Chapter 2
Basic Concepts from Probability, Complexity, Algebra and Number Theory

In this chapter we study the basic notions from mathematics and computer science that we will be using throughout the book. Its purpose is not to give a comprehensive account of these concepts but, rather, to study them from the point of view of their applications to cryptography, emphasizing the algorithmic aspects. For this task we will be using Maple as a tool to implement the relevant algorithms and to experiment with them. Bearing in mind these premises, we include here the required concepts and results from probability, complexity theory and modular arithmetic, including quadratic residues and modular square roots, and we also give a brief introduction to the relevant aspects of group theory and finite fields. Thus, for readers already familiar with these subjects, this chapter will hopefully serve as a refresher of the topics mentioned in the Preface (with the exception of linear algebra which is not included here) and, at the same time, as an introduction to the use of Maple to explore the algorithms discussed. Even for readers not familiar with these subjects we hope that sufficient information is included here to allow them to profitably read the rest of the book. We include proofs of most of the algorithmic and mathematical results mentioned, with pointers to the literature for the reader who wishes to go deeper on these topics and, at the same time, we will be introducing here a great deal of the notation used throughout the book.

2.1 Basic Probability Theory

Randomness plays a crucial role in cryptology and in order to deal with this concept we need some basic notions from probability theory. We assume the reader already knows the elements of this theory but, in order to fix the notation, we recall here some of the most important probability-related concepts that we will use. We refer to any introductory text on (discrete) probability such as, for example, [94, 167] for more detailed information on these topics.

J. L. Gómez Pardo, *Introduction to Cryptography with Maple*,
DOI: 10.1007/978-3-642-32166-5_2, © Springer-Verlag Berlin Heidelberg 2013

Definition 2.1 A finite *probability space* is a pair $(\Omega, \Pr)$, where Ω is a finite set (called the *sample space* and whose elements are called *elementary events*) and

$$\Pr : \Omega \to \mathbb{R}$$

is a *probability distribution*, i.e., a function that satisfies the following conditions:

1. $0 \leq \Pr(\omega) \leq 1$ for every $\omega \in \Omega$,
2. $\sum_{\omega \in \Omega} \Pr(\omega) = 1$.

If we run a random experiment, then every elementary event of the sample space represents a possible outcome. The most important probability distribution used in cryptography is the *uniform distribution* where all the elementary events occur with the same probability, i.e., $\Pr(\omega) = \frac{1}{|\Omega|}$ for every $\omega \in \Omega$.

For example, rolling a die can be interpreted as an experiment on the sample space $\Omega = \{1, 2, \ldots, 6\}$ and, assuming that the die is fair, each possible outcome is equally likely and hence the probability of each outcome is $\Pr(n) = \frac{1}{6}$ for each $n \in \Omega$ (in other words, the distribution is uniform).

The assignment of probabilities can be extended to subsets $E \subseteq \Omega$, which are called *events*, by setting $\Pr(E) = \sum_{\omega \in E} \Pr(\omega)$. For the uniform distribution, $\Pr(E) = \frac{|E|}{|\Omega|}$ is just the ratio of the number of "favorable cases" to the number of all "possible cases" as in Laplace's original definition of probability.

The complement of the event $E \subseteq \Omega$ is the set $\bar{E} = \Omega - E$ and it is clear that $\Pr(\bar{E}) = 1 - \Pr(E)$. From Definition 2.1 have that $\Pr(\Omega) = 1$ and then $\Pr(\emptyset) = 0$. More generally, if we have two events $E, F \in \Omega$, then the probability of the *union event* $E \cup F$ is given by

$$\Pr(E \cup F) = \Pr(E) + \Pr(F) - \Pr(E \cap F).$$

For example, if Ω consists of the possible outcomes of rolling a die, then the event E described by "an even number is rolled" has probability $\frac{1}{2}$ and the event F given by "a multiple of 3 is rolled" has probability $\frac{1}{3}$. The probability of $E \cap F$ is just the probability of rolling a 6 and hence equal to $\frac{1}{6}$, so that the probability of the event $E \cup F$ is $\frac{1}{2} + \frac{1}{3} - \frac{1}{6} = \frac{2}{3}$.

The next definition captures the idea that two events may be such that the probability of one of them does not influence the probability of the other:

Definition 2.2 Two events $E, F \in \Omega$ are said to be *independent* if

$$\Pr(E \cap F) = \Pr(E) \cdot \Pr(F).$$

In other words, E and F are independent when the probability of them both occurring is the product of their individual probabilities of occurring.

Exercise 2.1 Suppose that two fair dice are rolled in succession and let E be the event that the first die is a 3 and F the event that the sum of the dice is 5. Determine whether these events are independent.

It is often necessary to compute the probability of an event F occurring when it is known that another event E has occurred. This is called the *conditional probability of F on E*:

Definition 2.3 If E, F are events such that $\Pr(E) > 0$ then the *conditional probability of F on E*, $\Pr(F|E)$, is defined by:

$$\Pr(F|E) = \frac{\Pr(F \cap E)}{\Pr(E)}.$$

The intuitive idea behind this definition is that, if we assume that E occurs, our sample space is now E instead of Ω and in this smaller space we compute the ratio between the probability of the part of F included in E and the probability of E itself. Observe that, in particular, the independence of E and F is equivalent to $\Pr(F|E) = \Pr(F)$ and also to $\Pr(E|F) = \Pr(E)$.

In our earlier example where Ω consists of the possible outcomes of rolling a die, E is the event "an even number is rolled" and F the event "a multiple of 3 is rolled" we have that $\Pr(F|E) = \frac{1}{3}$, while $\Pr(E|F) = \frac{1}{2}$.

As an immediate consequence of the previous definition we have:

Lemma 2.1 (Bayes' formula) *Let $E, F \in \Omega$ be events such that $\Pr(E) > 0$, $\Pr(F) > 0$. Then*

$$\Pr(E|F) = \frac{\Pr(F|E)\Pr(E)}{\Pr(F)}.$$

Proof From the definition of conditional probability we have that $\Pr(F|E)\Pr(E) = \Pr(F \cap E) = \Pr(E|F)\Pr(F)$. □

The following proposition gives another version of Bayes' formula which exploits the fact that an event is the union of the two disjoint events given by its intersection with another event and its complement:

Proposition 2.1 *Let $E, F \subseteq \Omega$ be two events such that $\Pr(E) > 0$, $\Pr(F) > 0$. Then*

(i) $\Pr(F) = \Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))$.

(ii) $\Pr(E|F) = \dfrac{\Pr(F|E)\Pr(E)}{\Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))}$.

Proof For (i), we have that $F = (F \cap E) \cup (F \cap \bar{E})$, where the union is disjoint. Thus we see that $\Pr(F) = \Pr(F \cap E) + \Pr(F \cap \bar{E}) = \Pr(F|E)\Pr(E) + \Pr(F|\bar{E})(1 - \Pr(E))$, where the last equality follows from the definition of conditional probability and from the fact that $\Pr(\bar{E}) = 1 - \Pr(E)$. (ii) follows from (i) using Bayes' formula. □

When running a random experiment on a finite sample space we might be more interested in some values that depend on the outcome of the experiment (for example, in the net gain in a game of chance) than in the outcome itself. This leads to the

definition of a real-valued *random variable* as a function defined on the sample space:

Definition 2.4 Let $(\Omega, \Pr)$ be a finite probability space. A *random variable* is a function $X : \Omega \to \mathbb{R}$.

Since the sample spaces we are using are finite, a random variable takes only a finite number of values. Random variables can be used to define events related to a real number $x \in \mathbb{R}$ such as $\{\omega \in \Omega | X(\omega) \leq x\}$, $\{\omega \in \Omega | X(\omega) > x\}$ or $\{\omega \in \Omega | X(\omega) = x\}$. The probabilities of these events are denoted simply by $\Pr(X \leq x)$, $\Pr(X > x)$, and $\Pr(X = x)$, respectively. The expected value (or expectation, or mean) of a random variable is the average of its values weighted by their probability of occurrence:

Definition 2.5 Let X be a random variable taking $x_1, x_2, \ldots, x_n$ as values. The *expected value* or *mean* of X is defined as: $\mathrm{E}(X) = \sum_{i=1}^{n} x_i \cdot \Pr(X = x_i)$.

Exercise 2.2 Let X be the random variable defined by the sum of the numbers appearing on two rolled fair dice. Compute the expected value of X.

Let us see now how the expected value of a random variable may be used to estimate the number of trials in an experiment. For example, suppose that a randomized algorithm (an algorithm that accepts random bits as input, see Sect. 2.3; a typical example is an algorithm that searches for primes among randomly chosen integers) produces some desired result with probability p. A natural question is then: How many times must the algorithm be executed, on average, to obtain this result?

The following proposition shows that the number of times the algorithm must be run is $1/p$:

Proposition 2.2 *Suppose that an event $E \subseteq \Omega$ occurs with probability p. Then the expected value of the number of trials until E occurs is $1/p$.*

Proof Let $E \subseteq \Omega$ be an event and $p = \Pr(E)$. Let X be a random variable that gives the average number of independent trials until E occurs and $\mu = \mathrm{E}(X)$. We see that $\Pr(X = 1) = p$ and $\Pr(X > 1) = 1 - p$ so that, with probability p, E occurs already on the first trial and with probability $1 - p$ one needs the average value of μ trials after the first (unsuccessful) trial, i.e., a total of $1 + \mu$ trials. Therefore, $\mu = 1 \cdot p + (1 + \mu) \cdot (1 - p)$, which implies that $\mu = 1/p$. □

Example 2.1 A *Bernoulli* experiment is a random experiment such that the outcome of a single trial is either 1 (success) with probability p or 0 (failure) with probability $1 - p$. If a sequence of independent Bernoulli trials with success probability p is performed, then we are in the situation described in Proposition 2.2. Although we are not going to explain density and distribution functions here (see, for example, [167] for more details), we mention that the *geometric* distribution estimates the number of failures there will be before the first success is achieved in a sequence of independent Bernoulli trials (more generally, the Pascal distribution estimates the

number of failures before r successes). This and many other probability distributions are implemented in Maple's package `Statistics`. In particular, Maple gives us the result in the above proposition by setting:

```
> with(Statistics):
  X := RandomVariable(Geometric(p));
  ExpectedValue(X);
                                  (-p + 1)/p
```

Thus $(1/p) - 1$ failures are expected before obtaining the first success, i.e., the expected number of trials for the first success is $1/p$.

2.2 Integers and Divisibility

Let $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ be the set of integers. The usual operations of addition and multiplication are defined in $\mathbb{Z}$ and give it the structure of a commutative ring with identity (see the definition in Sect. 2.5 or in [45, 48]). In fact, $(\mathbb{Z}, +, \cdot)$ is the prototypical ring from which this concept arises by abstraction. Actually, $\mathbb{Z}$ is a special type of commutative ring called a (commutative) domain because it has no *zero divisors*, i.e., there are no nonzero elements $a, b \in \mathbb{Z}$ such that $ab = 0$. We assume that the basic properties of addition and multiplication (associativity, commutativity, distributivity, etc.) are well known and start by studying the concept of divisibility.

Definition 2.6 Let $a, b \in \mathbb{Z}$ such that $a \neq 0$. We say that *a divides b* (or, in other words, that *b is a multiple of a*) if there exists an integer q such that $b = aq$.

If a divides b then a is called a *divisor* of b and we write $a|b$. If a is not a divisor of b we write $a \nmid b$. Some of the most basic properties of divisibility are collected in the following proposition whose proof is left as an exercise.

Proposition 2.3

(i) If $a \mid b$ and $b \mid c$ then $a \mid c$ (transitivity).
(ii) If $a \mid b$ and $b \mid a$ then $a = \pm b$.
(iii) If $a \mid b$ and $a \mid c$, then $a \mid (bx + cy)$ for all $x, y \in \mathbb{Z}$.

Next, we recall a couple of important results whose proofs are straightforward and can be found in any introductory number theory textbook (see, for example, [77]). We start with the following fact, which shows that even if $b \nmid a$, we can divide b into a obtaining a quotient and a remainder:

Theorem 2.1 (Division algorithm) *Let $a, b \in \mathbb{Z}$ such that $b > 0$. Then there exist unique integers q and r such that $a = bq + r$ and $0 \leq r < b$.*

The integers q and r are, respectively, the *quotient* and the *remainder* when a is divided by b. The division algorithm is implemented in Maple by means of the functions `irem` and `iquo`. For example

```
> iquo(25,7,'r'), r;
                                    3, 4
```

shows that the quotient of dividing 25 by 7 is 3 and the remainder is 4. The quotient is the output of the function and the remainder is stored in the variable `'r'`. Similarly, if we compute

```
> irem(25,7,'q'), q;
                                    4, 3
```

we obtain as output the remainder and the quotient is stored in the variable `'q'`.

We recall the definition of a prime number, a concept that plays an important role in modern cryptography:

Definition 2.7 An integer p is called a *prime number* if $p > 1$ and the only positive integers dividing p are 1 and p. A *composite number* is an integer greater than 1 that is not prime.

The first primes are $2, 3, 5, 7, 11, 13, \ldots$ and there are arbitrarily large primes because, as Euclid already knew more than 2300 years ago (Elements, Book IX, Proposition 20), the number of primes is infinite [194]. Another important fact that Euclid also knew is:

Theorem 2.2 (The fundamental theorem of arithmetic) *Let $a > 1$ be an integer. Then a can be written as a product, $a = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, with p_i prime and $e_i \geq 1$ for $i = 1, \ldots, k$. Furthermore, this factorization in prime powers is unique if the primes are written in increasing order.*

As we shall see, the problem of finding the prime factorization of a large integer is thought to be computationally hard and this fact turns out to be very useful for cryptography.

2.2.1 Representation of Integers

The usual method that we use to represent integers is by means of their decimal expansion. But computers work naturally with the binary expansion and any integer $b > 1$ can be used as the *base* of a number system. These representations rely on the following theorem in which we use the notation $\lfloor x \rfloor$ to denote the *floor* of a real number x, i.e., the greatest integer less than or equal to x.

Theorem 2.3 (Positional representation of numbers) *Let $b > 1$ be an integer. Then each positive integer n has a unique representation in the form*

$$n = \sum_{i=0}^{k-1} d_i b^i,$$

where $k > 0$, the d_i are integers such that $0 \leq d_i \leq b-1$ and $d_{k-1} \neq 0$. Moreover, $k = \lfloor \log_b n \rfloor + 1$.

Proof We use the division algorithm (Theorem 2.1) to build the representation. We successively divide:

$$\begin{aligned} n &= bq_0 + d_0, \\ q_0 &= bq_1 + d_1, \\ q_1 &= bq_2 + d_2, \\ &\cdots \\ q_{i-1} &= bq_i + d_i, \end{aligned}$$

and note that $0 \leq d_i < b$ and $n > q_0 > q_1 \cdots \geq 0$. This strictly decreasing sequence of non-negative integers must reach the value 0, so let k be the first index such that $q_{k-1} = 0$. Then the above sequence of identities ends in $q_{k-2} = d_{k-1}$ which is a positive integer. Now, starting at the first identity and replacing each q_i by its value given by the next identity we obtain the required expression for n. The uniqueness of this expression is left as an exercise. Finally, note that $b^{k-1} \leq n < b^k$ and, taking base-b logarithms, we obtain the chain of inequalities $k - 1 \leq \log_b n < k$ which is clearly equivalent to $k = \lfloor \log_b n \rfloor + 1$. □

From this theorem it follows that each positive integer has a representation of the form $n = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_b$ which, if there is no ambiguity, is also written as $d_{k-1}d_{k-2}\ldots d_1d_0$. It is called the *b-adic expansion* of n and its terms are called *digits* while its *length* is k. Notable special cases are the bases $b = 10$ (decimal), $b = 2$ (binary), $b = 8$ (octal) and $b = 16$ (hexadecimal or, simply, hex). If $b \leq 10$, then the base-b digits are represented by the usual decimal digits which are $<b$. In particular, the binary digits, or bits, are 0, 1, and we have the binary expansion $d_{k-1}d_{k-2}\ldots d_1d_0 \in \{0, 1\}^k \subset \{0, 1\}^*$, where $\{0, 1\}^k$ denotes the set of binary strings of length k and $\{0, 1\}^*$ the set of all (finite-length) binary strings. For other bases, additional symbols (usually letters) are used and, for example, hexadecimal digits are given, in increasing order, by 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f (sometimes upper-case letters are used instead of lower-case ones).

