or *binary exponentiation*, or even *square-and-multiply*). It consists in writing b as successive squares.

For example,

$$a^{11} = a \times a^{10} = a \times (a^5)^2 = a \times (a \times a^4)^2 = a \times (a \times (a^2)^2)^2.$$

With this principle, the computation of a^{11} only requires five multiplications: three exponentiations by squaring and two multiplications.

More generally, the complexity bound of Algorithm 1.5 is $O(\log_2 n)$ multiplications modulo n.

Algorithm 1.5 Modular Exponentiation

Input Three integers $a \neq 0$, b and $n \geq 2$.
Output $a^b \mod n$
1: **If** $b = 0$ **then**
2: **return** 1
3: **else**
4: Compute recursively the modular exponentiation $a^{\lfloor b/2 \rfloor} \mod n$
5: Let d be the result
6: Compute $d \leftarrow d * d \mod n$
7: **If** b is odd **then**
8: Compute $d \leftarrow d * a \mod n$
9: **End If**
10: **return** d
11: **End If**

Indeed, at each call, the exponent b is divided by 2. Hence, there are at most $\log_2 b$ recursive calls. At each call, we perform at most two multiplications: a squaring and

possibly a multiplication by a. These operations are performed modulo n, that is to say on numbers of $\log_2 n$ bits. Even using the naive multiplication algorithms (those we have seen in elementary school), the cost of such multiplication is $O(\log_2^2 n)$.

Thus, the overall cost of the algorithm is $O(\log_2 b \log_2^2 n)$. This cost is reasonable with regard to $O(\log_2 n)$, which is the time required to read a or write the result.

Exercise 1.22 (Computations of the Inverse)

1. *Propose an algorithm for the computation of the inverse in $\mathbb{Z}/n\mathbb{Z}$ whenever it exists, based on Euler's theorem.*
   Application: compute (quickly) $22^{-1} \mod 63$ *and* $5^{2001} \mod 24$. *One can use the following results:* $22^2 \mod 63 = 43$; $22^4 \mod 63 = 22$.
2. *Give three different algorithms for the computation of the inverse of y modulo $N = p_1^{\delta_1}.p_2^{\delta_2}.\ldots p_k^{\delta_k}$, with p_i distinct prime integers.*

Solution on page 287.

The Discrete Logarithm Problem (DLP) deals with the computation of the inverse of the modular power. We have seen that modular exponentiation can be computed in reasonable time. However, this is not the case for discrete logarithms. This skewness is a fundamental principle in cryptography.

The following result is called the discrete logarithm theorem. Recall that a *generator* of the set $\mathbb{Z}/n\mathbb{Z}^*$ is a number g such that $\{g^i,\ i \in \mathbb{N}\} = \mathbb{Z}/n\mathbb{Z}^*$.

Theorem 7 (Discrete Logarithm) *If g is a generator of $\mathbb{Z}/n\mathbb{Z}^*$, then for all $x, y \in \mathbb{N}$: $g^x = g^y \mod n$ if and only if $x = y \mod \varphi(n)$.*

Proof. ($\Leftarrow$) If $x = y \mod \varphi(n)$, one has $x = y + k * \varphi(n)$. Yet, $g^{\varphi(n)} = 1 \mod n$, hence $g^x = g^y \mod n$.

($\Rightarrow$) As the sequence of powers of g is periodic of period $\varphi(n)$, then $g^x = g^y \mod n \Longrightarrow x = y \mod \varphi(n)$. □

However, this does not enable one to compute the discrete logarithm with reasonable complexity. Given y, it is difficult to compute x such that $g^x = y$. The only simple method consists in enumerating exhaustively all possible x and it takes a time $O(n)$. No polynomial time algorithm in $\log_2 n$ (size of the input) is known for this problem.

Thereby, if $n = 10^{100}$, modular exponentiation requires less than 10^8 operations, and it takes less than a second for a PC. On the contrary, exhaustive enumeration for computing the discrete logarithm requires 10^{100} operations, which is unimaginable in reasonable time according to what we have seen before !!!

In practice, one can apply principles similar to those used in factorization algorithms in order to attempt to solve the discrete logarithm problem.

This kind of function – for which one way can be easily computed but not the other one – is crucial in coding, especially for public key cryptography.

1.3.3.4 One-Way Functions In cryptosystems called *public key cryptosystems*, the "encoding system" has to be known while the decoding has to remain unknown. In this example of encoding with modular exponentiation and decoding with the discrete logarithm, the point of having the encoding function E known and the decoding function D unknown seems contradictory: if one knows E, one inevitably knows D as $D = E^{-1}$.

Actually, replacing "unknown" by "extremely difficult to compute on a computer" (i.e., several years for instance), the functions E and D of a public key cryptosystem must satisfy:

- $D = E^{-1}$ in order to insure $D(E(M)) = M$;
- it is easy (i.e., it can be done quickly) to compute $\widetilde{M} = E(M)$ from M; and
- it is difficult (i.e., it takes a very long time) to recover M from $\widetilde{M}$.

In other words, the problem is to find an encryption function E, which is fast to compute but is long to invert. Such a function is called a *one-way function* (also known as OWF). This notion is absolutely crucial in cryptography and all modern codes are based on it. The principle is illustrated in Figure 1.8.

Adding a key to the functions will make decoding easier if one has the key and will make it more difficult if one does not have the key.

Good OWFs are functions such that the research of x given $f(x)$ is a mathematical problem that is putatively difficult.

There are two interests in calculating in modular arithmetic. First of all, computations "modulo n" are quite fast: their cost is $O(log^2 n)$ using the most naive algorithms. Moreover, iterations of a function F – even a simple one – computed using arithmetic modulo n tend to have some random behavior. We see in Section 1.3.7 that this kind of computation is used in the major part of pseudo-random generators. Knowing F and n large, it seems difficult to solve the equation: find x such that $F(x) = a \mod n$, hence to invert the function F.

1.3.4 Construction of Prime Fields and Finite Fields

We have mentioned that we try to give field structures to our codes when possible in order to make operations easier. Now we have a first method of generating good codes: prime fields. It is sufficient to choose a prime number p, and to equip the set

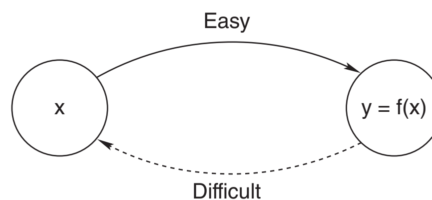

Figure 1.8 Principle of a one-way function

$\{0, \ldots, p-1\}$ with the addition and the multiplication modulo p. However, "finding a prime number" is not easy. It is a full-fledged field of research of which we will give a survey in order to leave no algorithmic void in our coding methods.

1.3.4.1 Primality Tests and Prime Number Generation Even if one does not know a polynomial time algorithm for the factoring of an integer n (polynomial with respect to the size $\log_2 n$ of n), it is still possible to quickly generate a prime number p. In coding, it is very useful to be able to generate prime numbers, both for building structured codes such as fields – which can easily be manipulated for error correction – and for setting up secured cryptosystems. For this, one uses primality tests, namely algorithms that determine whether a given number is prime or not. Taking a large odd number n, and applying the test on it, if n is "composite" one can restart the algorithm with $n+2$ until one finds a prime number. The number of prime numbers less than n is asymptotically $n/\ln(n)$. One deduces that – starting from n odd – on average $O(\ln(n))$ iterations are sufficient to find a prime number by adding 2 to n at each step.

The most used primality test was proposed by Miller and was made efficient in practice by Rabin. The Miller–Rabin test is an algorithm that determines whether a given number is probably prime or not. Therefore, the response given by the computation is only a probabilistic one, and it might be erroneous. Nevertheless, if one repeats the test a sufficient number of times and if the latter constantly gives the same response, the error probability will become smaller and smaller and eventually negligible.

Miller-Rabin test. Let n be odd and let s and t such that $n-1 = t2^s$ with t odd. For any integer $a < n$, one has

$$a^{(n-1)} - 1 = a^{t2^s} - 1 = (a^t - 1)(a^t + 1)(a^{2t} + 1) \ldots (a^{(2^{s-1})t} + 1).$$

If n is prime, according to Fermat's theorem, $a^{n-1} - 1 = 0 \mod n$; therefore

- Either $a^t - 1 = 0 \mod n$;
- Or $a^{t2^i} + 1 = 0 \mod n$ for some i, $0 \le i < s$.

The Miller–Rabin composition test is based on this property.

One says that a has succeeded in the Miller–Rabin composition test for n if $a^t - 1 \neq 0 \mod n$ and $a^{t2^i} + 1 \neq 0 \mod n$ for all $i = 0, \ldots, s-1$.

If n is odd and composite, there are less than $(n-1)/4$ integers a, which fail in the Miller–Rabin composition test. Therefore, by choosing a randomly in $\{1, \ldots, n-1\}$, the error probability is lower than $\frac{1}{4}$.

This test can be efficiently used in order to build a probable prime number with an error probability lower than 4^{-k}. One proceeds as follows:

1. Choose randomly an odd integer n.

Algorithm 1.6 Miller–Rabin Primality Test

Input An odd integer $n \geq 5$.
Output n is either *composite* or *probably prime*.

1: Let s and t be such that $n - 1 = t2^s$
2: Let a be a randomly chosen integer between 2 and $n - 1$.
3: Let $q \leftarrow a^t \mod n$
4: **If** $q = 1$ or $q = n - 1$ **then**
5: **return** "n is probably prime"
6: **End If**
7: **For** i from 1 to $s - 1$ **do**
8: $q \leftarrow q * q \mod n$
9: **If** $q = n - 1$ **then**
10: **return** "n is probably prime"
11: **End If**
12: **End For**
13: **return** "n is composite".

2. Choose randomly k distinct numbers a_i, $1 < a_i < n$. Apply the Miller–Rabin composition test for each integer a_i.
3. If no a_i succeeds in the composition test, we deduce that n is prime; the error probability is lower than 4^{-k};
4. Otherwise, repeat the loop using $n + 2$ instead of n.

The complexity bound of the Miller–Rabin test is similar to that of modular exponentiation, namely $O(\log_2^3 n)$; and $O(k \log_2^3 n)$ if one wants an error probability of around 4^{-k}.

Thus, the average arithmetic cost of the generation of prime numbers is bounded by $O(k \log^4 n)$. Indeed, as there are about $\frac{n}{\ln(n)}$ prime numbers less than n, it would take an average of $\ln(n)$ tries to find a prime number less than n.

In practice, using this test, it is easy to generate a 1000 digit prime number with an error probability arbitrarily low.

Besides, it is possible to make the Miller–Rabin algorithm deterministic by testing a sufficient number of integers a. For example, Burgess proved that testing all integers a lower than $n^{0.134}$ was enough to obtain a prime number with certainty. However, the test would then become exponential in the size of n.

Finally, in 1990, a theorem proved that, assuming the generalized Riemann hypothesis, one of the most famous conjectures in mathematics, it is enough to test the $2 \log_2 n$ first integers. Thus, theoretical studies show that this test is efficient and reliable.

Agrawal–Kayal–Saxena (AKS) primality test. In order to have an overall survey of this topic, let us mention a new primality test – the AKS test – which was built by Agrawal, Kayal, and Saxena. In 2002, they proved the existence of a polynomial time

deterministic algorithm that determines whether a number is prime or not without using the Riemann hypothesis. So far, despite this important theoretical result, in practice, one prefers probabilistic algorithms because of their efficiency.

The idea is close to Mille-r-Rabin's and uses the formalism of polynomials: if n is prime, then for all a

$$(X-a)^n = (X^n - a) \mod n. \tag{1.10}$$

The AKS algorithm checks this equality for some values of a, by developing explicitly $(X-a)^n$. For this test to have a polynomial algorithmic complexity bound, one needs to reduce the size of the polynomials (this is done by performing the test modulo $(X^r - 1)$ with r satisfying some properties[3]) and to use a sufficient number of witnesses a, but only of the order of $log^{O(1)}(n)$, that is, a polynomial of the size of n.

We only give a rough idea of the justification of the AKS test, all the more as we have not introduced polynomials yet. This is what we are going to do now, because they constitute a necessary formalism to construct finite fields and codes.

We know how to build large prime fields by just computing large prime numbers. However, these are not the only existing finite fields. In order to build finite fields of any size (even if this size is not a prime number) – and provided that there exist such fields – we now have to introduce the ring of polynomials over any field.

1.3.4.2 Arithmetic of Polynomials Let V be a field. We call a *sequence* a mapping of $\mathbb{N}$ onto V. The image of $i \in \mathbb{N}$ in a sequence a is denoted by a_i. The *support* of a sequence a is the number of nonzero elements in the image of a. A *polynomial P* on V is a sequence with a finite support. The numbers a_i are called the *coefficients* of P. The *degree* of a polynomial P is the highest i, such that a_i is nonzero and is denoted by $\deg(P)$. The coefficient $a_{\deg(P)}$ is then called the *leading coefficient*. A polynomial is *monic* if its leading coefficient is equal to the neutral element for the multiplication in V.

The addition of two polynomials P and Q with coefficients a_i and b_i results in the polynomial $R = P + Q$ with coefficients $c_i = a_i + b_i$ for all $i \in \mathbb{N}$. The multiplication of the two polynomials P and Q results in the polynomial $R = P.Q$, with coefficients $c_i = \sum_{k=0}^{i} a_k b_{i-k}$ for all $i \in \mathbb{N}$.

Let X be the polynomial such that $X_0 = 0$, $X_1 = 1$, and $X_i = 0$ for all $i > 1$. Any polynomial P of degree d may be written as

$$P(X) = \sum_{i=0}^{d} a_i X^i \text{ , where } d \in \mathbb{N}, (a_0, \dots a_d) \in V^{d+1}.$$

The utility of such a notation is among others to define a function P for any polynomial P: to each element x of V, $P(x) = \sum_{i=0}^{d} a_i x^i$. Now to efficiently evaluate the latter expression for an element x, one would use the *Horner scheme* of Algorithm 1.7.

[3] r must be coprime with n, the greatest prime factor q of r must satisfy $q \geq 4\sqrt{r}\log n$, and n must also satisfy $n^{\frac{r-1}{q}} \neq 1 \mod r$.

Algorithm 1.7 Horner Polynomial Evaluation

Input $(a_0, \dots a_d) \in V^{d+1}$, and $x \in V$.
Output $\sum_{i=0}^{d} a_i x^i$.
1: Let $s = a_d$;
2: **For** i from $d - 1$ to 0 **do**
3: $s = s * x + a_i$;
4: **End For**
5: **return** s.

The set of all polynomials with these operations is a ring, denoted by $V[X]$. The null element is the all-zero sequence and the neutral element is the polynomial with coefficients $e_0 = 1$ and $e_i = 0$ for all $i > 0$. It is a principal and Euclidean ring, with the degree as an Euclidean function. Indeed, one can define a Euclidean division: for all nonnull polynomials A and B, there exist two unique polynomials Q and R with $\deg(R) < \deg(B)$ such that

$$A = B \cdot Q + R.$$

The polynomial $Q = A \text{ div } B$ is the *quotient* in the Euclidean division of A by B; also the *remainder* R is denoted $A \mod B$.

The notion of Greatest Common Divisor (GCD) is then defined; the extended Euclidean algorithm can be applied to two nonzero polynomials A and B and provides a polynomial of maximum degree (it is unique if monic) that divides both A and B. Besides, Bézout's identity is valid. In other words, if A and B are two polynomials in $V[X]$ and $D \in V[X]$ is their gcd, there exist two polynomials S and T in $V[X]$ such that $\deg(S) \leq \deg(B)$ and $\deg(T) \leq \deg(A)$ and

$$A.S + B.T = D.$$

If A and B are different from the polynomial 1 (the neutral element), the *extended Euclidean algorithm* enables one to compute two polynomials S and T whose respective degrees are strictly lower than those of A div D and B div D.

Two polynomials A and B are said to be *coprime* if their GCD is equal to the polynomial 1; in other words, A and B have no common factor of nonzero degree. One says that the nonconstant polynomial P of $V[X]$ is an *irreducible polynomial* over V if P is coprime with all nonconstant polynomials of degree lower than $\deg(P)$.

As for the prime factor decomposition of any integer, any monic polynomial of nonzero degree has a unique factorization in powers of monic irreducible factors over V (up to a constant); one says that $V[X]$ is a *unique factorization domain*. In other words, it is possible to decompose any polynomial of nonzero degree A of $V[X]$ in the form

$$A = P_1^{d_1} \dots P_k^{d_k},$$

where the d_i are nonzero integers and the polynomials P_i irreducible over V. If A is monic, the P_i factors can be chosen monic: the decomposition is then unique, up to a permutation of the indices.

An element α of V is a *root* of $A \in V[X]$ if $A(\alpha) = 0$, where $A(\alpha)$ is the value of the function associated to the polynomial A evaluated in α.

If α is a root of A, then $(X - \alpha)$ divides A. Let B be the polynomial such that $A = (X - \alpha).B$. One says that α is a *simple root* of A if α is not a root of B, that is, $B(\alpha) \neq 0$. Otherwise, if $B(\alpha) = 0$, one says that α is a *multiple* root of A.

Example 1.5 In $\mathbb{F}_2[X]$, the polynomial $X^3 - 1$ can be factorized into $X^3 - 1 = (X - 1) \cdot (X^2 + X + 1)$. One can easily check that $X^2 + X + 1$ is irreducible (the only irreducible polynomials in $\mathbb{F}_2[X]$ of nonzero degree lower than 2 are X and $X - 1$; and neither 0 nor 1 are roots of $X^2 + X + 1$).

1.3.4.3 The Ring $V[X]/P$ and Finite Fields Let $(V, +, \times)$ be a field and let P be a polynomial of degree $d \geq 1$. One denotes by $V[X]/P$ the set of polynomials of degree strictly lower than d equipped with the addition and multiplication modulo P. Namely, for all polynomials A, B in $V[X]$, with $\deg(A) < d$ and $\deg(B) < d$:

$$A +_{V[X]/P} B = (A +_{V[X]} B) \mod P$$
$$A \times_{V[X]/P} B = (A \times_{V[X]} B) \mod P.$$

This is a commutative monic ring of neutral elements 0 and 1 with regard to the laws $+$ and $\times$. This ring is called the *quotient ring* of $V[X]$ modulo P.

If P is an irreducible polynomial, then $V[X]/P$ is a field. Indeed, if Q is a nonzero polynomial of degree lower than $\deg P$, then Q and P are coprime and with Bézout's identity $AQ + BP = 1$, one obtains $AQ = 1 \mod P$. In other words, Q is invertible in the quotient ring $V[X]/P$.