The proof of the preceding theorem provides an algorithm to compute the b-adic expansion of an integer n by means of successive integer divisions by b and this algorithm is used whenever we want to do a "base change", for example, for passing from the decimal expansion to the binary one. In some cases this is particularly easy. If $a = b^k$ then conversion from base a to base b is just a matter of replacing each base-a digit by its k-digit representation in base b, and conversely to pass from base b to base a. For example, to convert between hexadecimal and binary we only have to replace each hex digit by a corresponding 4-bit block (sometimes called a *half-byte* or a *nibble*), where 0 goes to 0000, 1 to 0001, and so on, until f which goes to 1111. For the inverse conversion one should group the bits in blocks of 4 and do the replacement, bearing in mind that if the first group from the left has fewer than 4 bits then it should be completed with zeros added to the left. So, for example, the binary

number 1011101 is converted to hexadecimal by replacing 0101 by 5 and 1101 by d, so that its hexadecimal representation is $5d$.

Maple has commands for converting between representations of integers in different bases. The generic conversion function is `convert/base` which converts between two bases a and b by taking the list of digits in base a (if $a = 10$ then the digits can also be given in the usual form instead of as a list) and converting it to the list of digits of the same number in base b. It should be noted that these Maple lists follow the *little-endian* convention in which the most significant digit is the one on the right (the last one on the list). For example,

```
> convert([1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1], base, 2, 16);
                                [9, 11, 4, 11, 3]
```

passes from the list of binary digits to the corresponding list of hex digits. Note that in this case the letters a, b, c, d, e, f are not used for the hex digits and their decimal form (with two decimal digits each) is used instead. These letters are used, however, when converting from decimal to hex by means of another variant of the `convert` function called `convert/hex`. We use this function together with `convert/binary` to convert the binary number whose bits are given by the list above to hex:

```
> convert(111011010010111001, decimal, binary);
                                     242873
> convert(%,hex);
                                     3B4B9
```

When using these functions in Maple, it should be taken into account that these representations of a number are not, in general, valid replacements for the number itself. For example, consider:

```
> x := convert(5, hex);
                                       5
> whattype(x);
                                    symbol
```

In this case, the result of converting the integer 5 to hex is a symbol, i.e., no longer a number and, for example, we could not perform arithmetic operations with it anymore. But even more caution is required with the following conversion:

```
> y := convert(5, binary);
                                      101
> whattype(y), irem(y, 5);
                                  integer, 1
```

Now, the result of converting 5 to binary is indeed an integer but this integer is no longer 5 but 101, i.e., the number whose decimal representation has the same digits as the binary representation of 5 …

2.3 Basic Computational Complexity

As we shall see later, the security of cryptosystems is often based on the assumed hardness of certain computational (mostly number-theoretic) problems. It is therefore important to have procedures to evaluate the hardness of computational problems

and to classify these problems according to their difficulty. An intrinsic estimation of the difficulty of a problem is hard to come by and so usually one has to settle for estimating the cost of solving the problem by means of a specific algorithm. Computational complexity theory classifies algorithms in terms of the resources (such as time, space, etc.) needed to run them. The most important of these resources is time, which is usually evaluated by means of a function that estimates how time varies as a function of input size. This function does not attempt to measure the real time the algorithm will spend on a given input but, rather, the number of *elementary operations* used, which will be roughly proportional to the time actually spent (with the proportionality constant depending on many additional factors, such as the power of the machine or machines running it, the specifics of the implementation, etc.). The important thing here is that this estimation should be precise enough to allow us to predict, given the time required to execute the algorithm with a given input on a concrete machine, how long will it take to run the algorithm with the same implementation and a larger input.

2.3.1 Asymptotic Notation

In order to achieve the estimation just mentioned it will be enough to know the *asymptotic behavior* of the running time function, i.e., the growth rate of this function as the input size gets large. This growth rate might be a complicated function as it will depend on many different factors, but the idea is to approximate it by simpler functions. For this, the following asymptotic notation is used:

Definition 2.8 Let $f, g : \mathbb{N} \to \mathbb{R}^+$ (where $\mathbb{R}^+$ denotes the positive reals; more generally, we can consider functions defined for sufficiently large positive integers that are eventually positive, i.e., $f(x), g(x) > 0$ for sufficiently large x). Then we say that:

1. $f = O(g)$ (f *is big-O of* g) if there exists $c \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $f(x) \leq cg(x)$ for all $x \geq x_0$. In this case we also say that f *is of order* g .
2. $f = \Omega(g)$ (f is *big-Omega* of g) means that there exists $c \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $f(x) \geq cg(x)$ for all $x \geq x_0$, equivalently, $g = O(f)$.
3. $f = \Theta(g)$ (f is *big-Theta* of g) means that there exist $c, d \in \mathbb{R}^+$ and $x_0 \in \mathbb{N}$ such that $cg(x) \leq f(x) \leq dg(x)$ for all $x \geq x_0$, equivalently, $f = O(g)$ and $g = O(f)$.
4. $f = o(g)$ (f *is little-o of* g) if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$, equivalently, for every $\varepsilon > 0$, $f(x) \leq \varepsilon g(x)$ for every sufficiently large x. Intuitively, this means that g grows much faster than f and it clearly implies that $f = O(g)$.
5. $f \sim g$ if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$. In this case we say that f is *asymptotically equal* (or *asymptotic*) to g.

Remarks 2.1

1. The big-O notation denotes a binary relation between the functions f and g that is not symmetric in general, i.e. $f = O(g)$ does not imply $g = O(f)$ or, in other words, $f = O(g)$ does not imply $f = \Omega(g)$. For example, we have that $x\sqrt{x} = O(x^2)$ (interpreting these expressions as functions in the obvious way) but it is not true that $x^2 = O(x\sqrt{x})$. In general, the idea of using the big-O notation is that, whenever we write $f = O(g)$, g should be a simpler function than f that gives us a good approximation to how fast f grows asymptotically.
2. The functions in Definition 2.8 can be functions of several variables, i.e., defined over subsets $X, Y \subseteq \mathbb{N}^k$. Then we write, for example, $f = O(g)$ if there is a positive constant c and an integer t such that for all $(x_1, \ldots, x_k) \in \mathbb{N}^k$, with $x_i \geq t$, $1 \leq i \leq k$, the following conditions hold:

   a. $(x_1, \ldots, x_k) \in X \cap Y$, i.e., both functions are defined on $(x_1, \ldots, x_k)$,
   b. $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$.

Examples 2.1

1. If f is a polynomial of degree d with positive leading coefficient (see Sect. 2.8.2 for the definition of leading coefficient), then $f = O(x^d)$ and, in fact, $f = \Theta(x^d)$.
2. If f and g are polynomials of the same degree and with the same (positive) leading coefficient, then $f \sim g$.
3. If f and g are polynomials with positive leading coefficient and the degree of f is smaller than the degree of g, then $f = o(g)$.
4. When we write $O(g)$, this expression denotes an anonymous function f such that $f = O(g)$ and similarly for the other symbols in Definition 2.8. Thus $O(1)$ denotes a function bounded by a constant and $o(1)$ denotes a function that tends to zero as x tends to infinity. For example, $f(x) \sim g(x)$ is equivalent to $f(x) = (1 + o(1))g(x)$.
5. If ε is a positive constant, then $\ln x = O(x^\varepsilon)$ and, in fact, $\ln x = o(x^\varepsilon)$ too. This is because $\lim_{x\to\infty} \frac{\ln x}{x^\varepsilon} = 0$, i.e., the logarithmic function grows slower than any positive power.
6. If b is an integer ≥ 2 and $f(x)$ denotes the length of the b-adic expansion of x (i.e., the number of base-b digits of x), then $f(x) = O(\ln x)$. This is because, by Theorem 2.3, $f(x) = \lfloor \log_b x \rfloor + 1 \leq \log_b x + 1 = \frac{\ln x}{\ln b} + 1$. This can be interpreted as: "the number of digits grows like the logarithm" (and the logarithm base does not matter because logarithms to different bases are proportional). In fact, it is clear that we also have that $f(x) = \Theta(\ln x)$.
7. The length of an integer n, denoted $\text{len}(n)$ is the number of bits of n and, according to the preceding example, $\text{len}(n) = \Theta(\ln n)$, so that $\text{len}(n)$ and $\ln n$ can be interchanged when appearing within the big-O notation.
8. The big-O notation will be used to estimate the running time of algorithms as a function of the input size. When the input is an integer, the input size is, by definition, $\text{len}(n)$. Thus the running times of these algorithms will be estimated as functions of $\text{len}(n)$ or, equivalently, of $\ln n$.

9. If $\pi(x)$ denotes the number of primes $\leq x$, then $\pi(x) \sim \frac{x}{\ln x}$ and $\pi(x) \sim \mathrm{li}(x)$, where $\mathrm{li}(x) = \int_2^x \frac{dt}{\ln t}$ is the "logarithmic integral". This is the *Prime Number Theorem* (PNT) which is important for cryptography and is explained in more detail in Chap. 6 (see Theorem 6.2).

Exercise 2.3

(i) Prove that if f is a polynomial of degree d with positive leading coefficient, then $f = \Theta(x^d)$.
(ii) Prove that $\mathrm{len}(n) = \Theta(\ln n)$.
(iii) Using l'Hôpital's rule, prove that $\lim_{x\to\infty} \frac{\ln x}{x^\varepsilon} = 0$ for any positive ε, and hence that $\ln x = o(x^\varepsilon)$.
(iv) Prove that if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$ where c is any non-negative constant, then $f = O(g)$ and hence that, in particular, $f \sim g$ implies $f = \Theta(g)$.

2.3.2 *Efficient Computation and $\mathcal{P}$ Versus $\mathcal{NP}$*

Here we give a brief overview of complexity theory, focusing on some aspects of cryptologic interest. For detailed analyses of the complexity of many number-theoretic algorithms we refer to [180] and for a general presentation of complexity theory to [6].

As already mentioned, we are going to use the asymptotic notation to express estimates of the running times of some number-theoretic algorithms which have important cryptologic applications. For this we need to count the number of "elementary operations" performed as a function of input size. The algorithms we are going to deal with will typically accept an integer (or several integers) as input, so the input size(s) will be the binary length(s) of the involved integer(s). We will assume our integers are given in binary and the basic operations will be the arithmetic operations with integers of one bit (see [118, 180] for details). This means that our estimates will be *bit complexity* estimates.

It is not difficult to see that performing addition of two integers n, m can be done in $O(\max\{\mathrm{len}(n), \mathrm{len}(m)\})$ bit operations (where $\mathrm{len}(n)$ denotes, as before, the binary length of n) which is the same as $O(\max\{\ln n, \ln m\})$. Indeed, this follows from the fact that to perform the addition, we scan the binary representation of both integers from right to left adding bit by bit with carry. The same running time estimate applies to subtraction and, for multiplication, note that in order to multiply a k-bit integer m by an l-bit integer n by means of the usual grade-school algorithm, we have to perform at most $l - 1$ additions each of which requires k bit operations, obtaining the estimate $O(kl)$ or, in terms of the logarithms, $O(\ln m \ln n)$.

We should mention here that there are faster algorithms for multiplication, although their advantage is only realized in practice when multiplying very large numbers. Thus, for example, the multiplication algorithm of Schönhage and Strassen (see, for example, [6, 180]) can multiply two k-bit numbers in time $O(k \ln k \ln\ln k)$ which is better than the previous estimate of $O(k^2)$ as it is even better than $O(k^{1+\varepsilon})$ for any $\varepsilon > 0$. The running time of division with remainder, where a k-bit integer is

to be divided by an l-bit integer with $k \geq l$ is $O(l(k-l+1))$ which is also $O(kl)$ but may be significantly less than the latter. Thus, if $a > b$, dividing a by b with quotient q and remainder r requires time $O(\text{len}(b)\text{len}(q))$.

We are now ready to introduce the important notion of *polynomial time* (or polytime as it is sometimes called for brevity).

Definition 2.9 A function $f : \mathbb{N} \to \mathbb{R}^+$ is called polynomial (or polynomially bounded) if $f(x) = O(x^c)$ for some positive integer c. An algorithm is said to be a *polynomial-time algorithm* if its running time is a polynomial function of the input size.

Since we will be mainly interested in algorithms that accept integers as input we remark that, for these algorithms, the previous definition may be reformulated as follows:

Definition 2.10 An algorithm that accepts as input integers $n_1, \ldots, n_t$ (given in positional notation) is said to *run in polynomial time* if there are non-negative integers $c_1, \ldots, c_t$ such that the running time of the algorithm is

$$O(\text{len}(n_1)^{c_1}\text{len}(n_2)^{c_2}\cdots\text{len}(n_t)^{c_t}) = O(\ln^{c_1} n_1 \ln^{c_2} n_2 \cdots \ln^{c_t} n_t).$$

In particular, if the input is a unique integer n, the running time is $O(\text{len}(n)^c) = O(\ln^c n)$ (where $\ln^c n$ is an abbreviated notation for $(\ln n)^c$).

We will sometimes use the notation polylog(n) to denote a generic polynomial function in $\ln n$, so that $O(\text{polylog}(n))$ just means $O(\ln^c n)$ for some constant c. Note that the definition of polynomial time just means that the running time is $O(f(\text{len}(n)))$, where f is a polynomial function.

We see that the algorithms corresponding to the basic arithmetic operations all run in polynomial time. Let us give an example of an algorithm that does not.

Example 2.2 The usual algorithm for computing $n!$ runs on time $O(n^2 \ln^2 n)$. Indeed, we do $n-2$ multiplications in which one of the factors is bounded by $n!$ and the other one bounded by n. The length of $n!$ (which is the product of fewer than n numbers each of which is less than or equal to n) is $O(n \ln n)$ so each of these $n-2$ multiplications requires $O(n \ln^2 n)$ bit operations. Thus the total time for computing $n!$ is $O(n) \cdot O(n \ln^2 n) = O(n^2 \ln^2 n)$. Notice that this algorithm is not a polynomial-time algorithm. If we express the running time as a function of the logarithm of n, we see that this time is $O(e^{2 \ln n} \ln^2 n)$.

Definition 2.11 An algorithm that accepts an integer n as input runs in *exponential time* if its running time is $O(n^c) = O(e^{c \ln n})$ for some constant $c > 0$.[1] More generally, we can say that an algorithm runs in exponential time when it has a time

[1] Thus the algorithm used to compute $n!$ is indeed exponential because $O(\ln^2 n) = O(n^\varepsilon)$ and so the running time of the algorithm is $O(n^2 \ln^2 n) = O(n^{2+\varepsilon})$.

estimate of the form $O(e^{ck})$, where k is the total binary length of the integers to which the algorithm is being applied [119].