Example 1.6 (Over the Field $V = \mathbb{F}_2$) If $P = (X + 1)(X^2 + X + 1)$ (a nonirreducible polynomial), the ring $V[X]/P$ is such that:

$$V[X]/P = \left\{0, 1, X, 1 + X, X^2, 1 + X^2, X + X^2, 1 + X + X^2\right\}.$$

This ring is not a field because $(1 + X)(1 + X + X^2) = 0$ proves that $1 + X$ is not invertible. On the other hand, if one considers $P = X^2 + X + 1$ (an irreducible polynomial), the ring $V[X]/P$ is such that $V[X]/P = \{0, 1, X, 1 + X\}$. This ring is a field as $X(X + 1) = 1$.

Therefore, we now have finite fields that are more general than prime fields. Indeed, our last example provided us with a field of four elements, which is not a prime field.

Finite fields are called *Galois fields*. They are denoted by $\mathbb{F}_q$, where q is the cardinal number of the field. The next property enables us to handle all finite fields by this construction principle and to explain the notation $\mathbb{F}_q$ for "the" finite field of cardinal number q.

Property 6 *Two finite fields of same cardinal number are isomorphic.*

One can use an irreducible polynomial in order to build a finite field. As for prime number generation, looking for irreducible polynomials is a fully fledged domain of which we will give a survey.

1.3.4.4 Irreducible Polynomials In order to build finite fields, we need some irreducible polynomials, as we needed prime numbers in order to build prime fields.

In the same way as we have seen primality tests for numbers in Section 1.3.4.1, we begin by giving a test that enables to recognize irreducible polynomials.

The first test which is easy to perform is to make sure that the polynomial is *square-free*, namely that it does not have for divisor the square of another polynomial. This can be done over a finite field as for any other field by considering the derivative P' of P. For $P(X) = \sum_{i=0}^{d} a_i X^i$, we note $P'(X) \sum_{i=1}^{d} a_i \times iX^{i-1}$ its *derivative* polynomial.

Property 7 *A polynomial P is square free if and only if* $\gcd(P, P') = 1$.

Proof. If P is divisible by a square, then $P = g^2 h$ for some polynomials h and g. It follows that $P' = 2g'gh + g^2h' = g(2g'h + gh')$ and thus g divides the GCD of P and P'. Reciprocally, if $g = \gcd(P, P')$, let us consider an irreducible factor γ of g of degree at least 1. Then $P' = \gamma f$ and $P = \gamma\lambda$ with f and λ polynomials. By differentiating P, one obtains $\gamma f = \gamma'\lambda + \gamma\lambda'$, or $\gamma(f - \lambda') = \gamma'\lambda$. The polynomial γ being irreducible and γ' being of degree strictly lower than the degree of γ, γ, and γ' are coprime. By Gauss's lemma (see page 33), γ necessarily divides λ and γ^2 divides P. □

The principle of the irreducibility test is given by the following property.

Proposition 1 *For p prime and $d \geq 1$, in $\mathbb{F}_p[X]$, the polynomial $X^{p^d} - X$ is the product of all irreducible, monic polynomials whose degree divides d.*

In order to prove this proposition, we will need the following lemma:

Lemma 3 $r > 0$ *divides* $d > 0$ *if and only if* $p^r - 1$ *divides* $p^d - 1$.

Proof. If r divides d, then $p^d = (p^r)^k = (1)^k = 1 \mod p^r - 1$. Reciprocally, one has $p^d - 1 = 0 \mod p^r - 1$. Let us suppose that $d = qr + s$, with $s < r$. Then, one has $p^d - 1 = p^{qr}p^s - 1 = p^s - 1 \mod p^r - 1$. Yet, $0 \leq s < r$, thus one obtains $p^s - 1 = 0$ over the integers, which implies that $s = 0$. Thus, r divides d. □

Proof. [of Proposition 1] Let P be irreducible of degree r, such that r divides d. Then $V = \mathbb{F}_p[X]/P$ is a field. In V, the order of any nonzero element divides the cardinal number of the group of its invertible elements, namely $p^r - 1$ (Section 1.3.2.1). One applies this property to $X \in V$, so that $X^{p^r-1} = 1$. According to the lemma, $p^r - 1$ divides $p^d - 1$, hence $X^{p^d-1} = 1 \mod P$, and thus P divides $X^{p^d-1} - 1$.

Reciprocally, let P be an irreducible divisor of $X^{p^d} - X$ of degree r. Then, one has $X^{p^d} = X \mod P$. Now, set $G(X) = \sum a_i X^i$ of maximum order $p^r - 1$ in the group

of invertible elements of the field $\mathbb{F}_p[X]/P$ (there always exists at least one such element, see page 56). One then applies Equation (1.10) d times in order to obtain $G^{p^d} = \sum a_i(X^{p^d})^i \mod p$. Now, $X^{p^d} = X \mod P$ and thus $G^{p^d} = G \mod P$ or $G^{p^d-1} = 1 \mod P$. Hence, $p^d - 1$ is necessarily a multiple of $p^r - 1$, the order of G. The lemma enables to conclude that r actually divides d.

One then just needs show that no square divides $X^{p^d} - X$. Indeed, its derivative polynomial is $p^d X^{p^d-1} - 1 = -1 \mod p$ and the polynomial -1 is coprime with any other polynomial. □

Thus, the factors of $X^{p^d} - X$ are all the irreducible, monic polynomials whose degree divides d. If a polynomial of degree d has no common factor with $X^{p^i} - X$ for $1 \le i \le d/2$, it is irreducible. From this property, we can build a test called Ben-Or's irreducibility test (Algorithm 1.8).

Algorithm 1.8 Ben-Or's Irreducibility Test

Input A polynomial, $P \in \mathbb{F}_p[X]$.
Output "P is reducible" or "P is irreducible".
1: Let P' be the derivative polynomial of P.
2: **If** gcd $(P, P') \neq 1$ **then**
3:   **return** "P is reducible".
4: **End If**
5: Let $Q \leftarrow X$
6: **For** i from 1 to $\frac{\deg(P)}{2}$ **do**
7:   $Q \leftarrow Q^p \mod P$
8:   **If** gcd$(Q - X, P) \neq 1$ **then**
9:     **return** "P is reducible" (end of the algorithm).
10:   **End If**
11: **End For**
12: **return** "P is irreducible".

Therefore, we may generate random polynomials and, using this test, see if they are irreducible. One denotes by $m_r(p)$ the number of irreducible, monic polynomials of degree r in $\mathbb{F}_p[X]$. As $X^{p^r} - X$ is the product of all irreducible, monic polynomials whose degree divides r, one obtains

$$\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \le m_r(p) \le \frac{1}{r} p^r. \tag{1.11}$$

Indeed, p^r is the degree of $X^{p^r} - X$, thus $p^r = \sum_{d|r} d m_d(p) \ge r m_r(p)$. Hence $m_r(p) \le p^r/r$. On the other hand, $x m_x(p) \le p^x$ implies that $p^r - r m_r = \sum_{d|r, d \neq r} d m_d(p) \le \sum_{d|r, d\neq r} p^d \le \sum_{d \le \lfloor r/2 \rfloor} p^d$. The latter is a geometric series whose value is $\frac{p^{\lfloor r/2 \rfloor + 1} - 1}{p - 1} < p^{\lfloor r/2 \rfloor + 1}$. Finally, $\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \le m_r(p)$.

This statement shows that among all polynomials of degree r, about one over r is irreducible. One wishes to build an irreducible polynomial. At first sight, it is possible to choose a polynomial randomly, test its irreducibility, and restart the process until one chances on an irreducible polynomial. On average, one needs r draws to find an appropriate polynomial. However, in order to make computations with polynomials easier, it is interesting to obtain sparse polynomials, namely polynomials with very few nonzero coefficients. In this case, exhaustive research might turn out to be more efficient in practice.

We propose the following hybrid Algorithm 1.9 that produces an irreducible polynomial – preferably a sparse one. It is based on the idea of taking polynomials in the form $X^r + g(X)$ with $g(X)$ chosen randomly of degree significantly lower than r.

Algorithm 1.9 Generation of a sparse irreducible polynomial

Input A finite field $\mathbb{F}_p$, an integer $r > 0$.
Output An irreducible polynomial in $\mathbb{F}_p[X]$, of degree r.

1: **For** d from 2 to $r - 1$ **do**
2: **For all** $a, b \in \mathbb{F}_q, a \neq 0$ **do**
3: **If** $X^r + bX^d + a$ is irreducible **then**
4: Return $X^r + bX^d + a$
5: **End If**
6: **End For**
7: **End For**
8: **Repeat**
9: Select P, monic of degree r, chosen randomly in $\mathbb{F}_q[X]$.
10: **Until** P is irreducible
11: **return** P.

1.3.4.5 Construction of Finite Fields Now, we have all the elements necessary to build a finite field of size p^n with p a prime number. The method of building finite fields is contained in the proof of the following result:

Theorem 8 *For all prime number p and all integers $d > 0$, there exists a field K of p^d elements. This field is unique up to isomorphism.*

Proof. Let p be a prime number and let $\mathbb{F}_p[X]$ be the field of polynomials with coefficients in $\mathbb{F}_p$. For $d = 1$, $\mathbb{F}_p$ is a field with p elements. For $d = 2$, Proposition 1 guarantees the existence of at least one irreducible polynomial as there are p irreducible polynomials of degree strictly less than 2 and p^2, the degree of $X^{p^2} - X$, satisfies $p^2 > p$. For larger d, Equation (1.11) shows that $1 \leq m_d(p)$ and thus there exists at least one irreducible polynomial P of degree d in $\mathbb{F}_p[X]$. Then, the quotient ring $V = \mathbb{F}_p[X]/P$ is a field.

As P is of degree d and $|\mathbb{F}_p| = p$, there are p^d possible remainders. Thus $|V| = p^d$.

According to Property 6 on page 48, any field of cardinal number p^d is isomorphic to V. □

Remark 1 *The isomorphism between $V[X]/P$ and $V^{\deg(P)}$ equips $V^{\deg(P)}$ with a field structure.*

Indeed, to any vector $u = [u_0, \dots, u_{d-1}]$ in the vector space V^d, one can associate in a bijective way the polynomial $\psi(u) = \sum_{i=0}^{d-1} u_i X^i$. Moreover, one has the following property:

$$\text{For all } u, v \in V^d,\ \lambda \in V,\ \ \psi(u + \lambda \cdot v) = \psi(u) + \lambda \cdot \psi(v).$$

Hence, ψ is an isomorphism between V^d and $\psi(V^d) = V[X]/P$. This equips V^d with a field structure in which multiplication is defined by

$$\text{For all } u, v \in V^d, \quad u \cdot v = \psi^{-1}(\psi(u) \cdot \psi(v)).$$

Hence, one can use a field structure with the vectors in $V^{\deg(P)}$.

Exercise 1.23 *Let K be a finite field of cardinal number $q > 0$. Using the map $\Psi : \mathbb{Z} \to K$, defined by*

$$\text{For all } n \in \mathbb{Z},\ \Psi(n) = \underbrace{1_K + 1_K + \dots + 1_K}_{n \text{ times}} = n.1_K,$$

prove that there exists a unique prime number p (called the characteristic *of K), such that for all $x \in K$, $px = 0$.* *Solution on page 288.*

Exercise 1.24 *Sequel of the previous exercise*
Deduce from the previous exercise that the cardinal number of K is a power of p using the fact that K is a vector space over its subfields. Hint: One may obtain a subfield of K isomorphic to $\mathbb{F}_p$. *Solution on page 289.*

Exercise 1.25 (Construction of the Field $\mathbb{F}_4$)

1. *Give a necessary and sufficient condition for a polynomial in $\mathbb{F}_2[X]$ of degree $2 \le n \le 3$ to be irreducible. From this condition, deduce all irreducible polynomials of degrees 2 and 3.*
2. *Deduce all irreducible polynomials of degree 4.*
3. *Set $\mathbb{F}_4 = \{e_0, e_1, e_2, e_3\}$ with e_0 the neutral element for the addition and e_1 the neutral element for the multiplication. Using the first question, write the operation tables ($+, \times$, inverse) in $\mathbb{F}_4$.*

Solution on page 289.

1.3.5 Implementation of Finite Fields

1.3.5.1 Operations on Polynomials A typical construction of arithmetic in a finite field $\mathbb{F}_p$ is – for a given prime number p – to look for some irreducible polynomial P

in $\mathbb{F}_p[X]$ of degree d, then to write the elements of $\mathbb{F}_q = \mathbb{F}_p[X]/P$ as polynomials, or as vectors, and finally to implement the arithmetic operations in $\mathbb{F}_q$.

Example 1.7 (Construction of the Field $\mathbb{F}_{16}$) There exists a field with 16 elements as $16 = 2^4$. In order to build the field $\mathbb{F}_{16}$, one first looks for some monic irreducible polynomial P of degree 4 in $\mathbb{F}_2[X]$. Then one establishes the rules of calculation in $\mathbb{F}_2[X]/P$.

- *Finding P.*
  The irreducible polynomial P is written as $P = X^4 + aX^3 + bX^2 + cX + 1$ with a, b, and c in $\mathbb{F}_2$. In order to determine P, let us examine all possible values for the triplet (a, b, c). One cannot have $(a, b, c) \in \{(0, 1, 1), (1, 0, 1), (1, 1, 0), (0, 0, 0)\}$ as for all these cases, 1 is a root of P. Thus, the triplet (a, b, c) is to be searched for in the set $\{(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$. The only irreducible polynomials over $\mathbb{F}_2$ of degree at most 2 are X, $1 + X$, and $X^2 + X + 1$. To see whether P is irreducible, it is sufficient to compute the GCD of P and $(1 + X)(X^2 + X + 1)$. The calculation (with the Euclidean algorithm for example) of these **GCDs!** (**GCDs!**) shows that the only values of (a, b, c) for which P is irreducible are $(0, 0, 1)$, $(1, 0, 0)$, and $(1, 1, 1)$. Thus, $P = X^4 + X^3 + 1$ is a possible choice for P. Let us make this choice.
- *Operations on polynomials.*

  Thus, the elements of the field are $0, X, X^2, X^3, 1 + X^3, 1 + X + X^3, 1 + X + X^2 + X^3, 1 + X + X^2, X + X^2 + X^3, 1 + X^2, X + X^3, 1 + X^2 + X^3, 1 + X, X + X^2, X^2 + X^3, 1$. Therefore, the operations are performed modulo P. For example, $(X^2)(X + X^3) = 1 + X$.

1.3.5.2 Use of Generators There exist other ways to implement finite fields in which the multiplication will be performed much more quickly.

The idea is to use the property of finite fields according to which the multiplicative group of invertible elements of a finite field is cyclic. Namely, there exists at least one generator and the nonzero elements of the field are generated by this element. Hence, if g is a generator of the multiplicative group of a finite field $\mathbb{F}_q$, all invertible elements can be written as g^i.

One can choose to represent each invertible element g^i simply by using its index i and represent zero by a special index. This construction – in which one represents the elements using their logarithm – is called *exponential representation* or *cyclic representation*, or *Zech's construction*. Then, typical arithmetic operations are greatly simplified using the following proposition:

Proposition 2 *Let $\mathbb{F}_q$ be a finite field and let g be a generator of $\mathbb{F}_q^*$. Then $g^{q-1} = 1_{\mathbb{F}_q}$. In addition, if the characteristic of $\mathbb{F}_q$ is odd, then $g^{\frac{q-1}{2}} = -1_{\mathbb{F}_q}$. Otherwise, $1_{\mathbb{F}_{2^n}} = -1_{\mathbb{F}_{2^n}}$.*

Proof. Clearly, one has $g^{q-1} = 1_{\mathbb{F}_q}$. If the field is of characteristic 2, then, as in $\mathbb{F}_2[X]$, one has $1 = -1$. Otherwise $\frac{q-1}{2} \in \mathbb{Z}$ thus $g^{\frac{q-1}{2}} \in \mathbb{F}_q$. Yet, as we consider a field, the

equation $X^2 = 1$ has at most two roots, 1 and -1. g is a generator, thus the order of g is $q-1$ rather than $\frac{q-1}{2}$. The only remaining possibility is $g^{\frac{q-1}{2}} = -1$. □

This statement gives the following encoding for an element $x \in \mathbb{F}_q$, if $\mathbb{F}_q$ is generated by g:

$$\begin{cases} 0 & \text{if } x = 0 \\ q-1 & \text{if } x = 1 \\ i & \text{if } x = g^i, x \neq 1. \end{cases}$$

In particular, in our encoding scheme, let us denote by $\overline{q} = q - 1$ the codeword associated with $1_{\mathbb{F}_q}$. We will also denote by i_{-1} the index of $-1_{\mathbb{F}_q}$; it is equal to $\frac{q-1}{2}$ if the characteristic of $\mathbb{F}_q$ is odd and equal to $q-1$ otherwise. This enables one to write in a simple way all arithmetic operations.

- Multiplication and division of invertible elements are, respectively, implemented as an addition and a subtraction of indexes modulo $q-1$.
- Therefore, negation (taking the opposite) is simply the identity in characteristic 2 or an addition with $\frac{q-1}{2}$ modulo $q-1$ if the characteristic is odd.
- Addition is the most complex operation. One must implement it using other operations. For example, it is possible to do so the following way: if g^i and g^j (with $j > i$) are two nonzero elements in a finite field, $g^i + g^j = g^i(1 + g^{j-i})$. This requires to store the index of $1 + g^k$ for all k. This is done by precomputing a table, t_plus1, of size q, containing the index of the successor of each element in the field. Eventually, addition is implemented with one subtraction of indexes $(j - i)$, one access to a table $(t_plus1[g^{j-i}])$ and one addition of indices $(i + t_plus1[g^{j-i}])$.

In Table 1.4, we study the calculation of these operations over the indices, assuming the existence of a single "table of successors" of size q. Here, we focus on the

TABLE 1.4 Zech's Construction on Invertible Elements in Odd Characteristic

Operation	Elements	Indexes	Average Cost		
			+/−	Tests	Access
Multiplication	$g^i * g^j$	$i + j\ (-\overline{q})$	1.5	1	0
Division	g^i / g^j	$i - j\ (+\overline{q})$	1.5	1	0
Negation	$-g^i$	$i - i_{-1}\ (+\overline{q})$	1.5	1	0
Addition	$g^i + g^j$	$k = j - i\ (+\overline{q})$			
		$i + t_plus1[k]\ (-\overline{q})$	3	2	1
Subtraction	$g^i - g^j$	$k = j - i + i_{-1}\ (\pm\overline{q})$			
		$i + t_plus1[k]\ (-\overline{q})$	3.75	2.75	1

complexity of the calculation using the least amount of memory possible, considering random elements. We indicate the cost of the computations taking the mean number of additions and subtraction (+/−), the number of tests, and the number of times we use the table.

Exercise 1.26 *Check that the polynomial X is a generator of the field $\mathbb{F}_{16}$, constructed with the irreducible polynomial $P = X^4 + X^3 + 1$. Then for the two polynomials $P_1 = X + 1$ and $P_2 = X^2 + X$, perform $P_1 + P_2$ and $P_1 \times P_2$ using the operations described in Table 1.4.* *Solution on page 289.*

1.3.5.3 Primitive Roots In order to put this implementation into practice, we need to find a way of producing generators of finite fields in the same way as we needed a way of producing prime numbers in order to build prime fields or irreducible polynomials to build nonprime fields.

Generators of prime fields. A generator of the group of invertible elements in $\mathbb{Z}/n\mathbb{Z}$ is called a *primitive root* of n. The least primitive root of m is denoted by $\chi(m)$.

If p is a prime number, then $\mathbb{Z}/p\mathbb{Z}$ always has exactly $\varphi(p-1)$ primitive roots. Indeed, by Lagrange's theorem (Proposition 5 page 5), the order of an element of a group divides the number of elements in the group. This means that the order of any nonzero element of $\mathbb{Z}/p\mathbb{Z}$ divides $p-1$. Now suppose that there exists one primitive root g, that is, g generates the group of invertible elements of $\mathbb{Z}/n\mathbb{Z}$. Then for any nonzero element x, there exists an index j such that $x = g^j$. Then, the order of x is $p-1$, that is, x is also a generator primitive root, if and only if its index is coprime with $p-1$. Now, one has to compute at least one of these $\varphi(p-1)$ primitive roots. For this, one uses the following test, which checks whether the order of an element taken at random is $p-1$ or not. The main difficulty is to factorize $p-1$, at least partially, and we see how to do this in Section 1.4.3.5.

Algorithm 1.10 Test Primitive Root

Input A prime number $p > 0$.
Input An integer $a > 0$.
Output Yes, if a is a primitive root of p; No, otherwise.

1: **For all** q prime and dividing $p-1$, **do** {Factoring of $p-1$}
2: **If** $a^{\frac{p-1}{q}} = 1 \mod p$ **then**
3: Return "No".
4: **End If**
5: **End For**
6: **return** "Yes".

Theorem 9 *Algorithm Test Primitive Root is correct.*

Proof. One uses the result of Section 1.3.2.1: if a is an integer, of order k modulo p, then $a^h = 1 \mod p$ if and only if $k|h$.

One deduces that if the order of a is lower than $p-1$, as it divides $p-1$, then necessarily one of the values $\frac{p-1}{q}$ will be a multiple of the order of a. Otherwise, the only possible value for the order of a is $p-1$. □

Therefore, a first method of finding a primitive root is to test all integers lower than p one after the other, which are not equal to 1, nor to -1, nor any power of an integer; in this way, one is able to find the least primitive root of p. Numerous theoretical results exist, proving that it does not take a great number of attempts to find this first primitive root. It is generally of the order of

$$\chi(p) = O\left(r^4(\log(r)+1)^4 \log^2(p)\right)$$

with r the number of distinct prime factors of $p-1$. Another method is to draw random integers lower than p and to test whether they are primitive roots or not. As that there are $\varphi(p-1)$ primitive roots, the probability of success is $\frac{\varphi(p-1)}{p-1}$; thus, the expected value for the number of draws before finding a primitive root is $\frac{p-1}{\varphi(p-1)}$. This gives us a better chance than the brute-force method (trying all possibilities).

Generators of finite fields. Now we know how to find a generator for a prime field. Let us consider the finite fields $\mathbb{F}_{p^k}$. In order to build them, let us recall that, one has first to build $\mathbb{F}_p$ and then to find an irreducible polynomial over this field whose degree is k. The question is how to find a *generator polynomial* of this field in order to encode elements with their index rather than using polynomials. Encoding and arithmetic operations are then the *same* as those of prime fields.

Once again, we use a probabilistic approach. First of all, let us consider an algorithm testing whether a polynomial is a generator in $\mathbb{F}_p[X]$. This algorithm is similar to the one we have seen for primitive roots in $\mathbb{F}_p$.

Algorithm 1.11 Test Generator Polynomial

Input A polynomial $A \in \mathbb{F}_p[X]$.
Input An irreducible polynomial F of degree d in $\mathbb{F}_p[X]$.
Output Yes, if A is a generator of the field $\mathbb{F}_p[X]/F$; No, otherwise.

1: **For all** q, prime and dividing $p^d - 1$ **do** {Factoring of $p^d - 1$}
2:   **If** $A^{\frac{p^d-1}{q}} = 1 \mod F$ **then** {Recursive computation using square exponentiation}
3:     Return "No".
4:   **End If**
5: **End For**
6: **return** "Yes".

Therefore, once the field is built, an algorithm looking randomly for a generator is easy to implement. Besides, one can start the search into the set of polynomials of small degree ($O(\log(n))$). However, in order to manipulate sparse polynomials, it is useful to find an irreducible polynomial for which X is a primitive root. Such a polynomial is called *X-irreducible*, or *primitive* and in general can be quickly found. In practice, for finite fields of size between 4 and 2^{32}, it is possible to show that more than one irreducible polynomial in 12 is X-irreducible! Therefore, an algorithm looking randomly for an X-irreducible polynomial requires less than 12 attempts on average. Thus, an algorithm for finding an irreducible polynomial having X as generator is a simple modification of Algorithm 1.9. If Algorithm 1.11 returns that X is not a generator. one does not select the irreducible polynomial found in Algorithm 1.9.

Example 1.8 Let us return to the example of the field $\mathbb{F}_{16}$, which we built with the irreducible polynomial $P = X^4 + X^3 + 1$.

Algorithm 1.11 performed on X returns that X is a generator. Therefore, one can perform computations using the powers of X (Exercise 1.26).

Recall the identification of the elements in $\mathbb{F}_{16}$ and operation rules:
$X^1 = X \mod P$; $X^2 = X^2 \mod P$; $X^3 = X^3 \mod P$; $X^4 = 1 + X^3 \mod P$; $X^5 = 1 + X + X^3 \mod P$; $X^6 = 1 + X + X^2 + X^3 \mod P$; $X^7 = 1 + X + X^2 \mod P$; $X^8 = X + X^2 + X^3 \mod P$; $X^9 = 1 + X^2 \mod P$; $X^{10} = X + X^3 \mod P$; $X^{11} = 1 + X^2 + X^3 \mod P$; $X^{12} = 1 + X \mod P$; $X^{13} = X + X^2 \mod P$; $X^{14} = X^2 + X^3 \mod P$; and $X^{15} = 1 \mod P$.

With $\mathbb{F}_{16}$ written in form $\mathbb{F}_{16} = \{0, 1, X, X^2, X^3, X^4, X^5, X^6, X^7, X^8, X^9, X^{10}, X^{11}, X^{12}, X^{13}, X^{14}\}$, multiplication and inverse calculation in $\mathbb{F}_{16}$ are performed more easily. Addition is also much easier considering that $X^k + X^t = X^t(1 + X^{k-t})$ for all k and t in $\{1, \dots, 14\}$ such that $k > t$.

1.3.6 Curves Over Finite Fields

The exponentiation over finite fields is a good example of a one-way function, and we now have almost all tools to construct and efficiently compute in those fields. On one hand, the field structure provides many tools for the construction of codes. On the other hand, this structure itself allows more possibilities for code breakers in cryptography. It is possible to define this type of one-way function in a more general structure, a group, so that cryptanalysis is even more difficult. An example of such a group structure is the set of points of a curve defined over a finite field.

In a generic group, we denote by + the group law (which is the multiplication in the group of invertible elements of a finite field $\mathbb{F}_q^*$ for example). Then, the multiplication by an integer (i.e., that integer number of calls to the group law, which is the exponentiation by an integer in $\mathbb{F}_q^*$) can be used as a one-way function. The discrete logarithm problem, in this general formulation, is to find the number of times one has to add a given generator of the group in order to obtain a given element of the group. We denote this as $[n]P = P + P + P + \cdots + P$ for some scalar (integer) n and an element P of a group (Table 1.5).

TABLE 1.5 Discrete Logarithm and Exponentiation in Finite Fields and Generic Groups

Group	Exponentiation	DLP
$\mathbb{F}_q^*$	a^e with $a \in \mathbb{F}_q^*$ and $e \in \mathbb{N}$	Find $x \in \mathbb{N}$ s.t. $g^x = b \in \mathbb{F}_q^*$
G	$a + \dots + a$, $e \in \mathbb{N}$ times, that is, $[e]a$	Find $x \in \mathbb{N}$ s.t. $[x]g = b \in G$

1.3.6.1 Weierstrass Model Let $p \geq 5$ be a prime number, $q = p^k$, and let $a, b \in \mathbb{F}_{p^k}$ such that the polynomial $x^3 + ax + b$ does not have multiple roots and consider the equation

$$y^2 z = x^3 + axz^2 + bz^3. \tag{1.12}$$

If $(x, y, z) \in \mathbb{F}_q^3$ is a solution of (1.12), then any multiple $c(x, y, z)$ is also a solution. Two solutions are called *equivalent* if they are equal up to a constant multiplicative factor. This defines an equivalence relation. The *elliptic curve* $\mathbb{E}(q; a, b)$ is the set of equivalence classes of solutions of (1.12), which are called *points* of the curve. For one equivalence class, noted $(x : y : z)$, if $z \neq 0 \in \mathbb{F}_q$, there exists a representative of the class of the form $(x' : y' : 1)$. Indeed, just set $x' = xz^{-1}$ and $y' = yz^{-1}$. On the other hand, If $z = 0$ then x must also be zero, and there is exactly one equivalence class of solutions with this form. It is denoted by $\mathcal{O}$ and its chosen representative is usually $(0 : 1 : 0)$. In summary the set of points is entirely defined by the cases $z = 1$ and $z = 0$; therefore, the definition of an elliptic curve can be simplified to

$$\mathbb{E}(q; a, b) = \{(x, y) \in \mathbb{F}_q^2, y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}. \tag{1.13}$$

In fact, the general form of the equation of an ellipse is $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$. Now if the characteristic of the field is neither 2 nor 3, then the change of variable $(x, y) \hookrightarrow \left(x, y - a_1x/2 - a_3/2\right)$ yields an isomorphic curve $y^2 = x^3 + b_2x^2 + b_4x + b_6$ and then a second change of variable $(x, y) \hookrightarrow \left(x - b_2/3, y\right)$ enables to simplify the equation to (1.13). This can be generalized for fields of characteristics 2 and 3:

1. If $a_1 \neq 0 \in \mathbb{F}_{2^k}$, use $(x, y) \hookrightarrow (a_1^2x + a_3/a_1, y)$ to get $y^2 + b_1xy = x^3 + b_2x^2 + b_4x + b_6$, followed by $(x, y) \hookrightarrow (x, a_1^3y - b_4/a_1)$, to obtain $\mathbb{E}(2^k; a, b) = \{(x, y) \in \mathbb{F}_{2^k}^2, y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\}$.
2. Else, $a_1 = 0 \in \mathbb{F}_{2^k}$ and $(x, y) \hookrightarrow (x + a_2, y)$ gives $\mathbb{E}(2^k; a, b, c) = \{(x, y) \in \mathbb{F}_{2^k}^2, y^2 + cy = x^3 + ax + b\} \cup \{\mathcal{O}\}$.
3. If $a_1^2 \neq -a_2 \in \mathbb{F}_{3^k}$, use $(x, y) \hookrightarrow (x, y + a_1x - a_3/2)$ to get $y^2 = x^3 + b_2x^2 + b_4x + b_6$, followed by $(x, y) \hookrightarrow (x - b_4/(2b_2), y)$ to obtain $\mathbb{E}(3^k; a, b) = \{(x, y) \in \mathbb{F}_{3^k}^2, y^2 = x^3 + ax^2 + b\} \cup \{\mathcal{O}\}$.
4. Else, $a_1^2 + a_2 = 0 \in \mathbb{F}_{3^k}$ and $(x, y) \hookrightarrow (x, y + a_1x + a_3)$ gives $\mathbb{E}(3^k; a, b) = \{(x, y) \in \mathbb{F}_{3^k}^2, y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$.

Exercise 1.27 *Verify that the given variable changes are correct.*
Solution on page 289.

To make an exhaustive search for a discrete logarithm impossible in practice, the group of points in an elliptic curve has to be large enough. The following theorem states that the number of points is of the order of the size of the involved finite field.

Theorem 10 (Hasse) *For any prime power $q = p^k$, let $N_{q;a,b}$ be the number of points of $\mathbb{E}(q; a, b)$, then*

$$|N_{q;a,b} - (q + 1)| \leq 2\sqrt{q}.$$

1.3.6.2 The Group of Points of an Elliptic Curve Now that we have defined the points in an elliptic curve, we need to provide a group law. We first give the abstract definition.

Theorem 11 *Let $\mathbb{F}_q$ be a field of characteristic greater than 5 and $\mathbb{E}(q; a, b)$ an elliptic curve. Then $(\mathbb{E}(q; a, b), \oplus)$ with the following rules for addition is an abelian group:*

- *$\mathcal{O}$ is the neutral element for $\oplus$.*
- *For $P = (x, y)$, $-P = (x, -y)$ is the opposite of P for $\oplus$.*
- *For $P = (x_1, y_1)$ and $Q(x_2, y_2)$ then:*
  - *if $x_1 \neq x_2$, then $\lambda = \frac{y_2-y_1}{x_2-x_1}$ and*

    $$P \oplus Q = \left(x_3, y_3\right) = \left(\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1\right)$$

  - *else, $x_1 = x_2$ and:*
    * *if also $y_1 = y_2$, then $Q = P$, $P \oplus Q = P \oplus P = [2]P$,*

      $$\lambda = \frac{3x_1^2 + a}{2y_1} \text{ and } [2]P = \left(x_3 = \lambda^2 - 2x_1, y_3 = \lambda(x_1 - x_3) - y_1\right)$$

    * *else $Q = -P$ and $P \oplus -P = \mathcal{O}$.*

The rules of addition derives from the representation of elliptic curves over the real field: if P and Q are two different points on the curve, then $P \oplus Q$ is the symmetric (with respect to the x-axis) of the third intersection of the curve with the line PQ. In the same manner, $[2]P$ is the symmetric (with respect to the x-axis) intersection of the tangent in P with the curve, as shown on Figure 1.9. In both cases, λ is the slope of the line and the y-intercept can naturally be recovered as, for example, $y_1 - \lambda x_1$.

Exercise 1.28 *Let $\mathbb{F}_{5^2} = \mathbb{F}_5[T]/(T^2 + 2)$, $\mathbb{E}(5^2; -1, 1)$, $P = (1, 1)$ and $Q = (3 + T, 2 + T)$.*

1. *Check that $P, Q \in \mathbb{E}(5^2; -1, 1)$.*
2. *Check that $2 + T = (2 - T)^{-1} \in \mathbb{F}_{5^2}$.*

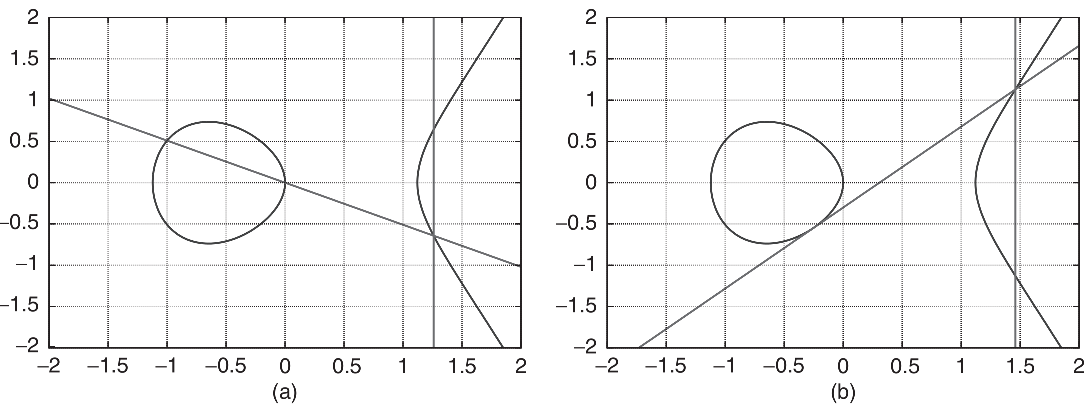

Figure 1.9 (a) Elliptic curve addition and (b) doubling on $y^2 = x^3 - x\sqrt[3]{2}$

3. *Compute $P \oplus Q$ and check that it belongs to the curve.*
4. *Compute $[2]P$, the doubling of P, and check that it belongs to the curve.*
5. *Compute $[2]Q$, the doubling of Q, and check that it belongs to the curve.*

Solution on page 290.

Once again, this law of addition can be generalized in characteristics 2 and 3 as given in Table 1.6.

Moreover, note that using any of these addition laws, the algorithm for multiplication by an integer remains almost exactly Algorithm 1.5, page 41, where multiplications are replaced by $\oplus$ and squarings are replaced by doublings. In this setting, exponentiation by squaring (or square-and-multiply) is often called *double-and-add*.

Exercise 1.29 *Let $\mathbb{E}(7; -1, 1)$ and $P = (1, 1)$, compute $[6]P$ with only three operations.*

Solution on page 290.

Note that there exists many other coordinate systems, such as projective and Jacobian, which differ in the number of multiplications, squarings, additions, or inversions in the base field, for the same group law. The choice of system depends on the respective speed of these operations and the target architecture. Note also that for certain subset of curves, such as Edwards curves, the coordinate system can be simplified, often leading to practical enhancements.

The US National Institute of Standards and Technology (NIST) recommends some elliptic curves[4], which contains a large number of points, with sizes ranging

TABLE 1.6 Group Laws in Characteristics 2 and 3 with $P = (x_1, y_1) \neq -P$ and $Q = (x_2, y_2) \neq \pm P$

p	Curve	Addition	Doubling
2	$y^2 + xy = x^3 + ax^2 + b$ opposite: $(x_1, y_1 + x_1)$	$\lambda = \frac{y_2 + y_1}{x_2 + x_1}$ $(\lambda^2 + \lambda + x_1 + x_2 + a,$ $\lambda(x_1 + x_3) + y_1 + x_3)$	$\lambda = \frac{y_1}{x_1} + x_1$ $(\lambda^2 + \lambda + a,$ $\lambda(x_1 + x_3) + y_1 + x_3)$
	$y^2 + cy = x^3 + ax + b$ opposite: $(x_1, y_1 + c)$	$\lambda = \frac{y_2 + y_1}{x_2 + x_1}$ $(\lambda^2 + x_1 + x_2,$ $\lambda(x_1 + x_3) + y_1 + c)$	$\lambda = \frac{x_1^2 + a}{c}$ $(\lambda^2,$ $\lambda(x_1 + x_3) + y_1 + c)$
3	$y^2 = x^3 + ax^2 + b$ opposite: $(x_1, -y_1)$	$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ $(\lambda^2 - x_1 - x_2 - a,$ $\lambda(x_1 - x_3) - y_1)$	$\lambda = a\frac{x_1}{y_1}$ $(\lambda^2 + x_1 - a,$ $\lambda(x_1 - x_3) - y_1)$
	$y^2 = x^3 + ax + b$ opposite: $(x_1, -y_1)$	$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ $(\lambda^2 - x_1 - x_2,$ $\lambda(x_1 - x_3) - y_1)$	$\lambda = -\frac{a}{y_1}$ $(\lambda^2 + x_1,$ $\lambda(x_1 - x_3) - y_1)$

[4]http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

from 2^{160} to 2^{570}. Many other databases exist, let us mention Bernstein and Lange's Explicit-Formula database[5] and the Acrypta database[6], which contains some Edwards curves.