Polynomial-time algorithms are considered efficient and computational problems that may be solved by these algorithms are considered "feasible" or even "easy". Of course, a specific polynomial-time algorithm might have a very large exponent and hence not be very efficient, but, in general, almost all polynomial-time algorithms that arise naturally have a low exponent and are really efficient. In addition, computational problems for which no polynomial-time algorithms are known are considered hard, and they are usually hard in practice. Thus the concept of polynomial time seems to capture the algorithms that are efficient in practice and both concepts are regarded as synonymous, as we will do in this book.

Besides the problems for which only exponential-time algorithms are known, there are other intermediate problems which are easier than exponential but harder than polynomial time and, in fact, some of the most important "hard problems" used in cryptography are of this type. In order to deal with algorithms whose complexity is intermediate between polynomial and exponential, the so-called L-notation is often used. Let n be a large positive integer that can be thought of as the input of an algorithm, γ a real number in the interval $[0, 1]$, and $c > 0$ a constant. We define:

Definition 2.12 The *subexponential function* with parameters $\alpha \in [0, 1]$ and $c > 0$ is

$$L_n[\alpha, c] = e^{c(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}.$$

We will, in particular, write $L(n) = L_n[1/2, 1] = e^{\sqrt{(\ln n)(\ln \ln n)}}$.

We see that, in particular:

- $L_n[0, c] = e^{c(\ln \ln n)} = (\ln n)^c$,
- $L_n[1, c] = e^{c(\ln n)} = n^c$.

Thus an algorithm whose running time is $O(L_n[0, c])$ is a polynomial-time algorithm while an algorithm with running time $O(L_n[1, c])$ is exponential. This leads to the following definition:

Definition 2.13 An algorithm which takes as input a positive integer n runs in *subexponential time* if its running time is $O(L_n[\alpha, c])$ for some $0 < \alpha < 1$.

Thus a subexponential algorithm is one whose running time is somewhat intermediate between polynomial and exponential or, more precisely, it is both $\Omega((\ln n)^c)$ for every $c \in \mathbb{R}^+$ (and hence slower than polynomial) and $O(n^c)$, for every $c \in \mathbb{R}^+$ (and hence faster than exponential). We will see that the most efficient algorithms known for certain hard problems on which several public-key cryptographic schemes are based run in subexponential time[2] $O(L_n[1/3, (64/9)^{1/3} + o(1)])$.

We have mentioned that computational complexity theory tries to classify computational problems and the algorithms used to solve them but we have not given a

[2] This estimate is *heuristic* in the sense that it relies on some widely believed but unproven conjectures and it agrees well with the results obtained in practice.

precise definition of what a computational problem is. This requires the formulation of a mathematical model of computation and for this there are several possibilities, with the computational model of choice being based on *Turing machines*. Fortunately, it can be shown that the classification provided by complexity theory is largely independent of the computational model used. More precisely, the *Church–Turing Thesis* basically says that all computational models are equivalent in the sense that a function computable in one model is also computable in the other models and the computational complexities defined by the different models are "polynomially related" so that polynomial time, and hence efficient computation, can be defined independently of the adopted model. Thus we do not explicitly choose any one model and we simply rely on the asymptotic notation previously introduced to give estimates of the hardness of the problems. We refer the interested reader to [6] for more detailed complexity theory notions and to [169, 191] for a more specific study of the relation between complexity and cryptography.

When dealing with computational problems, it is easier to look at *decision problems* which are those that admit only a yes/no answer or, in other terms, are solved by an algorithm that produces as output a unique bit. This might seem too restrictive but, in practice, it is not so because the computational problems we are interested in can often be formulated as decision problems in such a way that an efficient algorithm for the latter gives an efficient algorithm for the former and vice versa.

When classifying computational problems in terms of their relative difficulty it is useful to include them in *complexity classes*, some of which we are now going to define. We will consider only decision problems but other kinds of problems may also be similarly classified.

Definition 2.14 A decision problem belongs to the class $\mathcal{P}$ if there exists a deterministic polynomial-time algorithm that solves it. We usually say that these problems are *solvable in polynomial time*.

There are some decision problems that, while seemingly non-solvable in polynomial time, have the property that a solution can be verified in polynomial time:

Definition 2.15 A decision problem belongs to the class $\mathcal{NP}$ (non-deterministic polynomial time) if a 'yes' answer can be verified in polynomial time given some extra information, called a *certificate* or a *witness*. Similarly, the decision problem belongs to the class co$\mathcal{NP}$ if a 'no' answer can be verified in polynomial time given a certificate.

Example 2.3 (**The integer factorization problem**)
We next give a brief summary of the *integer factorization problem* because of its great importance for cryptography; algorithms to solve this problem are studied in Chap. 6. This is the computational problem of, given a positive integer n as input, computing the prime factorization of n. It can be reduced to the *factorization search problem* which is the following: Given an integer $n > 1$, find a nontrivial factor a of n (i.e, an integer a such that $1 < a < n$ and there exists an integer b with $n = ab$) or determine that no such factor exists (i.e., that n is prime).

The factorization problem can be reduced to the corresponding search problem in the sense that there is a polynomial-time algorithm to transform an algorithm to solve the second problem into an algorithm to solve the first. In order to obtain the factorization of n we would repeatedly call the search algorithm to find factors of n until obtaining all the factors, i.e., the search algorithm would act as a subroutine of the factoring algorithm. The important thing is that, since clearly the number of prime factors of n is $O(\text{len}(n))$, the search algorithm must be called a number of times which is polynomial in the size of n (and the factoring algorithm also has to do the corresponding divisions by the factors found). This means that solving the factorization problem is no harder than solving the search problem, up to a polynomial-time amount of computation. As we shall see, the numbers for which the factorization problem has cryptologic interest are the product of just two primes and, in this case, the factorization algorithm has the same running time as the search algorithm.

Since the definitions of $\mathcal{P}$ and $\mathcal{NP}$ refer to decision problems, we will now consider the *integer factorization decision problem* (see, e.g., [119]) which asks, given positive integers n, k with $k < n$, if there exists a factor a of n such that $2 \leq a \leq k$. Again, we can see that the search factorization problem (and hence also the factorization problem) is reducible in polynomial time to the corresponding decision problem. In this case, the reduction process consists of applying a *binary search* to pinpoint a nontrivial factor (actually the smallest one) by calling the algorithm solving the decision problem $O(\text{len}(n))$ times. Indeed, suppose that $l = \text{len}(n)$, so that 2^l is the smallest power of 2 which is greater than n. We first call the decision problem algorithm with $k = 2^{l-1} - 1$. If the answer is 'no', i.e., if n does not have nontrivial factors $\leq k$, then we know that n is prime and we are done, otherwise we call the decision algorithm again with $k = 2^{l-2} - 1$. If the answer is 'yes' then we know that n has a factor whose binary length is $< l - 1$ whereas if the answer is 'no', then it has a factor of length $l - 1$. Thus we have determined the bit d_{l-2} of the binary expansion $d_{l-2}d_{l-3} \ldots d_1 d_0$ of the factor, which is 0 in the first case and 1 in the second. We continue calling the decision problem, each time halving the size of the interval where the nontrivial factor of n must lie by taking the midpoint of the previous interval as the value of k, so that the factor is in the lower half interval if the answer is 'yes' and in the upper half if the answer is 'no'. Thus the next bit, d_{l-3}, is determined by calling the decision algorithm again with $k = 2^{l-3} - 1$ (in case the previous application of the algorithm gave a 'yes' answer) and with $k = 2^{l-2} + 2^{l-3} - 1$ (in case the previous answer was 'no'). Again, a 'yes' answer means that $d_{l-3} = 0$ and a 'no' answer that $d_{l-3} = 1$ and, after calling the algorithm l times, we will have computed the binary expansion of the smallest nontrivial factor of n (in case n is not prime).

Observe that the first call to the decision algorithm (with $k = 2^{l-1} - 1$) is only to determine whether n is prime or not, and in this case one would rather apply a primality test which is usually much faster than a factoring algorithm: before trying to factor a number one should already know that the number is not prime. Finally, note that, since the factorization search algorithm requires $O(\text{len}(n))$ calls to the

decision problem algorithm, its complexity is equal to the complexity of the latter multiplied by $O(\text{len}(n))$.

The factorization decision problem belongs to the class $\mathcal{NP}$. This is because, if the answer is 'yes', i.e., n has a nontrivial factor $\leq k$, this can be checked in polynomial time using a factor a as a certificate by just dividing a into n. This problem is also in the class co$\mathcal{NP}$: if the answer is 'no', then the complete prime factorization of n can act as a certificate that, just by comparing all the prime factors with k, allows a polynomial time verification of the fact that none of them is $\leq k$. This requires checking that the provided factors are indeed prime (to ensure that there are no smaller factors) but, as we will see in Sect. 6.2, this can also be done in polynomial time.

The following Maple function may be used to simulate an algorithm for the factorization decision problem which, when asked whether an integer n has a factor a such that $2 \leq a \leq k < n$, answers either `true` if this is the case or `false` if there is no such factor. To do this, we will let Maple compute the complete factorization first, but we may use Maple's factoring algorithm as an oracle (or a black box to whose inner workings we do not have access) so that the function answers our queries without revealing the factors of n. The function is the following:

```
> DecisionFactor := proc(n, k)
  evalb(ifactors(n)[2][1][1] <= k)
  end proc:
```

Exercise 2.4 Use the method outlined in the previous example, together with the function `DecisionFactor`, to factor 247. What is the exact number of calls to the function in this case?

Exercise 2.5 Write a Maple function that automates the factorization method used in the previous exercise and, on input a positive integer n, outputs the least positive factor of n.

Example 2.4 ($\mathcal{P}$ **versus** $\mathcal{NP}$)
If a problem is in $\mathcal{P}$ then it is trivially in $\mathcal{NP}$ (and also in co$\mathcal{NP}$), so that we have the inclusion $\mathcal{P} \subseteq \mathcal{NP}$. However, it is not known whether this inclusion is proper or not and it is generally thought that $\mathcal{P} \neq \mathcal{NP}$. This open problem, called the $\mathcal{P}$ *versus* $\mathcal{NP}$ *problem*, is one of the so-called *Millennium Prize Problems* for the solution of each of which the *Clay Mathematics Institute* (see [50]) offers a prize of one million dollars. This problem is highly relevant for cryptography because, as we shall see later on, both private-key and public-key cryptography rely on the existence of *one-way functions*. That one-way functions exist is only a conjecture which, as we will see when discussing RSA, would be proved if we could show, for example, that no polynomial-time factorization algorithm exists. This conjecture implies that $\mathcal{P} \neq \mathcal{NP}$ so that, in other words, if $\mathcal{P} = \mathcal{NP}$ then public-key cryptography as we understand it today would not be possible (and neither would private-key cryptography).

More specifically, in Chap. 7 we will see that a necessary condition for the security of RSA encryption is the hardness of the factorization problem and hence that

$\mathcal{P} \neq \mathcal{NP}$. One might wonder whether $\mathcal{P} \neq \mathcal{NP}$ is also sufficient for the factorization problem to be difficult but this does not seem to be the case. There is a subclass of problems within $\mathcal{NP}$, called $\mathcal{NP}$-complete problems, which have the property that all other problems in $\mathcal{NP}$ can be reduced to any one of them in polynomial time and hence proving that an $\mathcal{NP}$-complete problem is in $\mathcal{P}$ is enough to show that $\mathcal{P} = \mathcal{NP}$. $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$ and there are a great number of them (for example, the *Traveling Salesperson problem* is one) but the factorization decision problem is not believed to be among them. See [6] for a more detailed discussion of these questions.

2.3.3 Running Times of Some Simple Algorithms

We are going to obtain estimates of the running time of a couple of very ancient algorithms and we will perform an experiment using Maple to check how close these estimates are to the actual running time of a specific implementation. As we will see, prime numbers play a crucial role in cryptography and hence it is very important to have efficient algorithms that allow us to distinguish between primes and composites. This is the task of a *primality test*, an algorithm that takes as input a positive integer and outputs `true` in case the integer is prime and `false` otherwise. The distribution of prime numbers and efficient primality tests are studied in Chap. 6 but here, as a preliminary example, we are going to study a couple of primality tests which, although inefficient, are important because they play an auxiliary role in many other algorithms: the *sieve of Eratosthenes* and *trial division*.

2.3.3.1 The Sieve of Eratosthenes

One of the oldest algorithms that can be used for primality testing is the sieve of Eratosthenes, which was known more than 2200 years ago and is still today the most efficient method to generate all the primes less than a given integer. The sieve takes as input a positive integer $n \geq 2$ and builds the list of all the primes $\leq n$ as follows. Start with the list of all the integers between 2 and n. The first number in the list is 2, which is prime, and we start by crossing off all proper multiples of 2. We repeat this process by picking, at each stage, the next number i in the list that is not crossed off (note that i is then necessarily a prime) and crossing off all proper multiples of i. This process is repeated while $i \leq \sqrt{n}$, at which point the numbers that remain without having been sieved out are the primes $\leq n$.

Next we give a Maple implementation of the sieve. To economize on memory only odd integers (plus the number 2) are considered and, in fact, they are not actually stored but the array is filled with zeros instead and the crossing off is implemented by replacing zeros by ones (for which we use the very efficient Maple function `ArrayTools:-Fill`). Moreover, the crossing off of multiples of each found prime number i can be started at i^2 since lower multiples have already been crossed off

during the previous steps. At the end, the positions corresponding to the remaining zeros give the primes in the list.

```
> Eratosthenes := proc(n)
  local s, f, a, i, j, k;
  s := floor((isqrt(n)+1)/2);
  f := floor((n+1)/2);
  a := Array(1..f);
  a[1]:=2;
  for i from 2 to s do
    if a[i] = 0 then
      j := 2*(i^2-i)+1;
      ArrayTools:-Fill(1,a,j-1,2*i-1);
    end if;
  end do;
  for i from 2 to f do
    if a[i] = 0 then
      a[i]:= 2*i-1
    end if;
  end do;
  convert(subs(1=NULL,a),list);
  end:
```

For example, the 10 primes preceding 3000000 (Gauss computed all the primes up to 3000000, so these were presumably the last ones computed by him) are the following:

```
> Eratosthenes(3000000)[-10 .. -1];
  [2999873,2999879,2999897,2999903,2999911,2999921,2999933,2999951,2999957,2999999]
```

Although the sieve can be used as a primality test, it is clearly not a very efficient one due, among other things, to the fact that it achieves much more than required: to test for the primality of n we have to compute all the primes $\leq n$ and check whether n is among them! It is also clear that the sieve requires a lot of memory (more than 20 MB for the preceding Maple computation, where n is rather small). The complexity of the algorithm can be estimated as follows, assuming that all arithmetic operations take constant time which is reasonable in view of the fact that the numbers involved will be machine-size integers—otherwise the memory requirements would be enormous. The number of crossings off for each prime p is $\lfloor \frac{n}{p} \rfloor$, so that the total number is $\sum_{p \leq \sqrt{n}} \frac{n}{p}$. From [180, Theorem 5.10] it follows that

$$\sum_{p \leq n} \frac{n}{p} = n \ln \ln n + O(n),$$

and this gives for the algorithm a running time estimate of $O(n\text{len}(\text{len}(n)))$. This complexity is exponential but, on the other hand, the space requirements of the algorithm are even more daunting. Without going into details, we remark that, by the PNT (see Examples 2.1 or Theorem 6.2), the number of primes less than a given integer can be approximated by the logarithmic integral, given in Maple by the function `Li` so that, for example, there are approximately

```
> Li(2.^1024);
                              2.536315701*10^305
```

primes of 1024 bits or less. Thus this algorithm is very far from being able to deal with, say, 1024-bit primes (as we will see, primes of this size are often used in cryptography). A practical trick to economize memory usage when sieving is to segment the array of numbers from 2 to n. There are many variations on the basic algorithm which, for example, can be modified to output all the primes in an interval $[a, b]$. Moreover, generalizations of the sieve of Eratosthenes are used in many other algorithms, including the most powerful factoring algorithms that we shall discuss later: the quadratic sieve and the number field sieve. For more information about this see [60].