1.3.7 Pseudo-Random Number Generators (PRNG)

Generation of random numbers is widely used in all the methods we have just seen and will be often used in the sequel. In particular, generating numbers randomly is a condition for the perfection of Vernam's OTP scheme (see page 15). Now, it is time to look more deeply at this problem which needs some development.

The definition of randomness is crucial in coding. Indeed, any message presenting some kind of organization (organization is supposed to be the opposite of randomness) is an angle of attack for compression and code breaking. Therefore, one should rely on a solid theory concerning randomness in order to build secure and efficient codes.

Producing a truly random event is unsolvable by computers – which by definition only respond to determined and predictable processes. In order to obtain values produced by "true" randomness (even if this notion is not completely absolute and refers to what one can observe and predict), one has to call upon assumed unpredictable physical phenomena, such as thermal noise or the description of the Brownian motion of electrons in a resistor.

However, this production of numbers may be called randomness only because we are not able – given our current knowledge in these areas – to explain their mechanisms and because only probabilistic theories enable us to grasp them. With computers, we wish to proceed in the same way in order to generate random numbers. We apply some procedures that make the result unpredictable in practice. This is what we call pseudo-random generation.

Production of random numbers is a very complicated task that has attracted the attention of both machine designers ("hardware" components, such as thermal noise) and software designers ("software" products, see examples in the next section) for some time. One must pay attention to this main issue because there exist some efficient methods (we will study some of them) that enable one to detect nonrandom features in a sequence, which is supposed to be random, to recover the method that produced it, and then to break a randomly generated key. If the machine that generates the winning lotto combination were not based on some good random generation process, one would be able to predict the next winning combination.

One often generates a sequence of pseudo-random numbers by computing each number from the previous one (which obviously makes the process completely deterministic and eliminates all randomness), in such a significantly complicated way that – examining the sequence without knowing the method – one could believe that it was truly generated randomly.

[5]http://hyperelliptic.org/EFD/index.html
[6]http://galg.acrypta.com/telechargements/ARCANA_ECDB.tgz

A generator must satisfy certain properties to be called a pseudo-random generator. All generated numbers have to be independent from each other. Moreover, they must have a great entropy, and hopefully no rule can be recovered from the sequence of generated numbers. There are several ways to determine whether a generator is acceptable. First of all, one has to make it pass some statistical tests in order to check that the distribution it produces does not significantly differ from the one expected from a theoretical model of randomness. Besides, one can also use algorithmic complexity principles: that is show that in reasonable time no algorithm will be able to predict the mechanisms of the generator.

For example, one can build a generator based on the model of Fibonacci's sequence, by producing the numbers $x_n = x_{n-1} + x_{n-2} \mod m$, m being the maximum integer that can be produced. The main advantages of this method are the following: it is very easy to implement, very fast in execution, and "modulo" operation enables one to obtain some hard-to-predict behavior for the generator. However, this generator – like most typical and simple generators – has drawbacks, and it is possible to recover its behavior based on statistical analysis.

The requirements for a pseudo-random generator are very similar to the properties one expects from a ciphertext. Indeed, it must be impossible, when receiving a message or a number, to find out the way it was produced without knowing the key. That is why some methods for random number generation look like cryptographic methods or use them.

1.3.7.1 Congruential Generators One says that a generator is a Linear Congruential Generator (LCG) if it satisfies the following principle: if x_i, $i \in \mathbb{N}$ is the sequence of generated random numbers, one calculates x_i from its predecessor: $x_i = ax_{i-1} + b \mod m$, with m a large number, and $a, b \in \mathbb{Z}/m\mathbb{Z}$. The generator is called multiplicative if $b = 0$. Such a sequence is always periodic; thus, one will have to choose a, b, m such that the period is significantly high. For example, for $m = 10, x_0 = a = b = 7$, the period is 4. Hence, this generator is not satisfactory at all.

The maximum period is obviously m. There exists a result providing a description of all generators of period m with $b = 0$:

Theorem 12 *The Linear Congruential Generator defined by $a, b = 0, m, x_0$ is of period m if and only if x_0 is coprime with m and a is a primitive root of m.*

One usually chooses m as the greatest prime number which can be given by a machine (we have seen how to generate such a number on page 44). Obviously, a large period is not a sufficient criterion for the production of random generators (consider the choice $a = 1, b = 1$). Exercise 1.42, on page 85, is an approach to methods of attacking LCGs.

1.3.7.2 Linear Feedback Shift Register (LFSR) One can generalize Linear Congruential Generators using not only the previous value to build the next element in the sequence but also several previous values, namely x_n is computed from linear

combinations of $x_{n-1}, \ldots, x_{n-k}$. In other words:

$$x_n = (a_1 x_{n-1} + \ldots + a_k x_{n-k}) \mod m.$$

These generators are particularly interesting if m is a prime number because their maximum period is then $m^k - 1$, and this maximum is reached, see Theorem 13. Hence, it is possible, even with a small modulo, to have very large periods.

For example, in order to generate random sequences of bits, one chooses $m = 2$. In this case, the operations can be performed very quickly on a machine with "eXclusive ORs" (xor) for the addition modulo 2 and with shifts on the bits x_i to generate the next bits. There even exist specialized chips performing the necessary operations. Then, one talks about Linear Feedback Shift Register (LFSR). Figure 1.10 summarizes their mechanisms.

For some computations, it is interesting to write an LFSR in a polynomial form: set $\Pi(X) = X^k - a_1 X^{k-1} - \ldots - a_k$.

Hence, $LFSR_{\Pi}(x_0, \ldots, x_{k-1})$ refers to the infinite sequence of bits x_i linearly generated by the polynomial Π, having the first k initial values set to $x_0, \ldots, x_{k-1}$.

Exercise 1.30 *Write the first eight values generated by the following shift register:*

$$LFSR_{X^4+X^3+X^2+1}(0, 1, 1, 0)$$

Solution on page 290.

Finally, we have the equivalent of Theorem 12 for LCG with the primitive root replaced by a primitive polynomial.

Theorem 13 *For some polynomial Π of degree k, the $LFSR_{\Pi}$ modulo a prime number p is of maximum period $p^k - 1$ if and only if Π is a primitive polynomial in $\mathbb{F}_p[X]$.*

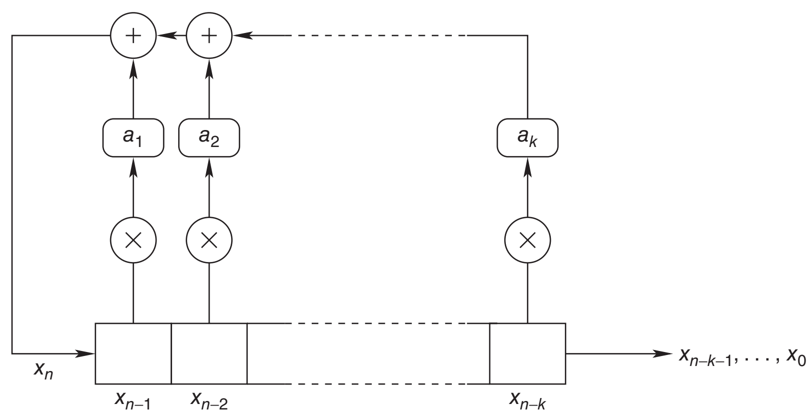

Figure 1.10 Functional diagram of an LFSR

These generators are quite fast. Besides, they have also a very large period. However, we will see in Section 1.4.3.2 that the Berlekamp–Massey algorithm enables one to predict the following bits without knowing the generator polynomial, provided that $2k$ successive values have been intercepted.

These generators are used in practice to generate quickly some bits with good statistical properties but they have to be combined with other generators to be cryptographically secure.

Example 1.9 (Securing the *Bluetooth* Protocol) *Bluetooth* is a short-range wireless technology whose aim is to simplify the connections between electronic equipment. It was designed to replace the wires between computers and their devices such as printers, scanners, mice, cell-phones, pocket-PCs, or even numerical cameras.

In order to make this protocol safe, one uses some kind of Vernam encryption scheme (Section 1.2.1) but with a pseudo-random generator based on LFSR: the encryption algorithm uses four LFSRs of respective length 25, 31, 33, and 39 for an overall $25 + 31 + 33 + 39 = 128$ bits. The 128 bits of the initial value represent the secret key of *Bluetooth* encryption. Figure 1.11 shows the functional diagram of this encryption scheme.

We notice that the four polynomials that are used are as follows:

- $X^{39} + X^{36} + X^{28} + X^4 + 1$;
- $X^{33} + X^{28} + X^{24} + X^4 + 1$;
- $X^{31} + X^{24} + X^{16} + X^{12} + 1$;
- $X^{25} + X^{20} + X^{12} + X^8 + 1$.

These four polynomials are primitive polynomials in $\mathbb{F}_2[X]$ for an overall period of $\text{lcm}(2^{39} - 1, 2^{33} - 1, 2^{31} - 1, 2^{25} - 1) = 7 \cdot 23 \cdot 31 \cdot 79 \cdot 89 \cdot 601 \cdot 1801 \cdot 8191 \cdot 121369 \cdot 599479 \cdot 2147483647 \approx 2^{125}$.

The 4 bits $\{a; b; c; d\}$ produced by these four successive LFSRs are then combined using a nonlinear discrete function f which produces the next bit z_t on the output, from its initial state $(\{cl_{-1}; ch_{-1}; cl_0; ch_0\} = IV \in \{0, 1\}^4)$ and the successive values

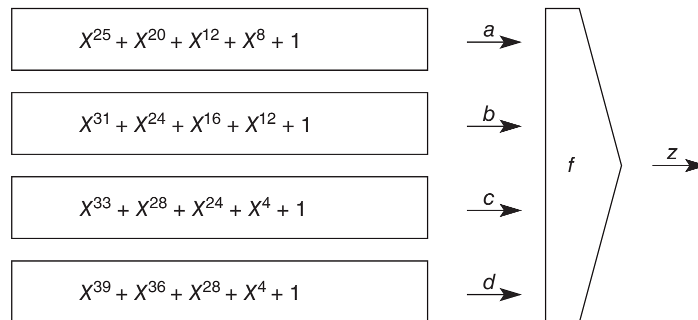

Figure 1.11 *Bluetooth* encryption

of the LFSR, according to the following algorithm:

1. $z_t = a_t \oplus b_t \oplus c_t \oplus d_t \oplus cl_t$ (operations over $\mathbb{F}_2$);
2. $s_{t+1} = \lfloor \frac{a_t+b_t+c_t+d_t+2ch_t+cl_t}{2} \rfloor \in [0, 3]$ (operations over $\mathbb{Z}$);
3. sl_{t+1} and sh_{t+1}) are the 2 bits encoding $s_{t+1} \in [0, 3]$;
4. $cl_{t+1} = sl_{t+1} \oplus cl_t \oplus cl_{t-1} \oplus ch_{t-1}$ (operations over $\mathbb{F}_2$); and
5. $ch_{t+1} = sh_{t+1} \oplus ch_t \oplus cl_{t-1}$ (operations over $\mathbb{F}_2$).

1.3.7.3 Cryptographically Secure Generators One can use the principle of a one-way function, namely a function easy to compute but difficult to invert (computation in unreasonable time), to determine the quality of a generator.

The formal definition of a good quality generator is as follows. Given a generator and a finite sequence of bits it has generated, if it is possible, without knowing the method, to predict with good probability and in reasonable time the next bit of the sequence, then the generator cannot be considered as a random generator. Here, a good probability means significantly greater than a random guess, that is, $\frac{1}{2}$ for a sequence of bits.

If one is able to prove that there exists no efficient algorithm performing this prediction, then the generator is called *cryptographic* or *cryptographically secure*.

For example, the Blum–Micali generator proceeds in the following way:

- Generate a large prime number p.
- Let α be a primitive root of p (a generator of the group of invertible elements in $\mathbb{F}_p$).
- Let f be the modular exponentiation function $f(x) = \alpha^x \mod p$.
- Let B be the function with values in $\{0, 1\}$ defined by:
  - $B(x) = 1$ if $0 \leq log_\alpha x \leq (p-1)/2$;
  - $B(x) = 0$ if $log_\alpha x > (p-1)/2$.

The pseudo-random sequence of bits $b_1 \dots b_k$ is then computed from the sequence $x_0, x_1, \dots, x_k$, with x_0 any nonzero element in $\mathbb{F}_p$ and $x_i \leftarrow f(x_{i-1})$ for $i > 0$. One sets $b_i = B(x_i)$.

The function B is easy to compute: we have seen in the previous subsections how to generate a prime number, how to find a primitive root, and how to carry out modular exponentiation. Finally, when computing B, one has the value of $log_\alpha x$ using $f(x) = \alpha^x \mod p$ that has just been calculated. However, without the sequence $x_0, x_1, \dots, x_k$, one can prove that finding $B(x_i)$ is as difficult as computing the discrete logarithm. As we do not know any efficient algorithm solving the discrete logarithm problem, it is a difficult problem. Therefore, the generator is cryptographically secure.

1.3.7.4 Several Statistical Tests The previous methods are based on the well-known algorithmic difficulty of some problems, which makes it impossible to predict the behavior of generators. In order to measure the quality of a generator,

one can also examine the sequences generated, and test whether they diverge from what we expect from a truly random generator. This is a difficult task as the criteria are numerous and are not necessarily trivial.

Statistics provide us with an adequate tool for these tests. For example, the χ^2 test enables one to measure the deviance with respect to an expected uniform discrete law.

For all characters v_i in the alphabet V, one has the expected probability p_i and the number e_i of occurrences in the generated sequence of size n. The expected frequencies are never exactly equal to the empiric frequencies. Therefore, one has to set the threshold of divergence from which the generator is no longer considered as random.

One has to keep in mind that, when considering a random generator, all sequences are possible *a priori*, even those whose distribution is completely different from the expected one because the generator is actually random. These sequences are only very unlikely to appear. If one observes such sequences in the output of a generator, then the generator is probably not so good (even if these sequences can theoretically appear in the output of a good generator). Here is how the χ^2 test works.

One measures the gap between the expected distribution and the observed distribution using the parameter:

$$K = \sum_{i=1}^{n} \frac{(e_i - np_i)^2}{np_i}.$$

Now, it remains to determine the acceptable values for the parameter K. They are given by the tables of the χ^2, of which we give an extract in Table 1.7.

In this table, the first column gives the number of "degrees of liberty." One sets this number to the value $|V| - 1$. Namely, for an alphabet of size 21, the line number 20. The first line gives the probability of having the value K lower than the value of the table. For example, the probability of having K greater than 24.72 for an alphabet of size 12 is 0.01.

Exercise 1.31 (χ^2 test) *A pseudo-random generator which is supposed to generate numbers between 0 and 10 according to a uniform law gives the sequence: 0 0 5 2 3 6 4 2 0 2 3 8 9 5 1 2 2 3 4 1 2.*

TABLE 1.7 χ^2 Table (Extract)

Degrees of liberty	$p = 0.75$	$p = 0.95$	$p = 0.99$
9	11.39	16.92	21.67
10	12.55	18.31	23.21
11	13.70	19.68	24.72
12	14.85	21.03	26.22
15	18.25	25.00	30.58
20	23.83	31.41	37.57
30	34.80	43.77	50.89

Perform the χ^2 test on this sequence. What do you think of this generator? What do you think of the test? *Solution on page 290.*

Obviously, such a test – although it can be useful and sufficient to reject a generator – is not sufficient to accept a generator. For example, it will not be able to reject the sequence 123456123456123456, whereas a not-so-drilled eye will notice the regularity in it (although one should distrust the impression of regularity one can have looking at a sequence, as it can be biased by false intuitions on what is actually true randomness).

One can, for example, strengthen this test by applying it to the extensions of the source induced by the message (Section 1.2.3.4). There exist numerous statistical tests reinforcing the trust one could have concerning a generator. It is important to notice that each test enables one to reject a generator, but only the set of all tests will enable one to accept it (besides, without mathematical rigor). There is no guarantee that – after succeeding in x tests – the generator will not reveal itself to be weak under the $(x + 1)$th test.

1.4 DECODING, DECRYPTION, ATTACKS

To conclude this introductory chapter, we develop encoding methods adopting the point of view of the inverse operation, decoding. We have already seen that, for many reasons, decoding – which consists in inverting the encoding functions – is not a trivial task:

- as in the fax code we detailed at the beginning of this chapter, if the ciphertext is a sequence of bits, recovering the source message without ambiguity by parsing the sequence into blocks requires a particular form of the code;
- an exact decoding is sometimes not even completely attainable, if the encoding method does not include all the initial information, when performing a compression for example; we have seen that the fax image loses in quality; there exist many encoding methods "with loss," which makes encoding and decoding asymmetrics;
- we have seen that the principle of one-way functions makes the computation of the encoding function and the decoding function completely different; it may happen that there is an efficient algorithm for one of them but not for the other.

We are going to develop all these aspects, using the word *decoding* as a general term allowing one to recover the source from a codeword, the word *decryption* for cryptographic decoding, and the words *breaking* or *attack* for an unauthorized decryption, namely when the recipient is the only one possessing the information.

1.4.1 Decoding without Ambiguity

The first virtue of a code is its ability to be decoded. This is obvious, but not necessarily a trivial issue.

Let us suppose that the code is a bijective function, which transforms the message written by the sender into a message transmitted through the channel. For a source message $a_1...a_n$, a string over any source alphabet, and for a code alphabet V, let us denote by f the encoding function. Then, one has the codeword $c_1...c_n = f(a_1)...f(a_n)$, with $c_i \in V^+$ for all i. The *code*, seen as the set of all codewords, is then the image of the encoding function f. However, f being bijective is not enough for the message to be decoded without ambiguity by the recipient.

As an example, let us consider the encoding of the 26 alphabet letters $S = \{A, \ldots, Z\}$ using integers $C = \{0, \ldots, 25\}$ written in base 10:

$$f(A) = 0,\ f(B) = 1, \ldots, f(J) = 9,\ f(K) = 10,\ f(L) = 11, \ldots, f(Z) = 25.$$

Then, the codeword 1209 may correspond to several messages: for example, BUJ, MAJ, or BCAJ.

Thus, in order to avoid such a problem, one has to add some constraints on the code for any message to be decoded without ambiguity. That is to say, when receiving a codeword, the recipient has to be able to recover a *unique* message from it. A code C over an alphabet V is called *nonambiguous* (one sometimes says *uniquely decodable*) if, for all $x = x_1 \ldots x_n \in V^+$, there exists at most one sequence $c = c_1 \ldots c_m \in C^+$ such that

$$c_1 \ldots c_m = x_1 \ldots x_n.$$

The following property is just a simple reformulation:

Property 8 *A code C over an alphabet V is nonambiguous if and only if for all sequences $c = c_1 \ldots c_n$ and $d = d_1 \ldots d_m$ in C^+:*

$$c = d \Longrightarrow (n = m \text{ and } c_i = d_i \text{ for all } i = 1, \ldots, n).$$

Example 1.10 (Over the Alphabet $V = \{0, 1\}$)

- the code $C = \{0, 01, 001\}$ is not uniquely decodable.
- the code $C = \{01, 10\}$ is uniquely decodable.
- the code $C = \{0, 10, 110\}$ is uniquely decodable.

The decoding constraint implies that all codewords should have a minimum length. Kraft's theorem gives a necessary and sufficient condition on the length of codewords to insure the existence of a uniquely decodable code.

Theorem 14 (Kraft) *There exists a uniquely decodable code over some alphabet V with n codewords of length $l_1, \ldots, l_n$ if and only if*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leq 1.$$

Proof. ($\Rightarrow$) Let C be a uniquely decodable code, of arity q (the vocabulary of the code contains q characters). Let m be the length of the longest word in C. For $1 \leq k \leq m$, let r_k be the number of words of length k. One develops the following expression, for any integer u, with $u \geq 1$:

$$\left(\sum_{i=1}^{n} \frac{1}{q^{l_i}}\right)^u = \left(\sum_{k=1}^{m} \frac{r_k}{q^k}\right)^u .$$

Once developed, each term of this sum is of the form $\frac{r_{i_1} \dots r_{i_u}}{q^{i_1+\dots+i_u}}$. Then, by regrouping for each value $s = i_1 + \dots + i_u$, one obtains the terms $\sum\limits_{i_1+\dots+i_u=s} \frac{r_{i_1} \dots r_{i_u}}{q^s}$. Set $N(s) = \sum_{i_1+\dots+i_u=s} r_{i_1} \dots r_{i_u}$. The initial expression can be written as follows:

$$\left(\sum_{i=1}^{n} \frac{1}{q^{l_i}}\right)^u = \sum_{s=u}^{mu} \frac{N(s)}{q^s}.$$

Notice that $N(s)$ is the number of combinations of words in C whose overall length is equal to s. As C is uniquely decodable, two combinations of words in C cannot be equal to the same word over the alphabet of C. As C is of arity q, and $N(s)$ is lower than the overall number of messages of length s on this alphabet, one has $N(s) \leq q^s$. This implies that

$$\left(\sum_{i=1}^{n} \frac{1}{q^{l_i}}\right)^u \leq mu - u + 1 \leq mu.$$

Thus, $\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leq (mu)^{1/u}$, and $\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leq 1$ when u tends toward infinity. ($\Leftarrow$) The reciprocal proposition is a consequence of McMillan's theorem, which is studied later on in this chapter.

□

1.4.1.1 Prefix Property One says that a code C over an alphabet V has the *prefix property* (one sometimes says that it is *instantaneous*, or *irreducible*) if and only if for all pairs of distinct codewords (c_1, c_2), c_2 is not a prefix of c_1.

Example 1.11 $a = 101000$, $b = 01$, $c = 1010$: b is not a prefix of a. However, c is a prefix of a.

If the prefix property applies, one is able to decode the words of such a code as soon as one has received the whole word (instantaneousness), which is not always the case with uniquely decodable codes: for instance, if $V = 0, 01, 11$, and one receives message $m = 001111111111 \dots$. Then one will have to wait for the next occurrence of a 0 to be able to decode the second word (0 or 01?).

Property 9 *Any code having the prefix property is uniquely decodable.*

Proof. Let C be a code over V that is not uniquely decodable and has the prefix property. Then there exists a string $a \in V^n$ such that $a = c_1 \dots c_l = d_1 \dots d_k$, with c_i and d_i codewords of C and $c_i \neq d_i$ for at least one index i. Let us choose the minimum index i such that $c_i \neq d_i$ (for all $j < i$, $c_j = d_j$). Then $length(c_i) \neq length(d_i)$, otherwise, given the choice of i, one would have $c_i = d_i$, which contradicts the definition of i. If $length(c_i) < length(d_i)$, then c_i is a prefix of d_i. Otherwise, d_i is a prefix of c_i. Thus, C does not have the prefix property. □

The reciprocal proposition is false: indeed, the code $C = \{0, 01\}$ is uniquely decodable but it does not have the prefix property. The following property is obvious, but it insures the decoding ability for some widely used kinds of codes.

Property 10 *If all the words of some code are of the same length, then it has the prefix property.*

Exercise 1.32 *Let S be the source of alphabet $\{a, b, c, d\}$ with probabilities:*

S	a	b	c	d
$P(S)$	0.5	0.25	0.125	0.125

One encodes S using the following codes:

a	b	c	d
0	10	110	111

1. *Encode adbccab. Decode 1001101010.*
2. *Is it an instantaneous code?*
3. *Compute the entropy of the source.*

Solution on page 291.

Exercise 1.33 *We wish to build a binary compression code over a source $\mathcal{S} = (S, \mathcal{P})$ (supposed to be infinite) where $S = (0, 1)$ is the source alphabet and $\mathcal{P} = (P(0) = p, P(1) = 1 - p)$ is the probability law of $\mathcal{S}$.*

One proposes the following code: one enumerates the number of occurrences of "0" before the appearance of "1." The two encoding rules are as follows:

- *A string of four consecutive "0"s (without "1") is encoded with 0.*
- *If less than four "0s" appear before a symbol "1," one encodes the string with the codeword "$1e_1e_2$," e_1e_2 being the binary representation of the number of "0s" before the symbol "1."*

For instance, the appearance of four consecutive zeros "0000" is encoded with "0," whereas the string "001" is encoded with "110" because two "0"s appear before the symbol "1" (and "10" is the binary representation of 2).

1. *Write explicitly the five codewords of this compression code. Does this code have the prefix property?*

2. *Knowing that the probability of appearance of two successive symbols s_1 and s_2 is – when supposing that the source is without memory – $p(s_1) * p(s_2)$, compute the probability of occurrence in $\mathcal{S}$ of a string composed of k "0"s followed by a "1."*
3. *For each codeword, compute the number of bits of code required per bit of the source. Deduce the* compression rate *of this code, namely the* mean length per bit of the source.

Solution on page 291.

1.4.1.2 Huffman Trees A Huffman tree is an object that enables one to easily represent all codes having the prefix property, and this representation makes their manipulation a lot easier. Here, we give the definitions in the binary case. However, these can be extended to codes of any arity.

One calls a *Huffman tree* a binary tree such that any subtree has either 0 or 2 sons (the tree is locally complete). One assigns the symbol "1" to the edge connecting the local root to the left subtree and "0" to the edge connecting the local root to the right subtree.

To each leaf of a Huffman tree, one can associate a word in $\{0, 1\}^+$: it is a string composed of the symbols marking the edges of the path from the root to the leaf.

The maximum length of the words in a Huffman tree is called the *depth* of the tree. One calls a *Huffman code* the set of words corresponding to the paths in a Huffman tree; the depth of this tree is also called *depth* of the code C.

Example 1.12 (Code corresponding to the tree of Figure 1.12)

$$\{111, 110, 10, 0111, 0110, 010, 001, 0001, 0000\}.$$

1.4.1.3 Representation of Instantaneous Codes The introduction of Huffman trees is justified by the two following properties, which enable one to manipulate instantaneous codes with such trees.

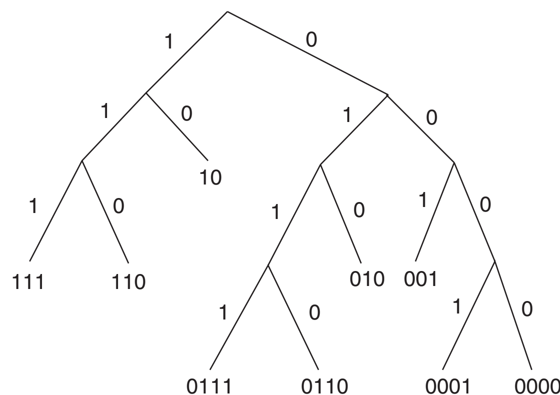

Figure 1.12 Example of a Huffman tree

Property 11 *A Huffman code has the prefix property.*

Proof. If a codeword c_1 is a prefix of c_2, then the path representing c_1 in the Huffman tree is included in the path representing c_2. As c_1 and c_2 are, by definition, associated with the leaves of the tree, $c_1 = c_2$. Thus, there do not exist two different codewords such that one of them is a prefix of the other. Hence, the Huffman code has the prefix property. □

Property 12 *Any code having the prefix property is included in a Huffman code.*

Proof. Let us consider a complete Huffman tree (all leaves are at the same distance from the root) of depth l (the length of the longest word in C). Each codeword c_i in C is associated to a path from the root to a node. Then, one can prune the subtree having this node as a root (all the words that could be represented in the nodes of this subtree would have c_i as a prefix). All other codewords in C remain in the nodes of the resulting tree. It is possible to perform the same operation for all the other words. One eventually obtain a Huffman tree containing all codewords in C. □

1.4.1.4 McMillan's Theorem We have seen that Huffman trees enable one to represent all instantaneous codes. However, they do not enable one to represent all uniquely decodable codes. McMillan's theorem insures that one can avoid the use of uniquely decodable codes (nonambiguous) not having the prefix property. Indeed, there always exists another code that has the prefix property with the same lengths of codewords. Therefore, nothing can be gained by using ambiguous codes.

Theorem 15 (McMillan) *Over an alphabet V, there exists a code having the prefix property whose codewords $\{c_1, \ldots, c_n\}$ are of length $l_1, \ldots, l_n$ if and only if*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leq 1 .$$

Exercise 1.34 *Using the representation of instantaneous codes with Huffman trees, give a proof of McMillan's theorem.* *Solution on page 291.*

Corollary 1 *If there exists a uniquely decodable code whose words are of length $l_1, \ldots, l_n$, then there exists an instantaneous code whose words are of the same length.*

This is a consequence of Kraft's and McMillan's theorems. All decodable codes not having the prefix property do not produce codes with shorter words than instantaneous codes; therefore, one can limit oneself to the latter codes for information compression (their properties make them easier to use).

1.4.2 Noninjective Codes

All codes do not insure a nonambiguous decoding and are not even bijective. It might happen, for several reasons, that encoding functions only process a *digest* (or a *fingerprint*) of a message, or only the information which is considered to be sufficient

for the needs of transmission. For instance, fingerprints are used for error detection: when receiving a message and its fingerprint, the recipient is able to check that the overall message has not been modified during the transmission by recomputing the fingerprint from the message he has received and comparing it to the fingerprint that was transmitted.

Lossy compression is used, for example, in processing images or sounds. The information is encoded in a way that will enable one to retrieve maybe only a variation of the original data. The differences should be slight enough so that they are not perceptible (for human ear or eye) or so that the new data is still useful.

1.4.2.1 Fingerprint Integrity Check The most simple principle of error detection is an example of fingerprint computation. We have seen an example of this kind of encoding with the fax code, even if the code we added to each line did not depend on the content of the line. Besides, this only had a limited detection capacity.

The first principle of fingerprints for error detection is the addition of a simple *parity bit* to the cipherblocks. For a word $m = s_1 \dots s_k$, the parity bit is equal to $b(m) = (\sum_{i=1}^{k} s_i) \mod 2$. Obviously, this equality is false when an odd number of bits change their value in the set "message+fingerprint." Hence, the addition of a parity bit enables one to detect errors on a odd number of bits. We will see this mechanism in more detail in Chapter 4, in particular in Figure 4.2 on page 212.

1.4.2.2 Hash Functions The hash function follows the same principle, but it encodes a more evolved fingerprint, as it is meant to identify the message. This is the definition of a summary of the message, which will enable one not to recover it, but to identify it using a correspondence table. This works as a human fingerprint, which does not enable one to reconstitute the other characteristics of an individual but which enables one to identify him.

Hash functions are particularly useful in cryptography. They notably enable one to decrease the amount of information to be encrypted. If the image of x by the hash function is called the *fingerprint* of x, one can – for example – encrypt only the fingerprint. Moreover, they enable one to set up electronic signature and message authentication protocols (see Section 3.5.3) and also to check the integrity of a document, in the same way as the parity bit (which is a particular hash function). Formally, a *hash function* $H : \{0,1\}^* \longrightarrow \{0,1\}^n$ is an application that transforms a string of any size into a string of fixed size n, as illustrated in Figure 1.13.

One talks about a *collision* between x and x' when

$$\begin{cases} x \neq x' \\ H(x) = H(x'). \end{cases}$$

Considering that the input of a hash function can be of any size (in particular $> n$), collisions are inevitable. If y is such that $y = H(x)$, then x is called the *preimage* of y (one recalls that y is the *fingerprint* of x).

One of the basic constraints in setting up a hash function is efficiency: a fingerprint must be easy to compute. Besides, hash functions have a natural compression property.

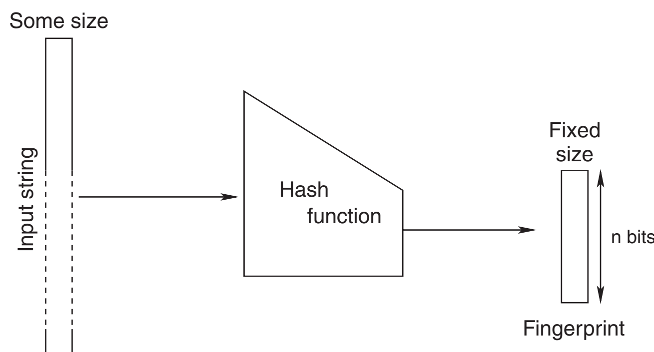

Figure 1.13 Principle of a hash function

Other properties can be expected:

- **Preimage resistant**: given y, one cannot find – in reasonable time – some x such that $y = H(x)$.
- **Second preimage resistant**: given x, one cannot find – in reasonable time – $x' \neq x$ such that $H(x) = H(x')$;
- **Collision resistant**: one can not find in reasonable time – x and x' such that $H(x) = H(x')$;

A *one-way* hash function is a hash function satisfying the properties of preimage resistance and second preimage resistance.

Exercise 1.35 (Security of Hash Functions) *Prove, using the contrapositive proposition, that collision resistance implies second preimage resistance, which implies preimage resistance.* *Solution on page 292.*

Exercise 1.36 (A Bad Hash Functionxs) *Let $f : \mathbb{F}_2^m \to \mathbb{F}_2^m$ be any function. One proposes an iterative hash function $g : \mathbb{F}_2^{2m} \to \mathbb{F}_2^m$, such that, for some x of size $2m$ bits, divided into two blocks x_h and x_l, one has $g(x) = g(x_h||x_l) = f(x_h \oplus x_l)$ where $x_h||x_l$ is the concatenation of x_h and x_l. Prove that g is not second preimage resistant.* *Solution on page 292.*

Hash functions can be used for

- Manipulation Detection Code (MDC) that enable one to check the integrity of a message (in the manner of parity bits);
- MAC that manage both the integrity and the authentication of the source of data.

We will see several examples of such applications in Chapter 3.

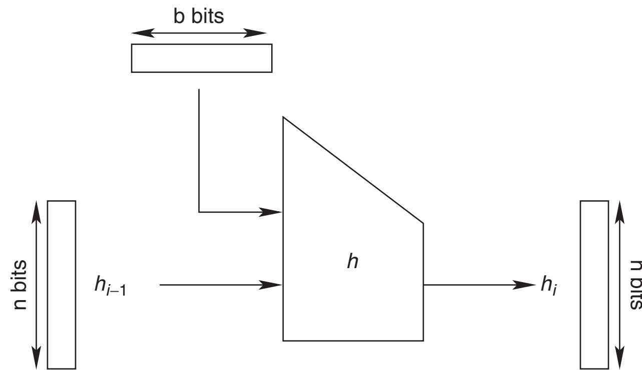

Figure 1.14 Compression function of a hash function

Construction of a Merkle–Damgård hash function. One of the most famous constructions of hash functions relies on a compression function

$$h : \{0,1\}^b \times \{0,1\}^n \longrightarrow \{0,1\}^n.$$

Such a function is illustrated in Figure 1.14.

Message M is split into blocks of b bits $M_1, \dots, M_k$ (one will possibly have to add some padding bits for the size of M to be divisible by b).

One iterates the compression function h according to the scheme presented in Figure 1.15.

IV (Initial Value) is a string (of size n) fixed by the algorithm or the implementation. A theorem – which was proved independently by Ralph Merkle and Ivan Damgård – enables one to describe the theoretical properties of such a construction:

Theorem 16 (Merkle–Damgård) *If h is collision resistant, then so is H (Figure 1.15).*

It is this result that actually makes Merkle–Damgård construction the most used construction in fingerprint computation.

Exercise 1.37 *Prove Merkle–Damgård's theorem using the contrapositive proposition.* *Solution on page 292.*

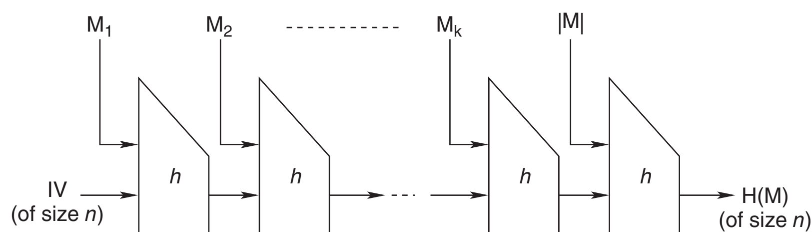

Figure 1.15 Merkle–Damgård construction

Exercise 1.38 (Construction by Composition) *Let $f : \mathbb{F}_2^{2m} \to \mathbb{F}_2^m$ be a hash function and let $h : \mathbb{F}_2^{4m} \to \mathbb{F}_2^m$ another hash function such that, if $x_1, x_2 \in \mathbb{F}_2^{2m}$, then $h(x_1||x_2) = f(f(x_1)||f(x_2))$, $||$ standing for the concatenation operation.*

1. *Prove that if f is collision resistant, then so is h.*
2. *What is the drawback of this construction ?*

Solution on page 293.

Hence, one only needs to make explicit the construction of compression functions h, which are collision resistant. For instance, the Davies–Meyer construction (Figure 1.16) defines $h_i = E_{M_i}(h_{i-1}) \oplus h_{i-1}$, where E_{M_i} is a symmetric block encryption function.

But an attack on preimage resistance was set up by Drew Dean in 1999, who exploited the existence of fixed points in this construction. Therefore, compression functions using this construction are less robust.

The Miyaguchi–Preneel construction (Figure 1.17) is an improvement on the previous construction and is particularly robust from a cryptographic point of view.