2.3.3.2 Trial Division

Trial division is another old primality test that, in contrast with the sieve, has the advantage that it requires very little memory. The algorithm proceeds by trying to divide n successively by the integers between 2 and $\lfloor\sqrt{n}\rfloor$. If $\sqrt{n}$ is reached without any of them dividing n, then n is declared prime, otherwise, n is composite. In fact, it suffices to trial divide by the primes between 2 and $\sqrt{n}$, for which one has to generate the list of these primes. As in the case of the sieve of Eratosthenes, this algorithm also does more than primality testing for it is actually a factoring algorithm that can output all the prime factors of n.

The following implements trial division (as a primality test) in Maple for integers $\leq 10^{14}$:

```
> primelist := Eratosthenes(10^7):

> TrialDivision := proc(n)
  local s, p, r;
  if n < 2 then
    return false
  end if;
  if 10^14 < n then
    error "number too big"
  end if;
  s := isqrt(n);
  r := 1;
  for p in primelist while r <> 0 and p <= s do
    r := modp(n, p)
  end do;
  evalb(r <> 0)
  end proc:
```

For example, the primes among the last 50 integers preceding 10^{14} are:

```
> select(TrialDivision, [$10^14-50 .. 10^14]);
            [99999999999959, 99999999999971, 99999999999973]
```

The bit complexity of trial division can be easily estimated as follows using the PNT (Theorem 6.2). By this theorem, there are approximately $\frac{n^{1/2}}{\ln n^{1/2}} = O(\frac{n^{1/2}}{\ln n})$ primes less than $\sqrt{n}$. If n is prime, then one division for each of those primes must be performed, with cost $O(\ln^2 n)$. This gives a total cost of $O(n^{1/2} \ln n)$ bit operations and, since $\ln n = O(n^{\varepsilon})$ for $\varepsilon > 0$, the running time of the algorithm can be estimated

as $O(n^{1/2+\varepsilon})$. The running time is, therefore, exponential, and using this test with a 1024-bit number would approximately require—in case the number is prime—

```
> evalf(Li(sqrt(2^1024)));
                              3.788709545*10^151
```

divisions, which is far from feasible. It is not necessary to resort to such big numbers to reach the practical limits of this algorithm. In fact, it has been estimated in [60] that testing the primality of a 50-digit integer this way would require a computational effort equivalent to all the computation done until the beginning of the twenty-first century, i.e., around 10^{24} bit operations. It must be pointed out, however, that trial division is often useful as a first step in factoring algorithms, where it serves to get rid of small factors quickly.

Exercise 2.6 Modify the function `TrialDivision` so that the function itself computes the primes used and does not require `primelist` to be computed independently.

2.3.3.3 An Experiment on Trial Division Timing

We are now going to do an experiment—that may serve as a model for other experiments with different algorithms—to test the running time of trial division in practice and check how well it agrees with the theoretically expected one. To approximate the complexity of the algorithm we want to work with numbers as large as possible and so, to minimize the time and memory required to build the list of primes by means of the sieve of Eratosthenes, we will use an array instead of a list. Another advantage of using arrays throughout is that this will allow us to write functions that are compilable by the external C compiler included with Maple, with a considerable gain in speed; thus this experiment will also serve to introduce new features of Maple as a programming language. In order to achieve these goals we will modify the previous procedure `Eratosthenes` in which, after running the sieve, the initial array contains zeros in the positions corresponding to primes and ones in the positions corresponding to composites. We want an array containing only the sequence of prime numbers less than or equal to n and the next function, `ReduceArray`, converts the larger array to this form.

```
> ReduceArray := proc(n::integer[4],
                      a::Array(datatype=integer[4]), b::Array(datatype=integer[4]))
  local i, j;
  j := 1;
  for i from 2 to n do
    if a[i] = 0 then a[i] := 2*i-1 end if
  end do;
  for i to n do
    if a[i] <> 1 then
      b[j] := a[i];
      j := j+1
    end if
  end do
  end proc:
```

To make it compilable, the preceding function has explicit type declarations of integers as hardware integers. We compile the function, obtaining the function `CompReduceArray`[3]:

```
> CompReduceArray := Compiler:-Compile(ReduceArray):
```

Next we give the new function, called `Sieve`, that implements the sieve and builds the array of prime numbers up to n. Note that in order to build this array we need to know in advance the number of primes $\leq n$. We did not use this knowledge when giving the first version of the sieve as it seems inappropriate to use a much more sophisticated algorithm in order to implement it. Now, however, we are not interested in a function to sieve different intervals but rather in a function that allows us to build an array of primes as large as possible to use in trial division. In this case it makes sense to assume that we know the number of primes as given by the Maple function `numtheory:-pi`. `Sieve` is just a modification of `Eratosthenes`, and now we use `CompReduceArray` in the final stage to produce as output an array containing the primes.

```
> Sieve := proc(n)
  local s, f, a, b, i, j;
  s := floor((isqrt(n)+1)/2);
  f := floor((n+1)/2);
  a := Array(1 .. f, datatype = integer[4]);
  a[1] := 2;
  for i from 2 to s do
    if a[i] = 0 then
      j := 2*i^2-2*i+1;
      ArrayTools:-Fill(1, a, j-1, 2*i-1)
    end if
  end do;
  b := Array(1 .. numtheory:-pi(n), datatype = integer[4]);
  CompReduceArray(f, a, b);
  b
  end proc:
```

We are going to build an array containing all the primes up to 2^{28}. The number of primes in this array is

```
> numtheory:-pi(2^28);
                                14630843
```

i.e., more than 14 million primes. The array containing the primes is then:

```
> primearray := Sieve(2^28):
```

Now we give the function that performs trial division based on this array; we do not compile it because we want it to work, even in 32-bit computers, with integers longer than 32 bits:

```
> TrialDivide := proc(n)
  local s, p, r;
  if n < 2 then
```

[3] Compiled functions like this one are not included in the Maple language files provided on the book's website and must be generated by calling `Compiler:-Compile` from the corresponding examples worksheet, where the appropriate command is already included.

```
    return false
  end if;
  if 2^56 < n then
    error "number too big"
  end if;
  s := isqrt(n);
  r := 1;
  for p in primearray while r <> 0 and p <= s do
    r := modp(n, p)
  end do;
  evalb(r < > 0)
  end proc:
```

The next function measures the time taken by `TrialDivide` when acting on a given integer:

```
> TrialDivideTiming := proc(value)
  local t;
  t := time();
  TrialDivide(value);
  time()-t
  end proc:
```

Now, in order to time the function `TrialDivide`, we build up a sample of primes (we use Maple's function `prevprime` for speed but, of course, we could use `TrialDivide` as well). The reason of taking prime numbers is that these are the integers that require the most time when trial divided (most composite numbers have small prime factors and hence require very little time to be tested).

```
> plist := [seq(prevprime(i*2^52), i = 1 .. 16)];
  plist := [4503599627370449, 9007199254740881, 13510798882111483, 18014398509481951,
            22517998136852473, 27021597764222939,31525197391593467, 36028797018963913,
            40532396646334423, 45035996273704937, 49539595901075453, 54043195528445869,
            58546795155816433, 63050394783186917, 67553994410557429, 72057594037927931]
```

Next we time `TrialDivide` acting on these primes:

```
> tlist := map(TrialDivideTiming, plist);
tlist := [8.047, 11.172, 13.531, 15.531, 17.234, 18.751, 20.266, 21.609, 22.859,
  23.938, 25.109, 26.172, 27.173, 28.171, 29.157, 30.047]
```

We now approximate the time function. We expect this function to be proportional to a power of n, where the exponent should be close to 0.5. Observe that the running time will be really exponential when given as a function of the size of the number (or of its logarithm) although it is just a power when given as a function of n. Because of this we use Maple's function `PowerFit` from the `Statistics` package, which applies the least-squares method to build a model function of the form $a \cdot n^b$ that fits these data points:

```
> t := Statistics:-PowerFit(plist, tlist, n);
                2.91575852460995 10^-7 n^0.475299605601998410
```

The function obtained is approximately $2.92 \cdot 10^{-7} \cdot n^{0.476}$ and we see that the exponent obtained is close to 0.5 as expected. It is reasonable to expect that, when performing these timings with Maple, there is some overhead time. If we subtract a small constant value from the times in `tlist` above and repeat the computation of the model function, we will see that the resulting function is even closer to the theoretically expected one (i.e., the exponent is closer to 0.5).

Exercise 2.7 Experiment by using the `TrialDivideTiming` function with different data sets of integers in order to obtain the timing functions that fit these data. In each case, compare the function obtained with the expected one and check that subtracting some small constant value from the times list gives a timing function which is closer to the theoretically expected one.

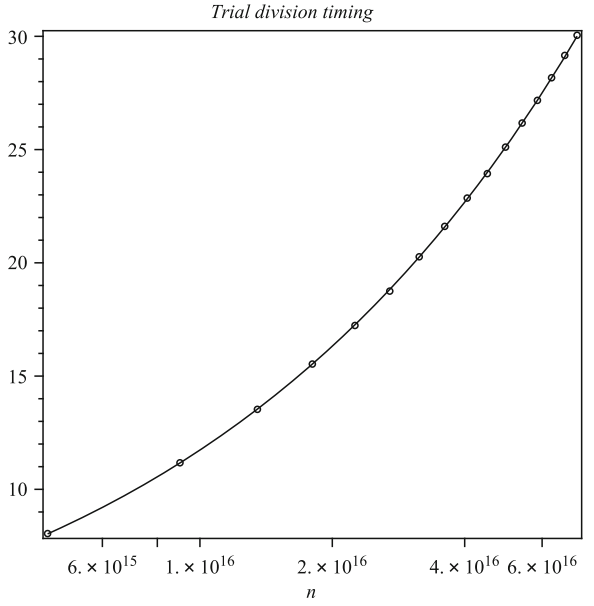
In order to be able to plot the data points we have used, we build the list of these points, where each point will be given as a list of length two, whose first element is an integer and whose second element is the time taken by `TrialDivide` on this integer:

```
l := [ListTools:-LengthSplit(ListTools:-Interleave(plist, tlist), 2)]:
```

Now we plot the curve t together with the set of data points we have obtained; this lets us graphically appreciate that the exponential function t does fit the data well. To plot the list of points we use the function `listplot` in the `plots` package and, to take into account the fact that the running time function grows exponentially in $\mathrm{len}(n)$ (or in $\ln n$) we plot t with the `semilogplot` command (also from the `plots` package), which gives the horizontal axis a logarithmic scale.

```
> with(plots):
  display(title='Trial division timing', {semilogplot(t, n=2^52..2^56, color=black),
  listplot(l, color=black, style=point, symbol=circle, symbolsize=10)}, axes=boxed)
```

The output of this command is the following plot:



2.3.4 Probabilistic Algorithms

The concrete algorithms we have considered so far are *deterministic*, in the sense that their behavior is completely determined by their input, so that if the algorithm is executed multiple times with the same input, it always produces the same output. But, as we shall see, in cryptology one often uses *probabilistic* (*or randomized*) *algorithms*. These algorithms have access to a source of randomness that yields random bits[4] with uniform probability distribution (i.e., where 0 and 1 each have probability $\frac{1}{2}$). Consequently, a probabilistic algorithm may produce a different output each time it is executed with the same input. Examples of algorithms of this type used in cryptography are:

- Algorithms to randomly generate a key of a given size.
- Randomized encryption algorithms which produce different ciphertexts when repeatedly used with the same plaintext and the same key (as we shall see, the use of probabilistic encryption algorithms is crucial to obtain good security).
- Probabilistic algorithms used to test primality, such as the *Miller–Rabin test*, to be discussed later.
- Algorithms to randomly choose a prime number of a given size.

We have already mentioned the relation between polynomial-time and efficient (or, at least, tractable) computation and now we want to point out that probabilistic polynomial-time algorithms (which are generally faster than deterministic ones) are actually the ones usually taken to define efficiency. There are several interesting classes of probabilistic algorithms, among which we mention:

- *Monte Carlo algorithms*. These run in polynomial time but may not always give the correct result. In the case of a decision problem a Monte Carlo algorithm is *yes-biased* if a 'yes' answer is correct with some probability $> 1/2$ but a 'no' answer is always correct. Similarly, *no-biased* Monte Carlo algorithms are defined by interchanging the roles of 'yes' and 'no'. The two-sided versions of Monte Carlo algorithms, which may err on either the 'yes' or the 'no' side, are also called *Atlantic City* algorithms.
- *Las Vegas algorithms*. These algorithms run in *expected* polynomial time, meaning that their running time is a random variable whose expected value is polynomial (see Sect. 2.1) but, in contrast, their output is always correct. However, Las Vegas algorithms may not even provide an output for some inputs.

It is sometimes said that Monte Carlo algorithms are always fast but only probably correct while Las Vegas algorithms, on the contrary, are always correct but only probably fast. As we shall see later, examples of both classes of algorithms are known for the decision problem of determining whether a given integer is prime (in fact, there is also a deterministic polynomial-time algorithm for this problem, but it is significantly less efficient than the probabilistic ones).

[4] It is often metaphorically said that the algorithm tosses "random coins".

Definition 2.16 A decision problem belongs to the class $\mathcal{RP}$ (*randomized polynomial time*) if there exists a no-biased Monte Carlo algorithm to solve the problem. Similarly, the problem is in co$\mathcal{RP}$ if there exists a yes-biased Monte Carlo algorithm that solves it. The intersection $\mathcal{RP} \cap \text{co}\mathcal{RP}$ defines the class $\mathcal{ZPP}$ (*zero-error probabilistic polynomial time*).

It is not difficult to see [6, 191] that the class $\mathcal{ZPP}$ actually consists of the decision problems which can be solved by a Las Vegas algorithm.

Example 2.5 Let us consider the decision problem that asks, given a positive odd integer n, whether n is composite. As we will see, by Theorem 6.7 the Miller–Rabin test is a no-biased Monte Carlo algorithm for this problem and hence the decision compositeness problem is in $\mathcal{RP}$. By the same reason the decision primality problem (given a positive integer n, is n prime?) belongs to $co\mathcal{RP}$. In fact, since the 1990s it has been known that this problem also belongs to $\mathcal{RP}$ and hence to $\mathcal{ZPP}$ but, from a theoretical point of view, all these results have been superseded by the proof in 2002 [3] that the decision primality problem is in $\mathcal{P}$ (taking into account that, clearly, $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq \mathcal{RP}$).

The classes $\mathcal{RP}$ and co$\mathcal{RP}$ are defined by probabilistic polynomial-time algorithms which make only one-sided errors. We can consider also algorithms that can make two-sided errors, i.e., what we referred earlier to as Atlantic City algorithms. This idea leads to the definition of the class $\mathcal{BPP}$:

Definition 2.17 A decision problem belongs to the class $\mathcal{BPP}$ (*bounded-error probabilistic polynomial time*) if there exists a constant $\varepsilon > 0$ and a probabilistic polynomial-time algorithm that produces a 'yes' or 'no' answer which is correct with probability $> 1/2 + \varepsilon$.

We remark that if a probabilistic algorithm gives a correct answer with probability greater than a constant $> 1/2$, then one may obtain from it an algorithm that, for any $\varepsilon > 0$ produces a correct answer with probability greater than $1 - \varepsilon$. In the case of a Monte Carlo algorithm it suffices to take k independent iterations of the original algorithm, where k satisfies $2^{-k} < \varepsilon$. In the case of an Atlantic City algorithm, one considers a sufficient number of iterations of the algorithm and then takes the "majority answer" (see, e.g., [191, Proposition 4.14]). This last remark has the consequence that, although the inclusion $\mathcal{RP} \subseteq \mathcal{BPP}$ is probably proper (and it is suspected that $\mathcal{BPP}$ may be much larger than $\mathcal{RP}$), the problems in $\mathcal{BPP}$ are considered tractable.

Example 2.6 Consider the following problem: Given a positive integer k, find a prime of (at least) k bits in time polynomial in k. We will see that many cryptographic algorithms use large primes so the problem has a definite cryptographic interest. However, no (deterministic) polynomial-time algorithm is known so far to solve this problem! (see [157] for a discussion and references). Despite this apparently unfortunate situation, there is no problem to find large primes for cryptographic use, which can be done efficiently by using probabilistic algorithms. The idea is simply to

choose integers of the required size at random and submit them to a primality test until a prime is found. The primality test used can be either probabilistic or deterministic (with the former being the more efficient by far) but the prime-finding algorithm will be probabilistic anyway because we have to start by choosing random integers. We will see in Chap. 6 that this allows us to find primes in expected polynomial time, the reason being essentially that primes are sufficiently dense among the integers so that if p is the probability that a randomly chosen integer of a given size is prime, then $1/p$ is a polynomial function on the size. Since the expected number of trials to find a prime is equal to $1/p$ by Proposition 2.2 and, since each trial involves a polynomial-time primality test, the algorithm does indeed run in expected polynomial time.

2.3.5 Final Remarks on Complexity

In this section we briefly discuss some aspects of complexity that, although not required to read the rest of the book, are relevant for cryptography and are also of considerable interest from the point of view of theoretical computer science. We also include some bibliographic references for the reader who wishes additional information on the subject.

All the notions of complexity discussed so far are based on evaluating the *worst case* of the computational problems involved. For example, the idea behind the concept of $\mathcal{NP}$-completeness is that an algorithm that can efficiently solve all instances of the problem, including the most difficult ones, would lead to an efficient algorithm to solve all the problems in $\mathcal{NP}$. But there are $\mathcal{NP}$-complete problems that are easy to solve most of the time and this is perfectly compatible with the hypothesis that $\mathcal{P} \neq \mathcal{NP}$, which only implies that there is no efficient algorithm to solve these problems in the worst case [6]. Public-key cryptography is based, as we shall see, on the hypothesis that some computational problems are difficult, but problems that are hard in the worst case but not on average are not good for this purpose and, in fact, one would like to use problems which are hard for most instances. For example, the security of RSA is based on the assumption that most integers that are the product of two randomly chosen large primes are hard to factor. The plausibility of this assumption makes the factorization problem adequate for cryptographic purposes although, as we have already mentioned, it is not thought to be $\mathcal{NP}$-complete. In contrast with this, even if we assume that $\mathcal{P} \neq \mathcal{NP}$, we do not have an a priori guarantee that an $\mathcal{NP}$-complete problem will be hard on average, and hence such a problem may not be suitable for cryptographic use. On the other hand, we will see that the hard problems of cryptographic interest are usually *random self-reducible* (see Definition 7.6) in the sense that any instance can be reduced in polynomial time to a random instance and hence, if the problem is hard in the worst case, then it is also hard on average.

These considerations lead to the study of *average-case complexity*, and it is by no means obvious how to precisely define what it means for a problem to be "intractable

on average". The idea is that to capture the notion of "being efficient on typical instances", one should allow an algorithm to take a large amount of time on inputs that occur with low probability, in such a way that if we have inputs that require increasingly large running time, these inputs should occur with proportionally decreasing probability. This is the idea underlying the definition of average-case complexity given by Levin [129] in the 1980s, which was later modified by Impagliazzo [104]. We refer to these papers and also to [9] for the details.

We have mentioned the fact that computational efficiency can be defined independently of the computational model used. This is true for classical models such as Turing machines or Boolean circuits but does not apply to *quantum computers*. These are computational devices based on the principles of quantum mechanics and there is a celebrated result due to Peter Shor [177] that shows that quantum computers can factor integers (and also solve other related problems) in polynomial time. This poses a lethal threat to RSA and, in fact, to most public-key cryptosystems currently in use. But it is not clear whether full-sized quantum computers—which, so far, exist only as experimental devices with very little capacity—will ever be built and, on the other hand, there are also public-key cryptosytems based on problems (mainly lattice-theoretic ones) which have not been shown to be easy for quantum computers. We say a few more words about this when discussing the current status of integer factorization in Sect. 6.4.10.

2.4 The Euclidean Algorithm

Definition 2.18 Given $a, b \in \mathbb{Z}$ not both zero, the *greatest common divisor* of a and b, denoted $\gcd(a, b)$ is the largest integer that divides both a and b. We say that a and b are *relatively prime* if $\gcd(a, b) = 1$.

The computation of the greatest common divisor is an interesting problem for several reasons. In the first place, this computation is a basic component in many number-theoretic algorithms, as we will see when implementing them. But it is also interesting because it is not at all obvious how to compute, for example, the gcd of two integers that have several hundreds of decimal digits each (as we will see later on, integers of this size are commonly used in cryptography).

A straightforward procedure to obtain the gcd is to list all the common positive divisors of a and b. Both lists will be finite and the largest number on both lists will be $\gcd(a, b)$. But it is intuitively clear that this method will not work with large numbers and we could try a related but more sophisticated method instead, using the fundamental theorem of arithmetic (Theorem 2.2). From this theorem we can derive the following method to compute the gcd of two integers a and b which we can assume, without loss of generality, to be greater than 1. Suppose that $a = \prod_{i=1}^{k} p_i^{e_i}$ and $b = \prod_{i=1}^{k} p_i^{f_i}$, where the primes p_i are written in increasing order and we allow the exponents e_i and f_i to be zero. Then it is obvious that

$$\gcd(a,b) = \prod_{i=1}^{k} p_i^{\min(e_i, f_i)}$$

However, this method is, again, unable to find, in general, the gcd of two integers of several hundred digits each. The reason is that no algorithms are known that can factor integers of this size in reasonable time (the largest 'difficult' integer factored so far is one of 232 decimal digits and it took about 2.5 years to factor it using many workstations; this factorization is discussed in more detail in Sect. 6.4.10). Fortunately, there is a very efficient algorithm to compute the gcd which is able to deal with very big numbers—even numbers much larger than the one just mentioned. This is the Euclidean algorithm, the oldest nontrivial algorithm and hence, according to Knuth [115], "*the granddaddy of all algorithms*". In describing it (and throughout the book) we will denote by $a \mod b$ the remainder in the set $\{0, 1, \ldots b-1\}$ of dividing a by b, also called the *least non-negative residue* of a modulo b.

The basic fact underlying the Euclidean algorithm is the following:

Theorem 2.4 *Let* $a, b \in \mathbb{Z}$ *such that* $a \geq b > 0$. *Then* $\gcd(a,b) = \gcd(b, a \bmod b)$.

The proof is a straightforward consequence of the obvious fact that d is a common divisor of a and b if and only if d is a common divisor of b and $a \mod b$. If $b < a$ (otherwise $b = a$ and $\gcd(a,b) = a$) then, since $a \mod b < b$, this theorem reduces the computation of $\gcd(a,b)$ to computing the gcd of two strictly smaller numbers. Setting $a = r_0$ and $b = r_1$ and repeating this process by computing the remainders $r_i = r_{i+1}q_{i+1} + r_{i+2}$, with $0 \leq r_{i+2} < r_{i+1}$, we obtain a strictly decreasing sequence of non-negative integers $a = r_0 > r_1 > r_2 > \cdots \geq 0$ and hence $r_{n+1} = 0$ for some index $n \leq a$. If n is the least such index we have $\gcd(a,b) = \gcd(r_0, r_1) = \gcd(r_1, r_2) = \ldots \gcd(r_n, 0) = r_n$.

We have just described the Euclidean algorithm, which can be given in pseudocode as a recursive function that calls itself:

Algorithm 2.1. Euclidean algorithm (recursive version).

```
recgcd:
Input: Integers a ≥ b ≥ 0, not both 0.
Output: gcd(a, b).

if b = 0 then
    return a
else
    recgcd(b, a mod b)
end if.
```

An alternative to the recursive version is the iterative version given in Algorithm 2.2.

Maple has a built-in function called `igcd` for computing the gcd of two integers. This function is fast and we use it in many of our functions throughout the book.

Algorithm 2.2. Euclidean algorithm (iterative version).

Input: Integers $a \geq b > 0$.
Output: $\gcd(a, b)$.

```
while b > 0
    r := a mod b
    a := b
    b := r
end while
return a.
```

However, for illustrative purposes, we next give simple implementations of the two versions of the Euclidean algorithm we have just seen. The recursive version of the Euclidean algorithm is implemented in the Maple function `recgcd` below, which is an almost verbatim transliteration of Algorithm 2.1. In order to visualize the numbers to which the algorithm is being applied in the successive steps, we include an optional parameter `verbose` which, if set to `true`, makes the function print these pairs of integers on screen:

```
> recgcd := proc(a::nonnegint, b::nonnegint, verbose := false)
  if verbose then
    print(a,b)
  end if;
  if b = 0 then
    return a
  end if;
  recgcd(b, a mod b, verbose);
  end proc:
```

Example 2.7 Let us use `recgcd` to compute the gcd of two numbers:

```
> recgcd(387349873729876, 4896209096872);
                                        4
```

We will see that the number of steps in applying the Euclidean algorithm to two integers is relatively small and, because of this fact, the algorithm is very efficient. This claim will be made precise when studying the algorithm's complexity but now we use the numbers above to give intuition about these steps. We visualize the recursive calls on screen as follows:

```
> recgcd(387349873729876, 4896209096872, true);
                      387349873729876, 4896209096872
                        4896209096872, 549355076988
                         549355076988, 501368480968
                         501368480968, 47986596020
                          47986596020, 21502520768
                          21502520768, 4981554484
                           4981554484, 1576302832
                           1576302832, 252645988
                            252645988, 60426904
                             60426904, 10938372
                             10938372, 5735044
                              5735044, 5203328
                              5203328, 531716
                               531716, 417884
```

```
417884, 113832
113832, 76388
 76388, 37444
 37444, 1500
  1500, 1444
  1444, 56
    56, 44
    44, 12
    12, 8
     8, 4
     4, 0
      4
```

It is also straightforward to implement the iterative version of Euclid's algorithm in Maple, closely following Algorithm 2.2:

```
> itergcd := proc(a, b, verbose := false)
  local x, y, r;
  x := a;
  y := b;
  while y <> 0 do
    if verbose then
      print(x,y)
    end if;
    r := x mod y;
    x := y;
    y := r;
  end do;
  x;
  end proc:
```

We are going to analyze the complexity of the Euclidean algorithm and, as we are going to see, the worst case is provided by consecutive Fibonacci numbers as was proved by Lamé already in the mid-nineteenth century. TheFibonacci numbers are defined by the linear recurrence

$$F_n = F_{n-1} + F_{n-2},$$

where $F_0 = 0$ and $F_1 = 1$. They can be computed in Maple by means of the command `fibonacci` in the `combinat` package. To simplify the notation, we define an alias for this command and we may use it as follows:

```
> alias(F = combinat:-fibonacci):

> F(1), F(10), F(100);
                        1, 55, 354224848179261915075
```

Let us identify the worst case in the complexity of the Euclidean algorithm:

Theorem 2.5 *Assume $a > b > 0$ and let r_n be the last nonzero remainder in the sequence of integer divisions $r_0 = a, r_1 = b, \ldots, r_{i-1} = r_i q_i + r_{i+1}, \ldots$ Then $a \geq F_{n+2}$ and $b \geq F_{n+1}$.*

Proof Note that, since the sequence of remainders is strictly decreasing, $r_{i-1} > r_i > r_{i+1}$ for $1 \leq i \leq n$ and so $q_i \geq 1$. Moreover, since $r_{n+1} = 0$, we have that $r_{n-1} = r_n q_n$ and so $q_n \geq 2$ (for otherwise we would have $r_{n-1} = r_n$). Thus, working backwards through the sequence of divisions we have that:

$$\begin{aligned} r_n &\geq 1 = F_2, \\ r_{n-1} &\geq 2r_n \geq 2F_2 = F_3, \\ r_{n-2} &\geq r_{n-1} + r_n \geq F_3 + F_2 = F_4, \\ &\vdots \\ r_2 &\geq r_3 + r_4 \geq F_{n-1} + F_{n-2}, \\ b = r_1 &\geq r_2 + r_3 \geq F_n + F_{n-1} = F_{n+1}, \\ a = r_0 &\geq r_1 + r_2 \geq F_{n+1} + F_n = F_{n+2}. \end{aligned}$$

□

Note also that computing $\gcd(F_{n+2}, F_{n+1}) = 1$ by means of Euclid's algorithm takes exactly n steps. Indeed, from the fact that $F_{i+1} = F_i + F_{i-1}$ it follows that all quotients are equal to 1 in this case and the remainders are the Fibonacci numbers in reverse order. Let's do an example with Maple:

Example 2.8 We compute the greatest common divisor of F_{12} and F_{11} and we display the sequence of remainders used in the successive divisions:

```
> itergcd(F(12), F(11), true);
                                144, 89
                                 89, 55
                                 55, 34
                                 34, 21
                                 21, 13
                                 13, 8
                                  8, 5
                                  5, 3
                                  3, 2
                                  2, 1
                                    1
```

We see that the computation indeed took 10 division steps and the output (the gcd) is 1. Next, we compute the sequence of Fibonacci numbers from F_1 to F_{12}:

```
> map(F, [$1 .. 12]);
              [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

We see that the sequence of remainders is indeed the sequence of Fibonacci numbers in reverse order.

We can now give a bound for the number of steps in the Euclidean algorithm.

Theorem 2.6 (Lamé's theorem) *Assume that $a > b > 0$ and let $\varphi = (1+\sqrt{5})/2$ be the golden ratio constant. Then the number of iterations in applying the Euclidean algorithm to a and b is bounded above by $1 + \log_\varphi b$.*

Proof First we show that, for all $n \geq 3$, $\varphi^{n-2} < F_n$. We use induction on n and we start from the fact that this inequality holds for $n = 3$ and $n = 4$ as can easily be checked with Maple:

```
> phi := (1+sqrt(5))*(1/2):