Function g adapts the construction to the size of the key of the encryption function E. Hence, one has $h_i = E_{g(h_{i-1})}(M_i) \oplus h_{i-1} \oplus M_i$.

Galois hashing. Another popular hash function is GHASH, for Galois hashing, which uses multiplication in the field with 2^{128} elements and Horner scheme. The idea is to choose an element h of $\mathbb{F}_{2^{128}}$ where the field is usually build as polynomials modulo 2 and modulo the primitive polynomial $X^{128} + X^7 + X^2 + X + 1$. Then, a message m is cut into $d + 1$ blocks m_i of 128 bits and each block is considered as a coefficient of the reverse polynomial $\sum m_{d-i} Y^i$. The hash value is obtained as the evaluation of this polynomial in h via Algorithm 1.7, where each block $m_i = b_0 \dots b_{127}$ is considered

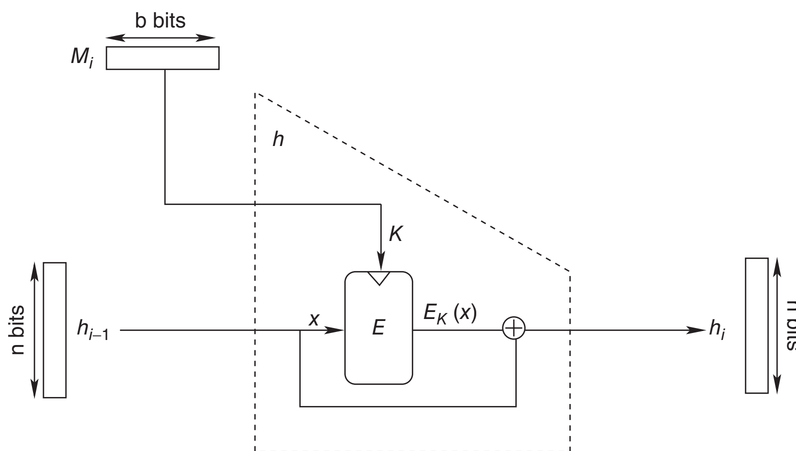

Figure 1.16 Davies–Meyer construction

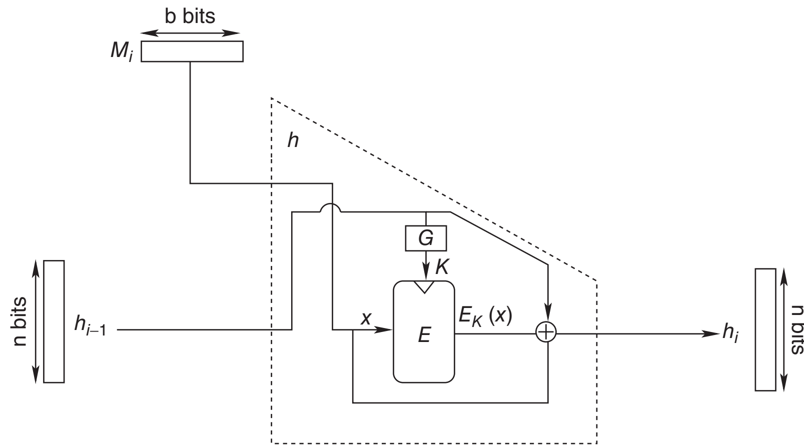

Figure 1.17 Miyaguchi–Preneel construction

as the element $m_i(X) = \sum_{j=0}^{127} b_j X^j \in \mathbb{F}_{2^{128}} = \mathbb{F}_2[X]/\left(X^{128} + X^7 + X^2 + X + 1\right)$:

$$GHASH_h \ : (m_0, \ldots, m_d) \hookrightarrow \sum_{i=0}^{d} m_{d-i} h^i \in \mathbb{F}_{2^{128}}.$$

Exercise 1.39 (Security of GHASH)

1. *Suppose that there exist i and j with $i < j$ such that for the chosen element of the $GHASH_h$ multiplication, we have $h^i \equiv h^j \in \mathbb{F}_{2^{128}}$. Deduce a way to build a collision in the hash function.*
2. *We know that* $2^{128} - 1 = 3 \cdot 5 \cdot 17 \cdot 257 \cdot 641 \cdot 65537 \cdot 274177 \cdot 6700417 \cdot c_{14}$*, with* $c_{14} = 67280421310721$. *How many possible distinct orders are there for elements of $\mathbb{F}_{2^{128}}$?*
3. *For a given h, with a given order o, what size of message would be required to obtain a collision using the first question?*
4. *If you were to choose an element h for the hashing, which elements would be best?*
5. *For a randomly chosen non zero element h, what is the probability that c_{14} does not divide its order?*
6. *If this is not the case what size of message would be required to obtain a collision using the first question?*

Solution on page 293.

1.4.2.3 Lossy Transformations Other transformation processes end with a message encoded with loss and make full decoding impossible. For example, a fax is just a bad copy of the original image, but one sees that it fulfills its duty of transmitting the

information contained in the document. One can also encode an image (respectively a sound) without keeping all the information, provided that the loss is not noticeable to the naked-eye (respectively to the ear).

Numerical information is, in essence, discrete information. We have seen that continuous or analogical data, like sounds and images, can be easily digitized. As for sounds, one can encode at each instant a frequency and an amplitude. For images, one decomposes them into pixels and encodes a color for each pixel.

Yet, this natural digitization might not be the best model for many codes. In images, for example, the colors of two contiguous pixels are often not independent, and it will be more judicious to encode a set of pixels as a function rather than a single pixel as a value. Therefore, one encodes blocks of pixels with a periodic (or almost periodic) function.

Hence, encoding happens to be performed on functions rather than on discrete or numerical entities, and this is the principle of the following section. Therefore, encoding will be a particular function, which will be applied to functions, and that we will call rather a *transform*, or a *transformation*.

1.4.2.4 Fourier Transform and Discrete Fourier Transform (DFT) Let us suppose that a message, or part of a message, can be formulated as a periodic and integrable function h (more precisely h should be in $\mathcal{L}^1(\mathbb{R})$), varying with respect to time t, and of period 2π. This happens with sounds. As we assume that h is periodic, the same message can be formulated as an amplitude H, varying with respect to a frequency f. The Fourier transform is an encoding process that enables one to switch from one representation another. Like any encoding process, it is associated to its inverse, decoding, and is formulated with the following formulas (Figure 1.18).

For a sound, even if the natural and immediate encoding is rather $h(t)$, one often uses $H(f)$ that encodes exactly the same information – as encoding is reversible – and is a lot cheaper, because it makes good use of the periodicity of h. Therefore, the Fourier transform is very efficient for compression.

The DFT follows the same principle but with discrete functions. This will obviously be very useful as – by essence – our numerical messages have to be encoded with discrete information. Now let us suppose that our functions $h(t)$ and $H(f)$ are discrete functions, that is to say some vectors $h_0, \dots, h_{n-1}$ and $H_0, \dots, H_{n-1}$ with discrete variables. One formulates the transformation by denoting $\omega = e^{-\frac{2i\pi}{n}}$ an nth root of unity. ω satisfies the equalities: $\sum_{k=0}^{n-1} \omega^k = \sum_{k=0}^{n-1} \left(e^{-\frac{2i\pi}{n}}\right)^k = \frac{e^{-\frac{2in\pi}{n}}-1}{e^{-\frac{2i\pi}{n}}-1} = 0$.

$$\text{Encoding: } H(f) = \int_{-\infty}^{+\infty} h(t)e^{-2i\pi ft}dt$$

$$\text{Decoding: } h(t) = \int_{-\infty}^{+\infty} H(f)e^{2i\pi ft}df$$

Figure 1.18 Fourier transform

Encoding: $H_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} h_j \omega^{kj}$

Decoding: $h_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} H_k \omega^{-kj}$

Figure 1.19 Discrete Fourier Transform (DFT)

In other words following Figure 1.19, if $h(X) = \sum_{j=0}^{n-1} h_j X^j$, then

$$DFT(h) = [H_0, \dots, H_{n-1}] = \frac{1}{\sqrt{n}}[h(\omega^0), \dots, h(\omega^{n-1}).] \tag{1.14}$$

Decoding is correct as

$$h_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} h_i \omega^{ki} \omega^{-kj} = \frac{1}{n} \sum_{i=0}^{n-1} h_i \sum_{k=0}^{n-1} (\omega^{i-j})^k = \frac{1}{n} \sum_{i=0}^{n-1} h_i \times \begin{cases} 0 & \text{if } i \neq j \\ n & \text{otherwise.} \end{cases}$$

The Discrete Cosine Transform (DCT) is a direct consequence of the DFT for some discrete function h. But instead of being time-varying (which is a good model for a sound), it is space varying (which enables one to encode an image); hence, h is a two-variable discrete function $h(x, y)$. For instance, it is the color of a pixel whose coordinates are x and y. In the same way, it is possible to represent differently the same information with a two-variable discrete function $H(i, j)$ standing for a *spectral* analysis of the image. The DCT and its inverse are shown in Figure 1.20, where $c(u) = 1$ whenever $u \neq 0$ and $c(0) = \frac{1}{\sqrt{2}}$.

DCT is also a good compression principle for images, as for any periodic (or almost periodic, i.e., periodic up to a small error) message.

These transformations are reversible. Moreover, not only do they prove themselves to be good compression processes but their interest also lies in the easiness of choosing and keeping only important information. Indeed, during such encoding, it is possible to keep only some coefficients of the DFT or the DCT – to reduce the size of information one has to encode – while not necessarily changing the audio/visual result. We will handle these principles in more detail in Chapter 2.

Encoding: $H(i, j) = \frac{2}{n} c(i)c(j) \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} h(x, y) \cos\left(\frac{(2x+1)i\pi}{2n}\right) \cos\left(\frac{(2y+1)j\pi}{2n}\right)$

Decoding: $h(x, y) = \frac{2}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c(i)c(j) H(i, j) \cos\left(\frac{(2x+1)i\pi}{2n}\right) \cos\left(\frac{(2y+1)j\pi}{2n}\right)$

Figure 1.20 Discrete Cosine Transform (DCT)

1.4.2.5 DFT Algorithm One can write the DFT transformation as a matrix $H = \Omega h$ with $\Omega_{k,j} = \frac{1}{\sqrt{n}}\omega^{kj}$. Thus, the inverse transformation can be written as $(\Omega^{-1})_{k,j} = \frac{1}{\sqrt{n}}\omega^{-kj}$.

Remark 2 *In some fields, $\sqrt{n}$ simply does not exist. It is therefore sometimes useful to define the transform with another constant factor: $\Omega = [\lambda\omega^{kj}]$ and $\Omega^{-1} = [\frac{\lambda}{n}\omega^{-kj}]$. For the sake of simplicity, in the following, we will avoid $\sqrt{n}$ and use $\lambda = 1$ so that $\Omega = [\omega^{kj}]$ and $\Omega^{-1} = [\frac{1}{n}\omega^{-kj}]$.*

An immediate algorithm for the calculation of this transform uses a matrix vector product and thus has a complexity of $O(n^2)$.

A "divide and conquer" algorithm decreases this complexity, which is extremely important for encoding. The "divide and conquer" principle is to split the problem into equivalent subproblems of lower size. Here, one divides the expression of the transform into two parts. Assuming that n is even, and setting $m = \frac{n}{2}$, one has

$$
\begin{aligned}
H_k = \mathrm{DFT}_{k;\omega}(h) &= \sum_{j=0}^{m-1} h_j\omega^{kj} + \sum_{j=m}^{n-1} h_j\omega^{kj} \\
&= \sum_{j=0}^{m-1} h_j\omega^{kj} + \sum_{j=0}^{n-m+1} h_{j+m}\omega^{k(j+m)} \\
&= \sum_{j=0}^{m-1} h_j\omega^{kj} + \sum_{j=0}^{m-1} h_{j+m}\omega^{kj}\omega^{km} \\
&= \sum_{j=0}^{m-1} (h_j + \omega^{km}h_{j+m})\omega^{kj}
\end{aligned}
$$

However, as ω is an nth root of unity, $\omega^{km} = (\omega^m)^k = (\omega^{n/2})^k = (-1)^k$ is equal to 1 or -1 according to the parity of k.

If k is even, then one defines the vector

$$\tilde{h}^p = (h_0 + h_m, \dots, h_{m-1} + h_{n-1})$$

and the even coefficients of H (the transform of h) are the coefficients of the transform $\widetilde{H}^p$ of $\tilde{h}^p$, which is half the size of h:

$$
\begin{aligned}
H_{2t} = \mathrm{DFT}_{2t;\omega}(h) &= \sum_{j=0}^{m-1} (h_j + h_{j+m})\omega^{2tj} \\
&= \mathrm{DFT}_{t;\omega^2}(\tilde{h}^{(p)}).
\end{aligned}
$$

Now, if k is odd, one defines the vector

$$\tilde{h}^{(i)} = (h_0 - h_m, (h_1 - h_{m+1})\omega, \dots, (h_{m-1} - h_{n-1})\omega^{m-1})$$

and the odd coefficients of H (the transform of h) are the coefficients of the transform $\widetilde{H}^i$ of $\tilde{h}^i$, which is half the size of h:

$$\begin{aligned} H_{2t+1} = \text{DFT}_{2t+1;\omega}(h) &= \sum_{j=0}^{m-1}(h_j - h_{j+m})\omega^{(2t+1)j} \\ &= \sum_{j=0}^{m-1}(h_j - h_{j+m})\omega^j \left(\omega^2\right)^j \\ &= \text{DFT}_{t;\omega^2}(\tilde{h}^{(i)}). \end{aligned}$$

One obtains Algorithm 1.12.

Algorithm 1.12 Discrete Fast Fourier Transform

Input Vector h whose size is a power of 2
Output Vector H: the transform of h

1: **If** the size of h is equal to 1 **then**
2:    **return** h
3: **else**
4:    Compute $\tilde{h}^{(p)} = (h_0 + h_m, \dots, h_{m-1} + h_{n-1})$;
5:    Compute $\tilde{h}^{(i)} = (h_0 - h_m, (h_1 - h_{m+1})\omega, \dots, (h_{m-1} - h_{n-1})\omega^{m-1})$;
6:    Compute recursively $\widetilde{H}^{(p)} = DFT_{\omega^2}(\tilde{h}^{(p)})$ of size $n/2$;
7:    Compute recursively $\tilde{H}^{(i)} = DFT_{\omega^2}(\tilde{h}^{(i)})$ of size $n/2$;
8:    The even coefficients of H are the coefficients of $\widetilde{H}^{(p)}$ and the odd coefficients of H are the coefficients of $\widetilde{H}^{(i)}$.
9: **End If**

The complexity of this algorithm is $C(n) = 2C(n/2) + 3n/2$, in consequence $C(n) = \frac{3}{2}n\log_2 n$. It is almost a linear complexity, thus an important improvement with respect to the initial algorithm. Moreover, the algorithm for the inverse Fourier transform is then straightforward, as

$$\text{DFT}_{\omega}^{-1} = \frac{1}{n}\text{DFT}_{\omega^{-1}}.$$

1.4.2.6 DFT and nth Roots of Unity in a Finite Field One considers the polynomial $X^n - 1$ in a field $\mathbb{F}_q$ with $n < q$. An *nth root of unity* in $\mathbb{F}_q$ is a simple root, if there exists such root, of the polynomial $X^n - 1$. The *order* of a root of unity γ is the least integer o such that $\gamma^o = 1$. As γ is a root of $X^n - 1$, one has obviously $o \leq n$. Besides, o divides n. Indeed, if one sets $n = ob + r$, then one has $\gamma^r = 1$. Thus $r = 0$.

A *nth primitive root of unity* is an nth root of unity of order n.

This notion is crucial for the application of the DFT: in order to compute the DFT in the field $\mathbb{C}$, we used a particular nth root – $e^{-\frac{2i\pi}{n}}$ – which is primitive in $\mathbb{C}$.

Now, nth primitive roots are available for any n in $\mathbb{C}$, whereas one is not even sure of their existence in a given finite field. Indeed, in $\mathbb{F}_q$, a $(q-1)$th primitive root of unity is what we simply called a primitive root in Section 1.3.5.3. In the same way as we did for these roots, the following theorem enables one to determine the fields in which it is possible to make fast calculations on vectors of a given size n:

Theorem 17 *Let q be a power of some prime number and let n be coprime with q. The finite field $\mathbb{F}_q$ contains an nth primitive root of unity if and only if n divides $q-1$.*

Proof. If a is an nth primitive root, then $a^n = 1$ and n is the order of a. As a is also an element of the field with q elements, its order necessarily divides $q-1$. Reciprocally, one uses a generator g of the field (a $(q-1)$th primitive root) whose existence is ensured by the algorithm in Section 1.3.5.3. Hence, if $q-1 = kn$ then g^k is an nth primitive root of unity. □

One says that a field *supports the DFT at order n* if there exist nth primitive roots of unity in this field. All fields supporting DFT for n equal to a power of 2 are obviously very interesting for applying the fast divide and conquer algorithm above. For instance, as we will see in Algorithm 1.13, the field with 786433 elements enables one to multiply polynomials of degree up to 2^{18} with the fast algorithm as $786433 - 1 = 2^{18} \cdot 3$.

Therefore, one has to compute such an nth root of unity. It is possible to use a generator, but one would rather use a variant of the algorithm presented in Section 1.3.5.3: draw randomly an nth root (a root of the polynomial $X^n - 1$ in $\mathbb{F}_q$) and test whether its order is actually n. The following corollary gives us the probability of success: $\varphi(n)$ chances over n.

Corollary 2 *If a finite field has at least one nth primitive root of unity, then it has exactly $\varphi(n)$ primitive roots.*

Proof. Let $q-1 = kn$. Let g be a generator of the field. Thus, g^k is an nth primitive root, as well as all g^{tk} for t between 1 and $n-1$ coprime with n. All these g^{tk} are distinct; otherwise g would not be a generator; and these are the only ones as g^{tk} with t not coprime to n is of order strictly lower than n. The nth primitive roots are necessarily written as g^{tk}: if g^u is an nth primitive root, then $g^{un} = 1$. As g is a generator, one has $un = t(q-1) = tkn$. This proves that u is in the form tk. □

Exercise 1.40 *Find a 6th primitive root of unity modulo 31.*

Solution on page 293.

However, if the field $\mathbb{F}_q$ does not contain an nth primitive root of unity, one may extend the field. In the same way as $\mathbb{C}$ with respect to $\mathbb{R}$, one can consider a field

containing $\mathbb{F}_q$ in which the polynomial $X^n - 1$ can be completely factorized into polynomials of degree 1. This field is an *extension* of the field $\mathbb{F}_q$ and it is called a *splitting field* of $X^n - 1$.

Exercise 1.41 *Find a* 4*th primitive root of unity in a field of characteristic* 31.
Solution on page 293.

1.4.2.7 Fast Product of Polynomials Using DFT As for the discrete Fourier transform (DFT), the product of two polynomials – which is often used in coding theory (see all constructions based on polynomials in this chapter) – has a naive algorithm of complexity bound $O(n^2)$.