> evalb(evalf(phi) < F(3));
                                    true
> evalb(evalf(phi^2) < F(4));
                                    true
```

Now φ is the positive root of the quadratic equation $x^2 - x - 1 = 0$, so that it satisfies $\varphi^2 = \varphi + 1$ which, multiplying by φ^{n-4} gives the identity $\varphi^{n-2} = \varphi^{n-3} + \varphi^{n-4}$. Our induction hypothesis is then that both $\varphi^{n-4} < F_{n-2}$ and $\varphi^{n-3} < F_{n-1}$ hold. Adding these inequalities we obtain $\varphi^{n-3} + \varphi^{n-4} = \varphi^{n-2} < F_{n-2} + F_{n-1} = F_n$, completing the inductive proof. If applying the Euclidean algorithm to a and b takes n steps, we see from Theorem 2.5 that $b \geq F_{n+1}$ and, by the preceding inequality, we have that $b > \varphi^{n-1}$. Taking logarithms we obtain $\log_\varphi b > n - 1$. □

Exercise 2.8 Show that, for $n \geq 3$, $\varphi^{n-2} < F_n < \varphi^{n-1}$ (use the same induction as in the proof of the preceding theorem, where the first of these two inequalities has already been proved).

Lamé's theorem is also sometimes stated as follows: *the number of division operations needed by the Euclidean algorithm to find the* gcd *of two positive integers is no more than five times the number of decimal digits of the smaller of the two numbers* [194]. This follows from Theorem 2.6 which tells us that if the number of steps is n then $n - 1 < \log_\varphi b$ (where b is the smaller of the two numbers). We also have that $\log_\varphi b = \log_{10} b / \log_{10} \varphi$ and, since $\log_{10} \varphi > 0.2$ we obtain $n - 1 < 5 \log_{10} b$. Then, if b has k decimal digits, we have that $\log_{10} b < k$ and so we get $n - 1 < 5k$ from which it follows, taking into account that both n and k are integers, that $n \leq 5k$.

Now, the complexity of the Euclidean algorithm is an easy consequence of Lamé's theorem:

Theorem 2.7 *The complexity of the Euclidean algorithm applied to integers a, b is* $O(\text{len}(a)\text{len}(b))$.

Proof We can assume that $a > b > 0$ (if this is not the case, then it will be after just one iteration). With the notation previously used, let $(r_i)_{2 \leq i \leq n+1}$ be the remainder sequence and $(q_i)_{2 \leq i \leq n}$ the corresponding quotient sequence. The computation of r_{i+1} by means of an integer division such that $r_{i-1} = r_i q_i + r_{i+1}$ (with $1 \leq i \leq n$) requires time $O(\text{len}(r_i)\text{len}(q_i))$ as we have already remarked. Since $r_i \leq b$ we have that $\text{len}(r_i) \leq \text{len}(b)$ for $1 \leq i \leq n+1$ and, on the other hand, $\text{len}(q_i) \leq 1 + \log_2 q_i$. Thus the time taken by the Euclidean algorithm on input (a, b) can be estimated as

$$O\left(\sum_{i=1}^{n} (\text{len}(b)(1 + \log_2 q_i))\right) = O\left(\text{len}(b)\left(n + \sum_{i=1}^{n} \log_2 q_i\right)\right).$$

Now, it follows from Theorem 2.6 that $n = O(\text{len}(b))$ and, on the other hand, $a = r_0 = r_1 q_1 + r_2 \geq r_1 q_1 = (r_2 q_2 + r_3) q_1 \geq r_2 q_2 q_1 \geq \cdots \geq q_n q_{n-1} \ldots q_1$, which

implies $\sum_{i=1}^{n} \log_2 q_i = O(\text{len}(a))$. Replacing these equalities in the time estimate above we obtain the estimate $O(\text{len}(b)\text{len}(a))$. □

From the Euclidean algorithm one easily obtains that $\gcd(a, b)$ is a linear combination of a and b, namely:

Theorem 2.8 *Let $a, b \in \mathbb{Z}$ such that $a \geq b > 0$. Then there exist $s, t \in \mathbb{Z}$ such that $\gcd(a, b) = sa + tb$.*

To prove this result it suffices to use the Euclidean algorithm to compute $\gcd(a, b)$, keeping all the intermediate results of the divisions and then backtrack by substituting each remainder in terms of the two previous ones until reaching $r_0 = a$ and $r_1 = b$, at which moment one has the required linear combination. A more efficient way to carry out this computation is by using the following algorithm in which, following Klappenecker, we use matrix notation:

Algorithm 2.3. Extended Euclidean algorithm.

Input: Integers $a \geq b > 0$.
Output: Integers d, s, t such that $d = \gcd(a, b) = sa + tb$.

$$\begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ a & b \end{pmatrix};$$

`while` $x \neq 0$
  $q := \lfloor \frac{d}{x} \rfloor$;

$$\begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} := \begin{pmatrix} s & u \\ t & v \\ d & x \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix};$$

`end while`;
`return` (d, s, t).

To see that the algorithm works as expected note that at the start of the loop we have $d = a$ and $x = b$ and, after the first step the new values of d and x are

$$(d, x) = (a, b) \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} = (b, a \bmod b) = (r_1, r_2),$$

where $q := \lfloor \frac{a}{b} \rfloor$ and we have used our earlier notation for the remainders. After the next step we similarly have $(d, x) = (r_2, r_3)$ and upon termination of the loop we have that $(d, x) = (r_n, 0) = (\gcd(a, b), 0)$, so that $d = \gcd(a, b)$.

Furthermore, at the start of the loop, the following equations hold: $as + bt = d$, $au + bv = x$. Subtracting the second equation q times from the first we obtain the equation $a(s - qu) + b(t - qv) = d - qx$ which, taking into account the assignments to the variables u, v and x made in the while loop, means that after each iteration of the loop we still have $au + bv = x$. After the assignments $s := u$, $t := v$, $d := x$ in

the last iteration, the equation $au + bv = x$ becomes $as + bt = d$, giving the desired result.

The extended Euclidean algorithm is implemented in Maple's function `igcdex`. This function accepts as arguments, besides the two integers whose gcd is to be computed, two names `'s'` and `'t'` where the values of the integers s, t mentioned above are to be stored. For example:

```
> igcdex(387349873729876, 48962090968, 's', 't');
                                          4