The DFT and the calculation algorithm we have just seen enables one to perform this computation in time $O(n \log(n))$.

Given two polynomials $P = a_0 + a_1X + \cdots + a_mX^m$ and $Q = b_0 + b_1X + \cdots + b_nX^n$, one denotes by $A = DFT(P)$ and $B = DFT(Q)$ the respective discrete Fourier transform of vectors $a = a_0 \dots a_m 0 \dots 0$ and $b = b_0 \dots b_n 0 \dots 0$, where the coefficients of the polynomials are extended with zeros up to the degree $n + m$ (degree of the product). Then, from the definition of the transform, one can immediately write the coefficients as $A_k = \sum_{i=0}^{n+m} a_i \omega^{ki} = P(\omega^k)$ and $B_k = Q(\omega^k)$. By simply multiplying these two scalars and using the arithmetic of polynomials, one obtains the value of the product PQ evaluated at ω^k: $C_k = A_kB_k = P(\omega^k)Q(\omega^k) = (PQ)(\omega^k)$; in other words $C = DFT(PQ)$.

This property enables one to build Algorithm 1.13 that computes the product of two polynomials in time $O(n \log(n))$.

Algorithm 1.13 Fast product of two polynomials

Input Two polynomials $P = a_0 + a_1X + \cdots + a_mX^m$ and $Q = b_0 + b_1X + \cdots + b_nX^n$.

Output The product polynomial PQ.

1: Extend the polynomials with zeros up to the degree $m + n$ (degree of the product).
2: Compute the discrete Fourier transforms $DFT(P)$ and $DFT(Q)$ of the vectors of coefficients of P and Q, with Algorithm 1.12.
3: Using termwise multiplication, Compute vector $DFT(P).DFT(Q)$.
4: Compute the inverse transform in order to obtain $PQ = DFT^{-1}(DFT(P).DFT(Q))$.

The complexity is truly $O(n \log n)$, as termwise multiplication is linear and the Fourier transform – as well as its inverse - has a complexity $O(n \log n)$.

1.4.3 Cryptanalysis

We have studied some skewness properties between encoding and decoding. One of them, probably the most important one in cryptography, is that which differentiates

decryption (by the recipient) from breaking (by a third party). We dedicate a small part of this book to attack techniques based on the weaknesses of codes, which are developed too quickly.

Cryptographic codes use pseudo-random generators for secret key generation, hash functions for authentication, and one-way functions for public key techniques. We will present separately known attacks for each of these steps. Knowing these attack techniques is essential to generate codes that can resist them.

1.4.3.1 Attacks on Linear Congruential Generator Linear congruential generators have been looked at in Section 1.3.7 A random number x_i is generated as a function of the previously generated number x_{i-1}, using the formula $x_i = ax_{i-1} + b \mod m$.

Exercise 1.42 (Attack on LCGs)

- *If m is prime, what is the maximum period of an LCG? In particular, Fishman and Moore studied generators modulo* $2^{31} - 1 = 2147483647$ *in 1986. They determined that if* $a = 950706376$ *then the period is maximum and the generator has good statistical properties. What can you say about* 950706376*?*
- *For* $m = p^e$ *with p an odd prime, what is the maximum period of an LCG?* *One can prove that if* $\lambda(m)$ *is the maximum period, then* $\lambda(2^e) = 2^{e-2}$ *for* $e > 2$ *and that* $\lambda(m) = lcm(\lambda(p_1^{e_1}) \dots \lambda(p_k^{e_k}))$ *if* $m = p_1^{e_1} \dots p_k^{e_k}$ *with* $p_1, \dots, p_k$ *distinct primes.*
- *Assuming that m is known, how can one recover a and b?*
- *Now, suppose that m is unknown. How can one find the generator if* $b = 0$*?* Hint: one may study $x_{n+1} - x_n$. *What happens if* $b \neq 0$*?*
- *What is the next integer in this list: 577,114,910,666,107?*

Solution on page 294.

1.4.3.2 Berlekamp–Massey Algorithm for the Synthesis of LFSRs The Berlekamp–Massey algorithm enables one to detect, for an infinite sequence (S_i), $i \in \mathbb{N}$, of elements in a field $\mathbb{F}$, whether – beyond some rank – its elements are linear combinations of the previous elements. This is what we have called *linear recurrent sequences*.

This algorithm is very useful in coding theory, notably in order to perform the cryptanalysis of random generators and cryptographic keys and even to correct the errors of cyclic codes (Section 4.4.6.1). In particular, it enables one to recover the generator polynomial of an LFSR (Section 1.3.7.2) only knowing the first terms of the sequence generated by this LFSR.

The question is, for the sequence $(S_i)_{i \in \mathbb{N}}$, how to find coefficients $\Pi_0 \dots \Pi_d \in \mathbb{F}$, if they exist, such that

$$\Pi_0 S_t = S_{t-1}\Pi_1 + S_{t-2}\Pi_2 + ... + S_{t-d}\Pi_d \text{ for all } t \geq d.$$

If one uses these constants in order to define the polynomial $\Pi(X) = \Pi_d X^d + \Pi_{d-1}X^{d-1} + \Pi_{d-2}X^{d-2} + ... + \Pi_0$, this polynomial is called an annihilator of the sequence.

The set of annihilators is an ideal in the ring $\mathbb{F}[X]$ of polynomials over $\mathbb{F}$. As $\mathbb{F}[X]$ is a principal ideal ring, there exists a unitary annihilator polynomial of minimum degree, called the minimal polynomial of the sequence.

How does one find this polynomial only from the coefficients of the sequence? If one knows the degree d of this polynomial, one has to write d linear equations corresponding to the property of linear recurrence for $2d$ coefficients. Then one has to solve the following linear system:

$$\begin{bmatrix} S_0 & S_1 & \dots & S_{d-1} \\ S_1 & S_2 & \dots & S_d \\ \vdots & \iddots & \iddots & \vdots \\ S_{d-1} & S_d & \dots & S_{2d-1} \end{bmatrix} . \begin{bmatrix} \Pi_d \\ \Pi_{d-1} \\ \vdots \\ \Pi_1 \end{bmatrix} = \begin{bmatrix} S_d \\ S_{d+1} \\ \vdots \\ S_{2d} \end{bmatrix}. \tag{1.15}$$

The only thing that remains to do is to determine the degree of the minimal polynomial. At first sight, one may try iteratively all possible degrees (starting from a constant polynomial of degree 0) and match each polynomial produced with the sequence in order to see whether it is an annihilator or not. If the sequence is truly infinite, this might never stop.

Otherwise, if the sequence is finite, one notices, when considering the system, that the maximum degree of the minimal polynomial is half the number of elements in the sequence. This algorithm implies that one should solve successively several linear systems. In practice, it is possible to take advantage of the symmetric structure of the system in order to solve it on-the-fly, while adding the elements of the sequence progressively. This gives the following Berlekamp–Massey algorithm that has a complexity bound of only $O(d^2)$.

The main idea of this algorithm is to explicitly compute the coefficients of the polynomial. Thus, the update of Π is performed in two steps. The trick of the test $2L > k$ is to enable one to perform each of these two steps alternately. Let us explain the algorithm by looking at the three first terms of the sequence. The degree of the minimal polynomial increases by one (at most) every time one adds two elements of the sequence. The δ are called discrepancies.

The first discrepancy is the first term of the sequence, $\delta_0 = S_0$ and $\Pi(X)$, becomes $1 - S_0X$. Discrepancies correspond to the values taken by the polynomial in the sequel of the sequence. If the discrepancy is null, then the polynomial one considers is an annihilator of an additional part of the sequence. Hence, the second discrepancy is $1 - S_0X$ applied to the sequence S_0, S_1, namely $\delta_1 = S_1 - S_0^2$. Therefore, the update of Π is $\Pi - \frac{\delta_1}{\delta_0}X\psi = (1 - S_0X) - \frac{S_1 - S_0^2}{S_0}X$, namely $\Pi = 1 - \frac{S_1}{S_0}X$, which is an annihilator of the sequence S_0, S_1. Then, the third discrepancy is equal to $\delta_2 = S_2 - \frac{S_1^2}{S_0}$ and the two polynomials Π and ψ are, respectively, equal to $1 - \frac{S_1}{S_0}X - \frac{\delta_2}{S_0}X^2$ and $1 - \frac{S_1}{S_0}X$. Hence, Π annihilates S_0, S_1, S_2 and ψ annihilates S_0, S_1.

Algorithm 1.14 Berlekamp–Massey Algorithm

Input $S_0, \ldots, S_n$ a sequence of elements in a field K.
Output The minimal polynomial of the sequence.

1: $b \leftarrow 1; e \leftarrow 1; L \leftarrow 0; \Pi \leftarrow 1 \in \mathbb{F}[X]; \psi \leftarrow 1 \in \mathbb{F}[X]$
2: **For** k from 0 to n **do**
3: $\quad \delta \leftarrow S_k + \sum_{i=1}^{L} \Pi_i S_{k-i}$
4: $\quad$ **If** $(\delta = 0)$ **then**
5: $\qquad e \leftarrow e + 1$
6: $\quad$ **elsif** $2L > k$ **then**
7: $\qquad \Pi \leftarrow \Pi - \frac{\delta}{b} X^e \psi$
8: $\qquad e \leftarrow e + 1$
9: $\quad$ **else**
10: $\qquad temp \leftarrow \Pi - \frac{\delta}{b} X^e \psi$
11: $\qquad \psi \leftarrow \Pi$
12: $\qquad \Pi \leftarrow temp$
13: $\qquad L \leftarrow k + 1 - L; b \leftarrow \delta; e \leftarrow 1$
14: $\quad$ **End If**
15: $\quad$ **If** $e >$ *EarlyTermination* **then**
16: $\qquad$ Stop the algorithm
17: $\quad$ **End If**
18: **End For**
19: **return** $\Pi(X)$

Hence, as multiplication by X of these annihilator polynomials comes to shifting by one position their application in the initial sequence, one obtains $\delta_3 = \Pi[S_1, S_2, S_3]$ and $\delta_2 = \psi[S_1, S_2] = (X\psi)[S_1, S_2, S_3]$. Then, it is possible to combine these two polynomials in order to also annihilate the sequence S_1, S_2, S_3, by $\Pi - \frac{\delta_3}{\delta_2} X\psi$, which is exactly what the algorithm does. In the sequel, if all next discrepancies are null, this means that the polynomial we have obtained is an annihilator of the sequel of the sequence. One can still continue until one is sure of having the minimal polynomial of the $n + 1$ terms of the sequence or one can stop the algorithm earlier without checking the last discrepancies (using the control variable *EarlyTermination*).

As for complexity, the loop ends after $2d +$ *EarlyTermination* and at most $n + 1$ iterations. In each iteration, computing the discrepancy operations and updating the polynomial require both $2\frac{k}{2}$ operations, for an overall number of operation of $(2d +$ *EarlyTermination*$)(2d +$ *EarlyTermination* $+ 1)$.

It is even possible to use a fast algorithm to reduce this complexity, at least asymptotically. The idea is to see the sequence as a polynomial too. Then, the minimal polynomial of the sequence is such that the product of $\Pi(X)$ and $(S_0 + S_1X + S_2X^2 + \ldots)$ has only a finite number of nonzero terms, the terms of degree at most d. It is possible to rewrite this statement in the following way:

$$\Pi(X) \cdot (S_0 + S_1X + \ldots + S_{2n-1}X^{2n-1}) - Q(X) \cdot X^{2n} = R(X). \tag{1.16}$$

Hence, one notices that computing Π, Q, and R can be performed by the Euclidean algorithm interrupted in the middle of the computation, as the degree of R is lower than n. Thus, the complexity bound of the computation is the same as for Euclidean algorithm, namely $O(dlog^2(d))$. However, in practice, the Berlekamp–Massey algorithm remains more efficient for values of d up to dozens of thousands.

1.4.3.3 The Birthday Paradox The Birthday Paradox is a probability result, and it is called a paradox because it seems to go against the first intuition one could have. It is used in several attack methods in cryptanalysis. It also shows that one should distrust intuitions when talking about probabilities.

There are 365 days in a year and still, in a group of more than 23 people, there is more than one chance in two of having at least two of them with the same birth date !

Indeed, let us take a population of k people. Knowing that the number of days in a year is n, the number of combinations of k distinct birth dates is $A_n^k = \frac{n!}{(n-k)!}$. Therefore, the probability of having all people with distinct birth dates is $\frac{A_n^k}{n^k}$. Thus, the probability of having at least two people with the same birthday is

$$1 - \frac{A_n^k}{n^k}.$$

Hence, when considering 365 days, this probability is around one chance in 10 in a group of 9 people, more than one chance in 2 in a group of 23 people, and 99.4% in a group of 60 people. More generally, one has the following theorem:

Theorem 18 *In a set of* $\lceil 1.18\sqrt{n} \rceil$ *elements chosen randomly among n possibilities, the probability of collision is higher than 50%.*

Proof. We have seen that the number of collisions, in a space of size $n = 2^m$ with k draws, is $1 - \frac{A_n^k}{n^k}$. One has to give an estimation of this probability: $1 - \frac{A_n^k}{n^k} = 1 - (1 - \frac{1}{n})(1 - \frac{2}{n}) \ldots (1 - \frac{k-1}{n})$. Yet, $1 - x < e^{-x}$, for x positive, thus

$$1 - \frac{A_n^k}{n^k} > 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{n}} = 1 - e^{-\frac{k(k-1)}{2n}}.$$

Then, for this probability to be greater than α, it is sufficient to have $k(k-1) = -2n\ln(1-\alpha)$, namely, as k is positive, $k = \frac{1}{2} + \sqrt{n}\sqrt{\frac{1}{4n} - 2\ln(1-\alpha)}$. Hence, for $\alpha = 0.5$, $k \approx 1.18\sqrt{n}$ (again one has, for $n = 365$, $k \approx 22.5$). □

This kind of collision probability – that one's intuition would tend to weaken – enables one to build attacks against systems for which intuition would give a limited chance of success.

1.4.3.4 Yuval's Attack on Hash Functions Resistance to collision of hash functions can be measured: one has to determine the probability of finding collisions, which is close to the probability of collision in birth dates (birth dates play the role of fingerprints for individuals).

That is why Yuval's attack on hash functions is also called a birthday attack. It is a question of transmitting some corrupted message $\widetilde{M}$ instead of a legitimate message M, in such way that the corruption is unnoticeable for a hash function h. Then, one looks for M' and $\widetilde{M}'$, such that $h(M') = h(\widetilde{M}')$. After that, one can, for example, fraudulently change M into $\widetilde{M}$, or send M and pretend to have sent $\widetilde{M}$, which is precisely what h should prevent!

Algorithm 1.15 Yuval's birthday attack

Input $h : \{0, 1\}^* \longrightarrow \{0, 1\}^m$ a hash function.
M legitimate, $\widetilde{M}$ fraudulent.

Output $M' \approx M$ (*i.e.* M' close to M) and $\widetilde{M}' \approx \widetilde{M}$ such that $h(M') = h(\widetilde{M}')$.

1: Generate $t = 2^{\frac{m}{2}} = \sqrt{2^m}$ slight modifications of M, denoted by M'.
2: For all t, compute $h(M')$.
3: Generate several $\widetilde{M}'$, slight modifications of $\widetilde{M}$, until finding a collision, with some M' (*i.e.* $h(\widetilde{M}') = h(M')$).

As a consequence of Theorem 18, the expected number of draws of $\widetilde{M}'$ in Yuval's attack is $O(t) = O(\sqrt{2^m})$.

If one uses Yuval's attack to send M' and then to repudiate it later arguing that $\widetilde{M'}$ was actually sent, one has more than one chance in two of succeeding in $\sqrt{2^m}$ attempts. This shows that brute force can be efficient if a hash function is not collision resistant.

But is this attack really feasible? A simple calculation is enough to be convinced: for a numerical fingerprint on 128 bits, one should perform around $O(2^{64})$ attempts, which is feasible on general public machines of today: a computer running at 3 GHz performs $3 * 10^9 * 24 * 3600 \approx 2^{48}$ operations per day, thus it would take a little more than two months on the 1000 PCs of a company to find a collision.

But if one uses hash functions on 160 bits, the cost is multiplied by a factor $2^{16} = 65536$, which is unreachable so far.

1.4.3.5 Factoring Composite Numbers It is quite easy to distinguish a composite number from a prime number. But knowing the numbers composing it seems to be a much more difficult problem. It is the factorization problem. Although it can be formulated in a quite simple way, so far there does not exist an efficient solution to it (for instance, the famous Sieve of Eratosthenes is useless for numbers of more than *10* digits).

The difficulty of this problem and the efficiency of attack methods are very important, as a lot of one-way functions rely on the difficulty of factorization or on the

difficulty of equivalent problems, such as the discrete logarithm problem. Thus, looking for good factorization algorithms is almost a cryptanalysis method.

Many different algorithms do exist. The goal of this section is not to enumerate them all but rather to give an idea of the most efficient ones for numbers of different sizes.

Pollard's Rho algorithm (Numbers of few digits). The first class of target numbers is composed of "everyday composite numbers," namely numbers of less than *20* digits. Pollard's algorithm is very efficient for such numbers.

The algorithm only requires a few lines of code (around forty) and is very easy to implement. Let m be the composite number one wishes to factorize. First of all, one has to compute a sequence of the form $u_{k+1} = f(u_k) \mod m$ of large period (the longer the u_k are distinct the better).

Then, the idea is to notice that, if p is a factor of m, the distinct u_k modulo m are less often distinct modulo p (Table 1.8). In this case, if $u_i = u_j \mod p$ then the GCD of m and $u_i - u_j$ is equal to kp and it is a nontrivial factor of m. If the u_i are actually pairwise distinct, the computation ends in at most p steps. A first version of the algorithm consists in producing some u_i and, when adding a new element, computing the GCD with all previous u_k's. This version has two major drawbacks: first of all, one has to store around p elements; also, it takes j^2 GCD computations if i and $j > i$ are the smallest indexes such that $u_i = u_j \mod p$. The second trick of Pollard is to use Floyd's cycle detection. It consists in storing only the u_k such that k is a power of 2. Indeed, as the u_k are generated by a function, if $u_i = u_j$, then for all $h \geq 0, u_{i+h} = u_{j+h}$ and a cycle is created modulo p, even if it is not directly noticeable.

When only storing powers of 2, the cycle will only be detected for $u_{2^a} = u_{2^a+j-i}$ with $2^{a-1} < i \leq 2^a$, as illustrated in Figure 1.21.

Yet, $2^a + j - i < 2i + j - i = i + j < 2j$. Hence, one performs at most twice the number of requested operations. One single GCD is computed at each step and one single additional element is stored throughout the algorithm. This gives the very fast algorithm presented in Algorithm 1.16.

If one takes $f(u) = u^2 + 1$, so that the u_i is actually pairwise distinct modulo m, for example, it will take at worst $2p$ iterations if p is the smallest factor of m, and even less in general:

Theorem 19 *Pollard's Rho algorithm has more than one chance in two of succeeding in $O(\sqrt{p})$ steps.*

Proof. Once again, the proof is similar to that of the birthday paradox. If k distinct values u_i are drawn randomly, then there are A_p^k combinations without collisions between

TABLE 1.8 Distribution of the Multiples of *p* Modulo *m*

0	1	2	…	p	p+1	p+2	…	kp	kp+1	kp+2	…	m-1
		u_i	u_l		u_k			u_h		u_j		

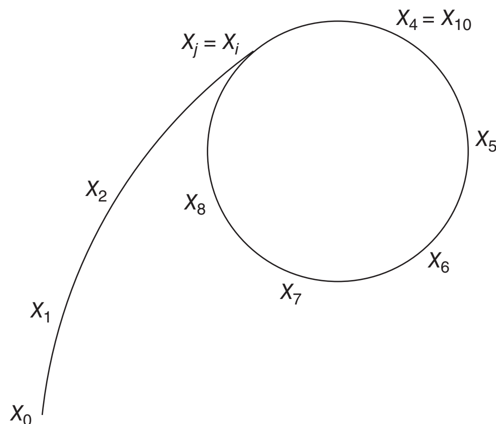

Figure 1.21 Floyd's cycle detection, in the form of a rho

Algorithm 1.16 Pollard's Factoring

Input An integer m, composite.
Output p, non trivial factor of m.

1: $p \leftarrow 1$;
2: Generate y randomly;
3: $k \leftarrow 0$;
4: **While** $(p = 1)$ **do**
5:   **If** k is a power of 2 **then**
6:     $x \leftarrow y$;
7:   **End If**
8:   $y \leftarrow f(y) \mod m$;
9:   $p \leftarrow \gcd(y - x, m)$;
10:   Increase k;
11: **End While**
12: **return** p.

---

the u_i for an overall p^k. For the probability of finding a nontrivial factor to be greater than $1/2$, one must have – according to Theorem 18 - $k > 0.5 + 1.18\sqrt{p}$. □

In practice, this algorithm factorizes numbers of 1 up to around 25 digits within seconds (with factors of 12 or 13 digits, this gives approximately 10 million operations!) but very quickly, it becomes useless when considering factors of more than 15 digits.

Elliptic curves (Numbers with a few dozen of digits). To go further, one can use a method based on elliptic curves, designed by Pollard and Lenstra.

This method uses elliptic curves of the form $y^2 = x^3 + ax + b$, defined in Section 1.3.6. The idea is to consider the set of solutions of the latter equation modulo the number m to be factorized and to try to add them as if this set was a group. As the addition of points (see Theorem 11) requires to invert coordinates modulo m, if the inversion of say y_1 fails then it means that y_1 is not invertible. In other words y_1 contains a proper factor of m, which can be revealed by computing $gcd(y_1, m)$. To simplify, suppose $m = pq$ is the product of only two distinct primes. Then, the curve equation defines two proper elliptic curves, one modulo p and one modulo q. In consequence for any point P, Lagrange's theorem (Theorem 1, page 31), ensures that $[k]P = \mathcal{O}$, say modulo q, only if k divides N_q, the number of points of the curve modulo q. If the curve is chosen randomly, then N_q and N_p are close to p and q, and not necessarily primes. Hence, some of their prime factors will differ with high probability. Therefore, if we compute $[e]P$, for a small e, it is likely that e will divide one of N_p or N_q, but less likely that it divides both numbers at the same time. When this is the case, it means that $[e]P$ is *not* on the curve modulo m and therefore that its computation will crash. The overall procedure is thus to try many small prime factors e. A way to perform this efficiently is to compute $[B!]P$ with a not too large B. The algorithm is detailed as Algorithm 1.17.

Algorithm 1.17 Lenstra's elliptic curve factoring

Input An integer m, composite.
Input A small bound B on the prime factors to be used.
Output p, non trivial factor of m, or Failure.

1: Pick a random point $P = (x, y)$ with random non-zero coordinates;
2: Pick a random non zero $a \mod m$ and compute $b \equiv y^2 - x^3 - ax \mod m$;
3: **For** i=2 **to** B **do**
4: $P \leftarrow [i]P$; {Via recursive double-and-add in the pseudo-group of the $\mathbb{E}(m; a, b)$ curve}
5: **If** Inversion of P_y fails **then**
6: **return** $gcd(P_y, m)$;
7: **End If**
8: **If** $P = \mathcal{O}$ **then**
9: **return** Failure;
10: **End If**
11: **End For**
12: **return** Failure;

The computational procedure remains simple (around 150 lines of code) and we can give a few ideas concerning the properties of this algorithm: it is conjectured that, in order to factorize an integer m of smallest factor p, this algorithm requires an average number of operations of the order of

$$O\left((\ln m)^2 e^{\sqrt{2 \ln p \ln(\ln p)}}\right).$$

In practice, this algorithm factorizes numbers of 25 to 40 digits (with two factors of similar sizes) within seconds. Besides, if one is lucky and chooses some particular elliptic curve, the latter might enable one to factorize very quickly: the project ECMNET (Elliptic Curve Method on the Net) provides an implementation of this algorithm, available on the Internet. This project has allowed the factorization of numbers with factors up to 73 digits.

The main issue is that good elliptic curves vary for each number one wishes to factorize and so far there does not exist a method of finding the appropriate curve for a given number. Notwithstanding, the rapidity of computation when one has found a "good" elliptic curve is at the origin of the factorization program on the Internet: indeed, many Internet users can retrieve the ECM program and launch it in order to try several elliptic curves. Hence, numerous elliptic curves can be studied at the same time and possibly speed up the search of prime factors.

Number Field Sieve (World champion). Finally, the current champion of RSA key factorization (product of two large prime numbers, see Section 3.4.2) is the *Number Field Sieve* algorithm, which seems to require – in order to factorize some number m, product of two factors of similar sizes – an average number of operations of the order of

$$O\left(e^{\sqrt[3]{(7.11112)\ln(m)\ln(\ln(m))^2}}\right).$$

A number field is an extension of the field of rational numbers (one considers the infinite field $\mathbb{Q}[X]/P$).

The number field sieve is a generalization of the quadratic sieve when considering the field of all integers modulo m. For the sake of simplicity, we only present the main idea of the latter.

The aim is to find couples of numbers whose squares are congruent modulo m : $x^2 = y^2 \mod m$. Then $x^2 - y^2 = (x-y)(x+y)$ is a multiple of m. Now, let $d = gcd(x-y, m)$, if one is lucky, $x \neq \pm y \mod m$ so that $1 < d < m$ and thus d is a nontrivial factor of m.

Example 1.13 Let us try to factorize 7429. We compute randomly some squares: $87^2 = 7429 + 140$ and $88^2 = 7429 + 315$. Yet, 140 and 315 are small with respect to 7429 and thus easier to factorize, for example using Pollard's method. One obtains $140 = 2^2 \cdot 5 \cdot 7$ and $315 = 3^2 \cdot 5 \cdot 7$. Therefore, one can write $(227)^2 = (87 \cdot 88)^2 = (2 \cdot 3 \cdot 5 \cdot 7)^2 = (210)^2 \mod 7429$. We have found a relation of the form $x^2 = y^2 \mod 7429$, which gives us a factor of 7429: $(227 - 210) = 17$, and $7429 = 17 \cdot 437$.

The whole difficulty of the algorithm is to find such integers x and y. For this, one has to compute several squares and store those whose remainders modulo m are small enough to be factorized using another method. Then, one has to find a linear combination of these squares that would give another square: in a matrix, let us store the exponents in the columns and the squares in the lines (Table 1.9).

TABLE 1.9 Quadratic Sieve for 7429

	Exponent of 2	Exponent of 3	of 5	of 7	of …
83^2	2	3	1	0	…
87^2	2	0	1	1	…
88^2	0	2	1	1	…
⋮					

According to this table, $(87 * 88)^2$ is a square if and only if the line of 87^2 added to the line of 88^2 only gives even exponents. In other words, if M is the matrix of exponents (rows and columns of Table 1.9), one has to find some binary vector x such that xM is even or to find a solution modulo 2 to the associated linear system: x such that $xM = 0 \mod 2$.

Although the basic idea is quite simple, the calculation is a little more delicate than for the previous algorithms. Still, this algorithm holds the current records. In particular, it enabled one to factorize a 200 digit (665 bits) RSA key in 2005. The computing time for this last factorization was gigantic: more than a year and a half of computation on more than 80 machines !

1.4.3.6 Strong Prime Numbers RSA cryptography (see Chapter 3) is based on the use of two prime numbers p and q and on the difficulty of factorizing their product m. In order to resist various factorization methods (some of them are presented in the form of exercises in Chapter 3, on page 173), the prime numbers one uses must satisfy several properties:

- In order to resist factorization based on elliptic curves, p and q must have similar sizes and must be large enough. For instance, in order to work with numbers on 1024 bits, they should both have a size of around 512 bits.
- $p - q$ must be large enough: otherwise, it is sufficient to try as a value for p or q all integers close to $\sqrt{m}$ (Fermat's square root attack).
- In order to resist Pollard's algorithms $p - 1$ and $p + 1$ (which take advantage of the factorization of $p - 1$ and $p + 1$ when possible), p and q have to be *strong* prime numbers, that is, each of them must satisfy the conditions:
  - $p - 1$ has a large factor, denoted by r.
  - $p + 1$ has a large factor.
  - $r - 1$ has a large factor.

Gordon's Algorithm 1.18 enables one to generate strong prime numbers:

Exercise 1.43 *Prove that the output of Gordon's algorithm is a strong prime number.* *Solution on page 294.*

Algorithm 1.18 Gordon's algorithm

Input A number of bits b.
Output A strong prime number on at least $2b + 1$ bits.
1: Generate two prime numbers s and t on b bits.
2: Look for r prime in the form $2kt + 1$.
3: Compute $l \leftarrow 2(s^{r-2} \mod r)s - 1$.
4: **return** p, the smallest prime number of the form $l + 2hrs$.

1.4.3.7 Solving the Discrete Logarithm Problem Alongside modular exponentiation, the other major class of one-way functions relies on the discrete logarithm problem.

Let G be a group of size n admitting a generator (i.e., G is cyclic). For instance, one could consider the group of invertible elements modulo some prime number p, $\mathbb{Z}/p\mathbb{Z}^*$.

Given a primitive root g and an element b in G, the problem is to find the discrete logarithm of b in base g, namely to find x such that $b = g^x$.

The naive algorithm for solving this problem is to try all possible x until one finds the right one. The worst and the average complexity bounds are $O(n)$, thus an exponential complexity with respect to $\log(n)$, the size of n.

So far, the best known algorithms for solving this problem have a complexity bound $O(\sqrt{n})$ in the general case. Most of the time, these algorithms are based on factorization algorithms: we will see that variants of Pollard's Rho – $O(\sqrt{n})$ – and index calculus – $O(n^{1/3})$ – algorithms can be applied to groups. However, complexities are raised to the square with respect to factorization. Indeed, if one considers numbers modulo some composite number – a product of two prime numbers – $n = pq$, factorization algorithms have a complexity bound that depends on the smallest prime factor, roughly: $O(p^{1/3}) = O(n^{1/6})$. That is why the discrete logarithm method enables one to consider numbers half the size as those used for factorization-based methods with the same level of security. These sizes are even further reduced if one considers more generic groups, such as the group of points of an elliptic curve, for which the best discrete logarithm methods do not apply.

Baby step, Giant step. This method was developed by Shanks, and it is divided into two phases: the baby step, tests from g^x to g^{x+1} for all x in some interval, and the giant step, jumps from $g^{x\lfloor\sqrt{n}\rfloor}$ to $g^{(x+1)\lfloor\sqrt{n}\rfloor}$.

The idea is to decompose x into two pieces $x = i\lfloor\sqrt{n}\rfloor + j$, with i and j between 1 and $\lceil\sqrt{n}\rceil$. Hence, one can write $b = g^x = (g^{\lfloor\sqrt{n}\rfloor})^i g^j$, or $b(g^{-\lfloor\sqrt{n}\rfloor})^i = g^j$. Thus, one has to compute all possible g^j (baby step), and all possible $b(g^{-\lfloor\sqrt{n}\rfloor})^i$ in order to checks if one of these values has been computed in the baby step (giant step).

Although computing all these values only takes $2\sqrt{n}$ multiplications, looking for the correspondences with a naive method implies that one has to try $\sqrt{n}\sqrt{n} = n$ possibilities in the worst case.

The trick that decreases this complexity is to sort the g^j in increasing order (complexity $O(\sqrt{n}\log(\sqrt{n}))$) in order to be able to perform comparisons with a dichotomous research with only $\sqrt{n}\log_2(\sqrt{n})$ tests!

Therefore, the time complexity is improved; unfortunately, the space complexity is such that this algorithm is not practical: one has to store all $\sqrt{n}$ integers. Even for a reasonable number of operations (around $n = 2^{128}$ nowadays), the required memory space is then of the order of several billion gigabytes.

Pollard's Rho returns. Pollard's Rho algorithm enables one to modify the baby step, giant step method and to introduce Floyd's cycle detection. Hence, one can preserve the time complexity bound $O(\sqrt{n}\log(n))$, while reducing significantly the memory complexity bound down to only $O(\log(n))$ bytes. In comparison with the factorization algorithm, one has to modify the generator function of the sequence in the following way:

Build three subsets S_1, S_2, and S_3 of G of similar sizes forming a partition of G (for example, in $\mathbb{F}_p^*$, with $p \geq 3$ one can always take $S_1 = \{u = 1 \mod 3\}$, $S_2 = \{u = 2 \mod 3\}$, and $S_3 = \{u = 0 \mod 3\}$).

Then, one defines the generator function f such that

$$u_{k+1} = f(u_k) = \begin{cases} bu_k & \text{if } u_k \in S_1 \\ u_k^2 & \text{if } u_k \in S_2 \\ gu_k & \text{if } u_k \in S_3 \end{cases}.$$

Hence, each element of the sequence can be written in the form $u_k = g^{i_k}b^{j_k}$ for some i_k and some j_k. Yet, in the same way as the Rho algorithm for factorization, the u_k are more or less equally spread modulo p. Therefore, a collision $u_k = u_l$ occurs on the average after $\sqrt{p}$ draws, even if $j_k \neq j_l$ and $i_k \neq i_l$. The function f insures, as for factorization, that this collision will be constantly reproduced after k steps; thus, it is possible to find y such that $u_y = u_{2y}$ thanks to Floyd's algorithm with only a memory complexity bound of several integers of size $O(\log(n))$.

Then, one has $u_k = g^{i_k}b^{j_k} = g^{i_l}b^{j_l} = u_l$ and – at the same time – $j_k \neq j_l$. In this case, one obtains $b^{j_k-j_l} = g^{i_l-i_k}$, which means that, in the space of indexes, one directly finds x (one recalls that $b = g^x$):

$$x = (i_l - i_k) \cdot (j_k - j_l)^{-1} \mod n.$$

Be careful: we are solving logarithms in $\mathbb{Z}/p\mathbb{Z}^*$ but the latter equation lies in $\mathbb{Z}/n\mathbb{Z}$ where $n = p - 1$; thus although $j_k - j_l$ is nonzero, it is not necessarily invertible modulo n. If this is not the case, then restart the algorithm.

Coppersmith's Index Calculus. The same way as we have just modified Pollard's algorithm and adapted it to the computation of discrete logarithms, it is possible to modify sieve algorithms. Then, the obtained method works for the discrete logarithm over the group of invertible of a finite field, but not for any group. In particular,

cryptology over curves are still resisting this kind of attack. Let us show anyway how this attack works over finite fields with an example.

Example 1.14 How to find x, such that $17 = 11^x \mod 1009$? One recalls that 1009 is prime and that 11 is a primitive root of $\mathbb{Z}/1009\mathbb{Z}$: $Z/1009\mathbb{Z}^* = \{11^i$, for $i = 1 \dots \varphi(1009) = 1008\}$. The idea is to draw randomly some values of i such that $v_i = 11^i \mod 1009$ can be easily factorized (i.e., it only has small prime factors, or *smooth*). In practice, one is given a basis of prime factors, for example $B = \{2, 3, 5, 7, 11\}$.

Then, for each prime number $p_j \in B$, one divides v_i with the greatest possible power of p_j. At the end of the process, if one obtains the value 1, then v_i can be factorized in the basis B.

After several random draws, we keep the values 104, 308, 553, and 708:

$$\begin{aligned} 11^{104} &= 363 = 3 \cdot 11^2 && \mod 1009 \\ 11^{308} &= 240 = 2^4 \cdot 3 \cdot 5 && \mod 1009 \\ 11^{553} &= 660 = 2^2 \cdot 3 \cdot 5 \cdot 11 && \mod 1009 \\ 11^{708} &= 1000 = 2^3 \cdot 5^3 && \mod 1009 \end{aligned}$$

When considering the logarithms, Theorem 7 guaranties that one obtains a linear system in the space of exponents whose unknown values are the discrete logarithms of 2, 3, and 5, now modulo 1008:

$$\begin{aligned} 104 &= \log_{11}(3) + 2 && \mod 1008 \\ 308 &= 4\log_{11}(2) + \log_{11}(3) + \log_{11}(5) && \mod 1008 \\ 553 &= 2\log_{11}(2) + \log_{11}(3) + \log_{11}(5) + 1 && \mod 1008 \\ 708 &= 3\log_{11}(2) + 3\log_{11}(5) && \mod 1008. \end{aligned}$$

This gives $\log_{11}(3) = 102 \mod 1008$ (or in other words $11^{102} = 3 \mod 1008$), then $\log_{11}(2) = (308 - 553 + 1)/2 = 886 \mod 1008$ and

$$\log_{11}(5) = 308 - 4 \cdot 886 - 102 = 694 \mod 1008.$$

One has to find a number in the form $17 \cdot 11^y$ whose remainder can be factorized in the basis $B = \{2, 3, 5, 7, 11\}$. Still with the same method, one finds – for example – after several random attempts: $17 \cdot 11^{218} = 2 \cdot 3 \cdot 5 \cdot 11 \mod 1009$. Thus, $x = 886 + 102 + 694 + 1 - 218 = 457 \mod 1008$ satisfies $17 = 11^{457} \mod 1009$.

Such index calculus algorithms are more efficient in some particular fields. The record is held by a French team which managed – in November 2005 – to compute a discrete logarithm in the field $\mathbb{F}_{2^{613}}$ in only 17 days on the 64 processors of the Bull supercalculator Teranova.

Now we have at our disposal enough tools to start specializing. We are able to build the objects we will use throughout this book, and we have described the common algorithms that are the foundations of coding. Each of the following chapters will refer to the work in this chapter, depending on needs of the specific objectives: compression, encryption, or correction.