> s, t, 387349873729876*s + 48962090968*t;
                            -2790500723, 22076265153864, 4
```

As we will see, the Euclidean algorithm plays an important role in cryptology because it is an essential step in many other interesting algorithms. It is also very important that it is a very efficient algorithm, not only in theoretical terms but also in practice. By working backwards in the Euclidean algorithm we may similarly show that the complexity of the extended Euclidean algorithm is also $O(\text{len}(a)\text{len}(b))$.

2.5 Groups, Rings and Fields

In this section we briefly recall some basic algebraic concepts that will be useful in what follows. For more detailed accounts we refer to [45, 48].

2.5.1 Basic Concepts

Recall that a *group* $(G, \cdot)$ is a set G together with a binary operation $G \times G \stackrel{\cdot}{\to} G$ which satisfies the following properties:

- *Associative law*. For all $x, y, z \in G$, $x \cdot (y \cdot z) = (x \cdot y) \cdot z$.
- *Identity element*. There is a (necessarily unique) element $e \in G$ such that, for every $x \in G$, $x \cdot e = e \cdot x = x$.
- *Inverse element*. Each element $x \in G$ has a unique inverse $x^{-1} \in G$, namely, an element x^{-1} that satisfies $x \cdot x^{-1} = x^{-1} \cdot x = e$.

A group is called *abelian* or *commutative* when, in addition, the operation satisfies:

- *Commutative law*. For all $x, y \in G$, $x \cdot y = y \cdot x$.

When the binary operation is understood, we often commit abuse of language and refer to the set G as a group. The symbols most frequently used to denote the group operation are $\cdot$ (multiplicative group) and $+$ (additive group) although, of course, the operation may not be related to the addition or the multiplication of numbers. Additive notation is frequently used for abelian groups and when multiplicative notation is used, the symbol $\cdot$ is often omitted and we write xy to denote the result of

operating x with y. The identity element of a multiplicative group is usually denoted by 1 and the identity element of an additive group is denoted by 0. Moreover, in an additive group, the inverse of an element x is called the *opposite* element of x (or also the *negative* of x) and is denoted by $-x$. For generic groups, we will always use multiplicative notation.

If G and H are groups, then a map $f : G \to H$ is called a *homomorphism* if $f(xy) = f(x)f(y)$ for all $x, y \in G$. Clearly, a homomorphism maps the identity of G to the identity of H and the inverse of an element to the inverse of its image. A bijective homomorphism is called an *isomorphism*. In this case, the inverse $f^{-1} : H \to G$ is also an isomorphism and G and H are said to be *isomorphic*, denoted $G \cong H$. This means that G and H are essentially the "same" group, in the sense that they share the same algebraic properties.

The group G is finite when G is a finite set, and the number of elements of G, denoted $|G|$, is called the *order* of the group G. A *subgroup* of the group G is a subset $H \subseteq G$ such that the operation of G induces by restriction a group structure in H. If G is a finite group and $x \in G$, then the *order* of x is the smallest positive integer i such that $x^i = 1$. The subgroup *generated by* x, i.e., the smallest subgroup of G containing x, consists of the elements $1, x, x^2 \ldots, x^{i-1}$, where i is the order of x. We will denote this subgroup by $\langle x \rangle$ and we say that x is a generator of $\langle x \rangle$. A finite group is called *cyclic*[5] when it has a generator, i.e., an element $g \in G$ such that $G = \langle g \rangle$, which happens if and only if the smallest subgroup of G containing g is G itself. As is easily seen, such groups are always abelian and, if $G = \langle g \rangle$, the order of the element g is equal to the order of the group G.

Examples 2.2

1. Every one-element set $\{x\}$ with the obvious operation is a group (in which x is the identity and its own inverse). This group is isomorphic to the subgroup $\{1\}$ of any group G.
2. $\mathbb{Z}$ is a group with ordinary addition of integers. It is an infinite abelian group and is also cyclic in the sense that the smallest subgroup containing 1 (or -1) is $\mathbb{Z}$ itself but $\mathbb{Z}$ is not a group under multiplication because the only elements that have inverse with respect to this operation are 1 and -1.
3. $\mathbb{R}$ and $\mathbb{C}$ are groups with ordinary addition but they are not groups with respect to multiplication because 0 has no inverse. However, $\mathbb{R}^* = \mathbb{R} - \{0\}$ and $\mathbb{C}^* = \mathbb{C} - \{0\}$ are multiplicative groups. All these groups are abelian but none of them are cyclic.
4. If X is a set, then the set $S(X)$ (also denoted S_X) of all permutations of X is a group under the composition of functions, called the *symmetric group on X*.

Exercise 2.9 Show that if G is a group and H a nonempty subset of G, then H is a subgroup if and only if $xy \in H$ for every $x, y \in H$ and $x^{-1} \in H$ for all $x \in H$. If H is finite then the first condition suffices, i.e., H is a subgroup if and only if $xy \in H$ for all $x, y \in H$.

[5] There are also infinite cyclic groups and they are all isomorphic to the additive group of the integers $\mathbb{Z}$ but in this book we shall be mainly interested in finite groups.

Next we give a couple of elementary properties of the order of an element.

Proposition 2.4 *Let G be a group, $x \in G$ and $t \in \mathbb{Z}$. Then $x^t = 1$ if and only if the order of x divides t.*

Proof If i is the order of x and $t = ik$ for some $k \in \mathbb{Z}$ then $x^t = x^{ik} = (x^i)^k = 1^k = 1$. Conversely, if $x^t = 1$ and i is the order of x, performing the division of t by i we obtain $t = iq + r$ with $0 \leq r < i$. Then $1 = x^t = x^{iq}x^r = x^r$ and, since i is, by definition, the smallest positive integer such that $x^i = 1$, we have that $r = 0$, so that $t = iq$. □

The order of a power may be computed as follows:

Proposition 2.5 *Let G a group, $x \in G$ an element of order i and k an integer. Then x^k has order $i/\gcd(i,k)$.*

Proof Observe that $(x^k)^{i/\gcd(i,k)} = (x^i)^{k/\gcd(i,k)} = 1$. Thus, to prove that $i/\gcd(i,k)$ is the order of x^k it suffices to show that it divides any integer n such that $(x^k)^n = 1$. But if $(x^k)^n = x^{kn} = 1$ we see, by the preceding proposition, that i divides kn. This in turn implies that $i/\gcd(i,k)$ divides n and we are done. □

Next we show how to test whether a given integer n is the order of an element $x \in G$ where G is a group. This will also be useful in finding a generator of G in case G is cyclic, when one applies this to $n = |G|$, but observe that it requires knowledge of the prime factors of n:

Proposition 2.6 *Let n be a positive integer, G a group and $x \in G$. Then n is the order of x in G if and only if $x^n = 1$ and $x^{n/p} \neq 1$ for each prime divisor p of n.*

Proof If n is the order of x then n is the smallest positive integer such that $x^n = 1$ and hence $x^{n/p} \neq 1$ holds for all prime divisors p of n. Conversely, assume that these conditions hold. Since $x^n = 1$ we know from Proposition 2.4 that the order of x divides n. If the order is different from n then it divides some n/p and so $x^{n/p} = 1$, a contradiction. Thus n is the order of x. □

If H is a subgroup of a group G, then H defines an *equivalence relation* on G, namely, a relation which is *reflexive*, *symmetric*, and *transitive*, as follows. Given $x, y \in G$ we say that $x \sim_H y$ if $xy^{-1} \in H$. Then the relation $\sim_H$ is reflexive because $xx^{-1} = 1 \in H$ and symmetric because if $xy^{-1} \in H$ then $yx^{-1} = (xy^{-1})^{-1} \in H$. Transitivity is also clear because if $xy^{-1} \in H$ and $yz^{-1} \in H$ then $xz^{-1} = xy^{-1}yz^{-1} \in H$. The *equivalence class* of $x \in G$, i.e., the set of all elements which are related to x is $Hx = \{hx | h \in H\}$ and, in this case, these sets are called the *right cosets* of H in G. Since any equivalence relation on a nonempty set defines a partition of the set, namely, a decomposition of the set as a disjoint union of the equivalence classes, we have that G is the disjoint union of all the right cosets (this is also easy to prove directly but, for those not familiar with equivalence relations and quotient sets, we refer to [45] or [48]).

Theorem 2.9 (Lagrange's theorem) *If G is a finite group and H a subgroup of G then the order of H divides the order of G.*

Proof We have just noticed that G is the disjoint union of all the right cosets defined by H. Now, observe that all the right cosets have the same number of elements because if Hx and Hy are two different cosets (i.e., x and y are not equivalent in the equivalence relation defined by H) then the map from Hx to Hy that maps hx to hy is a bijection. Thus G is a disjoint union of sets which have $|H|$ elements each (note that H itself is a right coset, namely, the one corresponding to $1 \in G$) and hence $|H|$ divides $|G|$. □

Corollary 2.1 *If G is a finite group and $x \in G$ then the order of x divides the order of G. In particular, $x^{|G|} = 1$.*

Proof The order of $x \in G$ is, as we have seen, the order of the subgroup $\langle x \rangle$ generated by x, so that the order of x divides $|G|$ by Lagrange's theorem. Thus, if i is the order of x we have that $|G| = ti$ for some $t \geq 1$ and hence $x^{|G|} = x^{ti} = (x^i)^t = 1^t = 1$. □

Exercise 2.10 Prove that if H is a subgroup of G, then G is the disjoint union of the left cosets xH, all of which are in bijective correspondence with H.

Exercise 2.11 Show that each subgroup of a cyclic group is also cyclic.

Exercise 2.12 Show that if G is a finite cyclic group and $d||G|$, then there exists exactly one subgroup of G of order d.

If H is a subgroup of G, then the *index* of H in G, denoted $(G : H)$ is defined as the number of right cosets (or, equivalently, of left cosets) of H in G. If G is a finite group and $K \subseteq H \subseteq G$ are subgroups, then it follows from the preceding discussion that $(G : K) = (G : H)(H : K)$. It is also clear that $|G| = (G : \{1\})$ and that $|G| = (G : H)|H|$.

Suppose now that G is an abelian group and H a subgroup of G. Then it is easy to see that the set of cosets, denoted G/H, has the structure of a group with operation given by $HxHy = Hxy$. G/H is then called the quotient group of G modulo H and its order satisfies $|G/H| = |G|/|H|$.

Remark 2.1 Quotient groups may be defined even if the group G is not abelian, provided that H is a *normal* subgroup of G, which means that $xH = Hx$ for every $x \in G$, so that the left and the right cosets coincide. Of course, the subgroups of an abelian group G are all normal.

Let $f : G \to H$ be a homomorphism of groups. Then the *kernel* of f is defined as $\mathrm{Ker} f = \{x \in G | f(x) = 1\} \subseteq G$ and the *image* of f is $\mathrm{Im} f = \{f(x) | x \in G\} \subseteq H$. $\mathrm{Ker} f$ is a normal subgroup of G and $\mathrm{Im} f$ is a subgroup of H. Observe that f is injective if and only if $\mathrm{Ker} f = \{1\}$ (because $f(x) = f(y)$ if and only if $xy^{-1} \in \mathrm{Ker} f$) and that f is surjective if and only if $\mathrm{Im} f = H$.

Proposition 2.7 *Let G be a group and $f : G \to H$ a homomorphism. Then f induces an isomorphism $\bar{f} : G/\mathrm{Ker}\, f \to \mathrm{Im}\, f$.*

Proof Let $K = \mathrm{Ker}\, f$. We define a map $\bar{f} : G/K \to \mathrm{Im}\, f$ by $\bar{f}(Kx) = f(x) \in \mathrm{Im}\, f$, for every $x \in G$. f is well-defined because if $Kx = Ky$, then $xy^{-1} \in K$ and so $f(x)f(y)^{-1} = f(xy^{-1}) = 1$, and multiplying both sides of the equality by $f(y)$ we obtain that $f(x) = f(y)$. Moreover, $\bar{f}$ is clearly a homomorphism and is surjective. On the other hand, $\mathrm{Ker}\, \bar{f} = \{Kx \in G/K \,|\, f(x) = 1\} = \{Kx \in G/K \,|\, x \in K\} = \{K\}$ and hence $\mathrm{Ker}\, \bar{f}$ is the trivial subgroup (its unique element is the identity K of G/K). Thus we see that $\bar{f}$ is indeed an isomorphism. □

Remark 2.2 Note that from Proposition 2.7 it follows that if $f : G \to H$ is a homomorphism, then $(G : \mathrm{Ker}\, f) = |\mathrm{Im}\, f|$. In particular, if f is surjective and $K = \mathrm{Ker}\, f$, then $(G : K) = |H|$.

Next we recall the concept of ring. A *ring* is a set R with at least two elements $0 \neq 1$ and two operations $+, \cdot$, satisfying the following properties:

- $(R, +)$ is an abelian group with identity element 0.
- The multiplication is associative with identity element 1.
- *Distributive laws*. For all $x, y, z \in R$, $x(y + z) = xy + xz$, $(x + y)z = xz + yz$.

If the multiplication of the ring is, furthermore, commutative, then we say that R is a commutative ring. The multiplicative identity 1 is called the *identity* of the ring (sometimes rings without identity are considered but all the rings that we will consider are commutative and with 1). If R is a ring, an element $a \in R$ is a *unit* (or an *invertible element*) if and only if it has a multiplicative inverse, i.e., there exists an element x^{-1} such that $xx^{-1} = x^{-1}x = 1$. It is immediate to see that the set R^* of the units of R is a group with the multiplication induced by that of R (and with identity 1). If R is a commutative ring, an element $x \in R$ is called a *zero divisor* if $x \neq 0$ and there exists an element $y \in R$ such that $y \neq 0$ and $xy = 0$. If x is a zero divisor then x is not a unit of R (note that 0 is neither a unit nor a zero divisor). A commutative ring without zero divisors is called an *integral domain*.

Examples 2.3

1. $\mathbb{Z}$, with the usual addition and multiplication of integers, is a commutative ring. The only units of $\mathbb{Z}$ are 1 and -1 (hence the group of units is $\mathbb{Z}^* = \{-1, 1\} \subseteq \mathbb{Z}$) and there are no zero divisors, so that $\mathbb{Z}$ is an integral domain.
2. $\mathbb{R}$ and $\mathbb{C}$ are rings with ordinary addition and multiplication. All of their nonzero elements are units (a property that, as we shall see below, characterizes the commutative rings that are fields). In particular, these rings are integral domains.

The concepts of ring homomorphism and ring isomorphism are defined similarly to those for groups, as follows:

Definition 2.19 Let R and S be rings and $f : R \to S$ a function. f is a *ring homomorphism* in case it preserves the ring operations and the identity, namely:

1. For all $x, y \in R$, $f(x + y) = f(x) + f(y)$ (i.e., f is a group homomorphism between the additive groups of R and S).
2. For all $x, y \in R$, $f(xy) = f(x)f(y)$.
3. $f(1_R) = 1_S$ (where 1_R and 1_S denote the identity elements of R and S, respectively).

If, moreover, f is bijective, then f is said to be an *isomorphism*.

A class of rings which is especially important is the following:

Definition 2.20 A *field* is a commutative ring with identity in which every nonzero element is invertible.

Example 2.9 $\mathbb{Z}$ is a commutative ring which is not a field because the only invertible elements are 1 and -1. $\mathbb{R}$ and $\mathbb{C}$ are both fields.

Exercise 2.13 Prove that a commutative ring $(R, +, \cdot)$ is a field if and only if $(R - \{0\}, \cdot)$ is a group (in this case $R^* = R - \{0\}$ is called the multiplicative group of R).

Exercise 2.14 Consider the bitwise Xor operation defined on $\mathbb{N}$ as follows. Given two non-negative integers $a, b \in \mathbb{N}$, compute their binary expansions and make them of equal length by adding leading zeros if necessary to the expansion of the smaller number. Then $a \oplus b$ is the integer whose binary expansion is the bit string of the same length which in each position has a 1 if the corresponding bits in the expansions of a and b are different and a 0 if these bits are equal (i.e., the exclusive OR of the corresponding bits of the two expansions is computed). Prove that $(\mathbb{N}, \oplus)$ is an abelian group but show that $(\mathbb{N}, \oplus, \cdot)$ (where $\cdot$ is the multiplication of integers) is not a ring.

2.5.2 Congruences and the Residue Class Ring

All the examples of groups, rings and fields mentioned so far (excluding the trivial group $\{1\}$) are infinite but the examples of cryptographic interest are usually finite. In order to introduce some of the more important ones we start with the definition of congruence.

Definition 2.21 Let $a, b \in \mathbb{Z}$ and $n \in \mathbb{Z}$, $n \geq 2$. We say that *a is congruent to b modulo n*, written $a \equiv b \pmod{n}$, whenever $n | a - b$. If n does not divide $a - b$ then we say that a is not congruent to b modulo n and we denote this fact by $a \not\equiv b \pmod{n}$. The formula $a \equiv b \pmod{n}$ is called a *congruence* and the integer n is the *modulus* of the congruence.

We will not make an exhaustive listing of the properties of congruences, many of which can be found, for example, in [194]. But we recall that the congruence relation modulo n (when viewed as a binary relation on $\mathbb{Z}$) is reflexive, symmetric and transitive, and hence is an equivalence relation. Thus it defines a partition, namely,

it gives a decomposition of $\mathbb{Z}$ as a disjoint union of nonempty equivalence classes which, in this case, are called *congruence classes*. The congruence class or *residue class* of a modulo n is the set $\bar{a} = \{b \in \mathbb{Z} | a \equiv b \pmod{n}\} = \{a + nq | q \in \mathbb{Z}\} = a + n\mathbb{Z}$, where $n\mathbb{Z}$ is the ideal[6] of $\mathbb{Z}$ consisting of all multiples of n. From the definition of congruence, it readily follows that, for $a, b \in \mathbb{Z}$, $a \equiv b \pmod{n}$ if and only if a and b give the same remainder (in the set $\{0, 1, \ldots, n-1\}$) when divided by n, i.e., if and only if $a \bmod n = b \bmod n$.

It is clear then that the set of all congruence classes modulo n has exactly n members and is the set $\mathbb{Z}/n\mathbb{Z} = \{\bar{0}, \ldots, \overline{n-1}\}$. In this set, two binary operations called addition and multiplication are defined in a natural way by setting $\bar{a} + \bar{b} = \overline{a+b}$ and $\bar{a} \cdot \bar{b} = \overline{a \cdot b}$ for $a, b \in \mathbb{Z}$. It is easily checked that these definitions are correct (for the resulting class does not depend on the *representatives* a and b chosen to represent the classes $\bar{a}$ and $\bar{b}$). Moreover, these operations give the set $\mathbb{Z}/n\mathbb{Z}$ the structure of a *commutative ring with identity*, where the required ring properties are derived from those of the addition and the multiplication in $\mathbb{Z}$, of which $\mathbb{Z}/n\mathbb{Z}$ is a *quotient ring* (called the *residue class ring* modulo n). This is just the quotient group $\mathbb{Z}/n\mathbb{Z}$ which, since $n\mathbb{Z}$ is an ideal of the ring, inherits the ring structure of $\mathbb{Z}$.

There is an alternative way of looking at this ring which is perhaps more convenient for our purposes. Let us consider the set $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ (its elements are called the *least non-negative residues* modulo n). The remainder of division of a by n is equal to the unique representative of the congruence class of a modulo n in $\mathbb{Z}_n$, namely $a \bmod n$. Then it is easily checked that the map

$$\begin{array}{ccc} \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}_n \\ \bar{a} & \longmapsto & a \bmod n, \end{array}$$

from $\mathbb{Z}/n\mathbb{Z}$ to $\mathbb{Z}_n$, which assigns to each residue class $\bar{a}$ the least non-negative residue of a modulo n, is a bijection. Moreover, if one defines operations in $\mathbb{Z}_n$ by $a + b = (a + b) \bmod n$ and $ab = ab \bmod n$ (we commit an obvious abuse of notation here), so that now addition and multiplication are just the ordinary addition and multiplication of integers followed by reduction modulo n, then the above map preserves these operations and hence we see that $\mathbb{Z}_n$ becomes a ring (commutative and with 1) which is, actually, isomorphic to the ring $\mathbb{Z}/n\mathbb{Z}$. This canonical isomorphism allows us to identify these two rings and to work with whichever of them is more convenient in a given setting. We shall usually work with $\mathbb{Z}_n$ and we remark that any nontrivial alphabet, i.e., any finite set with more than one element, can be identified with one of these sets and hence can be given the corresponding ring structure.

Remark 2.3 We will often use generic notation for the elements and operations of the ring $\mathbb{Z}_n$. For example, if $a, b \in \mathbb{Z}_n$, we may write $a + b$ for their sum in this ring and we only write $(a + b) \bmod n$ if we want to emphasize that the addition in $\mathbb{Z}_n$ proceeds by adding the elements as integers and then taking the least non-negative residue modulo n. A similar remark applies to the product and other standard notation,

[6] An ideal of a (commutative) ring is an additive subgroup which is closed under multiplication by elements of the ring.

like the one for the opposite of an element $a \in \mathbb{Z}_n$, which is denoted generically by $-a$ and is equal to $-a \bmod n = n - a \in \mathbb{Z}_n$. Also, if $a \in \mathbb{Z}_n^*$ is a unit of the ring $\mathbb{Z}_n$, we will denote its inverse by a^{-1}.

We have just seen that all the $\mathbb{Z}_n$ are rings but, in some cases, they have an additional structure which is useful in many cryptographic settings, namely, they are fields. This happens whenever every nonzero element of the ring has a multiplicative inverse. Thus it is easily seen that $\mathbb{Z}_n$ is a field for $n = 2, 3$ or 5 but it is not a field when $n = 4$ or $n = 6$. In the latter case we observe that, since n is not prime, it has a decomposition $n = rs$ where $r, s > 1$. Then we have that $r, s \in \mathbb{Z}_n$ and, in this ring, $rs = 0$. This means that r and s are zero divisors and it is obvious that a zero divisor cannot have a multiplicative inverse for, otherwise, multiplying both terms of the equation $rs = 0$ by r^{-1} we would obtain $s = 0$, a contradiction. This shows that n being prime is a necessary condition for $\mathbb{Z}_n$ to be a field and, using the preceding results, it is also easy to show that this condition is actually sufficient:

Theorem 2.10 *$\mathbb{Z}_n$ is a field if and only if n is a prime number.*

Proof We have already shown the necessity so, to complete the proof of the theorem, we just have to show that if n is prime then every $0 \neq a \in \mathbb{Z}_n$ has an inverse. By Theorem 2.8 there are integers r, s such that $1 = \gcd(a, n) = sa + tn$. Reducing modulo n we obtain that $1 \equiv sa \pmod{n} \equiv ((s \bmod n) \cdot a) \pmod{n}$ which shows that $s \bmod n$ is the inverse of a in $\mathbb{Z}_n$ and completes the proof. □

More generally we have the following theorem whose proof we leave as an exercise:

Theorem 2.11 *An element $a \in \mathbb{Z}_n$ is a unit if and only if a and n are relatively prime.*

Example 2.10 Modular inverses can be easily computed in Maple. A straightforward rendition of the above proof gives us the following function:

```
> modinv := proc(a::nonnegint, p::posint)
   local d, u;
   d:=igcdex(a, p, 'u') mod p;
   if d<>1 then
     error "%1 does not have an inverse modulo %2", a, p
   end if;
   u mod p;
 end:
```

But Maple already has ways to compute these inverses directly, using built-in functions. For example:

```
> modinv(3865,38984), modp(1/3865, 38984), 1/3865 mod 38984;
                                        817, 817, 817
```

2.6 The Chinese Remainder Theorem

The Chinese remainder theorem is a clever method to solve systems of linear congruences that goes back to the Chinese mathematician Sun Tsu (or Sun Zi) who lived sometime between 200 BC and 200 AD. It is said [162] that it was used by the Chinese generals to count the number of soldiers by lining them in multiples of 7, 10, etc. and counting the remaining soldiers in each case. In his classic three-volume book *Mathematics Manual*, Sun Tsu proposed the following problem: find a number that leaves a remainder of 2 when divided by 3, a remainder of 3 when divided by 5, and a remainder of 2 when divided by 7. This amounts to finding a positive integer satisfying the following system of congruences:

$$\left.\begin{array}{l} x \equiv 2 \pmod 3 \\ x \equiv 3 \pmod 5 \\ x \equiv 2 \pmod 7 \end{array}\right\} \tag{2.1}$$

The general version of the theorem about these systems is as follows:

Theorem 2.12 (Chinese remainder theorem, CRT) *Let $m_1, m_2, \ldots, m_n$ be pairwise relatively prime integers greater than 1 and let $a_1, a_2, \ldots, a_n$ be integers. Then the system of congruences*

$$\left.\begin{array}{l} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \qquad \vdots \\ x \equiv a_n \pmod{m_n} \end{array}\right\} \tag{2.2}$$

has a solution which is unique modulo $m = \prod_{i=1}^{n} m_i$.

Proof For each $i = 1, \ldots, n$, let $M_i = m/m_i = \prod_{j \neq i} m_j$. Since the m_i are pairwise relatively prime we have that $\gcd(m_i, M_i) = 1$ for each i such that $1 \leq i \leq n$. Thus each M_i is invertible in $\mathbb{Z}_{m_i}$ by Theorem 2.11 and, using the extended Euclidean algorithm, we may compute integers $y_i \in \mathbb{Z}$ such that for each $1 \leq i \leq n$,

$$y_i M_i \equiv 1 \pmod{m_i}$$

Then we set $x = (\sum_{i=1}^{n} a_i y_i M_i) \bmod m$ and we claim that x is the unique solution of the system (2.2) modulo m. Indeed, we observe that $y_i M_i \equiv 1 \pmod{m_i}$ implies $a_i y_i M_i \equiv a_i \pmod{m_i}$ for $1 \leq i \leq n$ and, since $m_i | M_j$ for each $j \neq i$, we obtain:

$$a_j y_j M_j \equiv 0 \pmod{m_i}$$

for $1 \leq i, j \leq n$, $i \neq j$. We see that $a_i y_i M_i$ is a solution of the ith congruence and does not contribute to the remaining ones, so the sum of all these terms satisfies

$$x \equiv a_i y_i M_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} a_j y_j M_j \equiv a_i \pmod{m_i}$$

for $1 \leq i \leq n$. This proves the existence and, for the uniqueness, let us note that if x' is another solution of the system, then $x \equiv x' \pmod{m_i}$ and, since the m_i are pairwise relatively prime, this clearly implies $x \equiv x' \pmod{m}$. □

Example 2.11 Let us consider Sun Tsu's problem (Eq. (2.1)). We compute:

$$\begin{aligned} m &= 3 \cdot 5 \cdot 7 = 105, \\ M_1 &= m/m_1 = 105/3 = 35, \quad y_1 = M_1^{-1} \bmod m_1 = 35^{-1} \bmod 3 = 2; \\ M_2 &= m/m_2 = 105/5 = 21, \quad y_2 = M_2^{-1} \bmod m_2 = 21^{-1} \bmod 5 = 1; \\ M_3 &= m/m_3 = 105/7 = 15, \quad y_3 = M_3^{-1} \bmod m_3 = 15^{-1} \bmod 7 = 1. \end{aligned}$$

Thus we obtain the solution:

$$x = \left(\sum_{i=1}^{3} a_i y_i M_i \right) \bmod m = (2 \cdot 2 \cdot 35 + 3 \cdot 1 \cdot 21 + 2 \cdot 1 \cdot 15) \bmod 105 = 23.$$

Of course, this can be also done with Maple, which has the CRT algorithm implemented in the function `chrem`. To solve Sun Tsu's problem we simply compute:

```
> chrem([2, 3, 2], [3, 5, 7]);
                                  23
```

Remarks 2.2

1. An important feature of the proof of the CRT given above is that it is constructive and it actually gives an algorithm for computing the solution(s).
2. The y_i and the M_i defined in the proof of Theorem 2.12 do not depend on the a_i (they depend only on the m_i). This is often useful in applications where the m_i are fixed, for then the y_i and the M_i can be pre-computed and one only has to substitute the a_i in the formula $x = (\sum_{i=1}^{n} a_i y_i M_i) \bmod m$ to obtain the solution in a faster way.
3. The modern interpretation of the CRT places the theorem in the setting of commutative ring theory as we will see in Theorem 2.14 below.

The complexity of the algorithm for the CRT given in the proof of Theorem 2.12 can be estimated as follows:

Theorem 2.13 *The CRT algorithm to solve the congruence system*

$$\left.\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_n \pmod{m_n} \end{aligned}\right\} \tag{2.3}$$

where $m_1, m_2, \ldots, m_n$ are pairwise relatively prime integers greater than 1 and $a_1, a_2, \ldots, a_n$ are integers, runs in time $O(\text{len}(m)^2)$, where $m = \prod_{i=1}^{n} m_i$.

Proof Computing $m = \prod_{i=1}^{n} m_i$ requires time $O(\text{len}(m)(\sum_{i=1}^{n} \text{len}(m_i)) = O(\text{len}(m)^2)$. The computation of all the M_i requires time $O(\text{len}(m)^2)$ as well. Computing the $y_i \in \mathbb{Z}$, $1 \leq i \leq n$, such that $y_i M_i \equiv 1 \pmod{m_i}$ requires, by Theorem 2.7 time $O(\sum_{i=1}^{n} \text{len}(m_i)\text{len}(M_i)) = O((\sum_{i=1}^{n} \text{len}(m_i))\text{len}(m)) = O(\text{len}(m)^2)$. Finally, the computation of $x = (\sum_{i=1}^{n} a_i y_i M_i) \mod m$ also requires time $O(\text{len}(m)^2)$, so the total time estimate is $O(\text{len}(m)^2)$. □

2.6.1 The Chinese Remainder Theorem and the Residue Class Ring

The Chinese remainder theorem has a natural interpretation in terms of the residue class ring $\mathbb{Z}_m$, where $m = \prod_{i=1}^{n} m_i$ is the product of the pairwise relatively prime moduli. First we define the *direct product* of a family of rings (note that the direct product of a family of groups can be similarly defined):

Definition 2.22 Let $\{R_i\}_{i=1}^{n}$ be a family of rings. The *direct product* of this family is the ring $\prod_{i=1}^{n} R_i = \{(x_1, x_2, \ldots, x_n) | x_i \in R_i\}$ with addition and multiplication defined component-wise, i.e., given two tuples $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$, their sum is $(x_1 + y_1, \ldots, x_n + y_n)$ and their product $(x_1 y_1, \ldots, x_n y_n)$.

An alternative notation for the direct product is $R_1 \times R_2 \times \cdots \times R_n$ and, in particular, the direct product of two rings R, S is commonly denoted by $R \times S$.

Example 2.12 Consider the rings $\mathbb{Z}_3$ and $\mathbb{Z}_4$. Then their direct product is the ring $\mathbb{Z}_3 \times \mathbb{Z}_4 = \{(0, 0), (0, 1), \ldots, (2, 2), (2, 3)\}$, which has 12 elements (in general, $|R \times S| = |R| \times |S|$). Note that the identity of this ring is the element $(1, 1)$.

Now, the Chinese remainder theorem can be reformulated as follows:

Theorem 2.14 (Chinese remainder isomorphism) *Let $\{m_i\}_{i=1}^{n}$ be a family of pairwise relative prime integers and $n = \prod_{i=1}^{n} m_i$. Then the map*

$$\begin{array}{ccc} \mathbb{Z}_m & \xrightarrow{f} & \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n} \\ x & \longmapsto & (x \bmod m_1,\ x \bmod m_2, \ldots, x \bmod m_n) \end{array}$$

is a ring isomorphism. Moreover, this map induces by restriction an isomorphism between the unit groups of these rings, namely an isomorphism

$$\mathbb{Z}_m^* \xrightarrow{f} \mathbb{Z}_{m_1}^* \times \cdots \times \mathbb{Z}_{m_n}^*.$$

Proof It is clear that f is a well-defined function. It is also easy to show that f is a ring homomorphism. For example, if $x, y \in \mathbb{Z}_m$ we have that $((x+y) \bmod m) \bmod m_i = (x+y) \bmod m_i = (x \bmod m_i + y \bmod m_i) \bmod m_i$ and a similar formula holds for the product. Now, in order to prove that f is bijective it suffices to show, for example, that f is surjective because both its domain and its codomain have the same number of elements. But, given an element $(x_1, x_2, \ldots, x_n) \in \mathbb{Z}_{m_1} \times \cdots \times \mathbb{Z}_{m_n}$, the Chinese remainder theorem (Theorem 2.12) implies that the system of congruences

$$\left.\begin{array}{l} x \equiv x_1 \pmod{m_1} \\ x \equiv x_2 \pmod{m_2} \\ \qquad \vdots \\ x \equiv x_n \pmod{m_n} \end{array}\right\} \tag{2.4}$$

has a solution which is unique modulo $m = \prod_{i=1}^n m_i$ and this gives an element of $\mathbb{Z}_m$ whose image is $(x_1, x_2, \ldots, x_n)$ (actually, the CRT gives also the injectivity of f because the uniqueness of this solution shows that the element of $\mathbb{Z}_m$ that maps to the given element of the product ring is unique).

For the final part observe that a ring isomorphism such as f has the property that $f(x)$ is a unit if and only if so is x. Indeed, if x is a unit with inverse x^{-1}, then $f(xx^{-1}) = f(1) = 1 = f(x)f(x^{-1})$, so that $f(x)$ is also a unit and the converse follows in a similar way using the inverse isomorphism. Thus the last claim in the statement of the theorem follows bearing in mind that the unit group of the product ring is $\mathbb{Z}_{m_1}^* \times \cdots \times \mathbb{Z}_{m_n}^*$ as it is clear that an element in the product ring is a unit if and only if each of its components is a unit in the corresponding factor ring. □

Exercise 2.15 Show that Theorems 2.12 and 2.14 are equivalent in the sense that not only is the latter a corollary of the former as we have seen but also the former can be deduced from the latter.

Exercise 2.16 Show that the ring $\mathbb{Z}_4$ is not isomorphic to the product $\mathbb{Z}_2 \times \mathbb{Z}_2$ and that their unit groups are not isomorphic either.

As we will see later, one interesting application of the CRT is the speed-up of RSA decryption. Another interesting application is in multiprecision arithmetic and in the construction of the so-called *residue computers* that use the CRT to subdivide large computations in $\mathbb{Z}_{\prod_{i=1}^n m_i}$ into several computations in the smaller rings $\mathbb{Z}_{m_i}$. This can provide a considerable efficiency gain because, on the one hand, the computations in

these rings can be done completely in hardware, without having to use multiprecision arithmetic and, on the other, they can easily be parallelized.

Exercise 2.17 Write a Maple function that computes $n!$ for a large integer n (say $n > 10000$) by computing the result modulo a sufficient number of primes preceding 2^{31} (which may be estimated by means of Stirling's formula, see [94]). These computations may be carried out with machine-size integers of type `integer[4]` and then the CRT may be used to compute the final result.

2.7 Euler's Theorem and Modular Exponentiation

2.7.1 Euler's Theorem

We have seen that the units of a commutative ring form an abelian group and, in particular, the group of units of $\mathbb{Z}_n$ is, by Theorem 2.11 , $\mathbb{Z}_n^*=\{a \in \mathbb{Z}_n | \gcd(a, n)=1\}$.

There is a result of Euler related to this group which, after being proved in the first half of the eighteenth century, found an interesting cryptographic application more than two centuries later for, as we shall see, it is crucial for the RSA encryption scheme. In order to state and prove this result we need the following concept:

Definition 2.23 Let $n \in \mathbb{Z}, n \geq 2$. The Euler ϕ-function (also called totient function) of n is defined by:

$$\phi(n) = |\{x \in \mathbb{Z} | 1 \leq x \leq n \text{ and } \gcd(x, n) = 1\}|.$$

In other words, $\phi(n)$ is the order of $\mathbb{Z}_n^*$, i.e.

$$\phi(n) = |\mathbb{Z}_n^*|.$$

Remark 2.4 Note that, although we do not consider congruences modulo 1, the definition of Euler's ϕ-function extends naturally to all positive integers by setting $\phi(1) = 1$.

Euler's theorem is then the following:

Theorem 2.15 (Euler's theorem) *Let a and n be positive integers such that $n \geq 2$ and $\gcd(a, n) = 1$. Then $a^{\phi(n)} \equiv 1$ (mod n).*

Proof Observe that the statement of this theorem may be formulated in terms of the group $\mathbb{Z}_n^*$. Saying that $\gcd(a, n) = 1$ is equivalent to saying that $a \bmod n$ is an element of $\mathbb{Z}_n^*$. The statement of the theorem is then equivalent to saying that, in the group $\mathbb{Z}_n^*$ we have $(a \bmod n)^{\phi(n)} = 1$, and this follows from Corollary 2.1. □

Exercise 2.18 Use Euler's theorem to compute the last (decimal) digit of 7^{2010}. (Hint: Apply Euler's theorem modulo 10.)

Observe now that, whenever p is a prime number, $\mathbb{Z}_p^* = \mathbb{Z}_p - \{0\}$ and hence $\phi(p) = p - 1$. This gives the following corollary of Euler's theorem:

Corollary 2.2 (Fermat's Little Theorem) *Let p be a prime number and a an integer such that p does not divide a. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Corollary 2.3 *Let p be a prime and a, e, positive integers such that p does not divide a. Then $a^e \equiv a^{e \bmod (p-1)} \pmod{p}$.*

Proof We have that $e = (p-1)q + e \bmod (p-1)$ for some non-negative integer q. Then, by Fermat's little theorem, $a^{p-1} \equiv 1 \pmod{p}$ and we have:

$$a^e \equiv a^{(p-1)q} a^{e \bmod (p-1)} \equiv a^{e \bmod (p-1)} \pmod{p}.$$

□

Exercise 2.19 Use the previous corollary to find the last decimal digit of 7^{2574}. (Hint: Compute $7^{2574} \bmod 2$ and $7^{2574} \bmod 5$ and use these values to compute $7^{2574} \bmod 10$.)

Remark 2.5 We see that Euler's theorem is really a particular case of Lagrange's theorem for groups, obtained by applying the latter—or its Corollary 2.1—to the group $\mathbb{Z}_n^*$ and similarly for Fermat's little theorem and the group $\mathbb{Z}_p^*$, where p is prime.

Example 2.13 In Maple, Euler's ϕ-function is given by the command `numtheory:-phi`. For example, the values of ϕ on the first 20 positive integers are (`map` may be used instead of ~ in Maple versions prior to v13):

```
> numtheory:-phi~([$1 .. 20]);
      [1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18, 8]
```

We show that the ϕ-function gives the number of generators in a finite cyclic group:

Theorem 2.16 *Let G be a finite cyclic group of order n and $g \in G$ a generator. Then the generators of G are the elements g^t with $t \in \{1, \ldots, n-1\}$ such that $\gcd(t, n) = 1$. In particular, G has $\phi(n)$ generators.*

Proof Since $G = \langle g \rangle$ and $|G| = n$, we have that $G = \{g^t | 0 \leq t < n\}$. Among these elements, the generators are those with order n which, by Proposition 2.5 are the g^t such that $n/\gcd(n, t) = n$, which in turn is equivalent to $\gcd(n, t) = 1$. The number of generators is then equal to $\phi(n)$ by definition of ϕ. □

Exercise 2.20 Prove that, for each integer $n \geq 2$, the additive group $\mathbb{Z}_n$ is cyclic and its generators are the elements $g \in \mathbb{Z}_n$ such that $\gcd(g, n) = 1$.

To derive a formula to compute ϕ we first observe that it is a *multiplicative function*, namely:

Theorem 2.17 *If n_1 and n_2 are relatively prime positive integers ≥ 2 then $\phi(n_1 n_2) = \phi(n_1)\phi(n_2)$.*

Proof This follows from Theorem 2.14 which, in particular, gives a bijection between $\mathbb{Z}^*_{n_1 n_2}$ and $\mathbb{Z}^*_{n_1} \times \mathbb{Z}^*_{n_2}$. □

The preceding theorem, together with the fundamental theorem of arithmetic, reduces the computation of $\phi(n)$ to the case in which n is a prime power. If $n = p^k$, where p is prime and $k > 0$, the only numbers between 1 and n which are not relatively prime to n are $p, 2p, 3p, \ldots, p^{k-1}p = p^k$, so that $\phi(p^k) = p^k - p^{k-1}$. From this, using the multiplicativity of ϕ, we have:

Theorem 2.18 *If $n = \prod_{i=1}^k p_i^{e_i}$ is the prime factorization of n, then*

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right).$$

Proof Bearing in mind the previous remark we have that $\phi(n) = \prod_{i=1}^k \phi(p_i^{e_i}) = \prod_{i=1}^k (p_i^{e_i} - p_i^{e_i - 1}) = \prod_{i=1}^k (p_i^{e_i} - \frac{p_i^{e_i}}{p_i}) = \prod_{i=1}^k p_i^{e_i}(1 - \frac{1}{p_i}) = n \prod_{i=1}^k (1 - \frac{1}{p_i})$. □

We next record a useful consequence of the multiplicativity of the ϕ-function:

Proposition 2.8 $\sum_{d|n} \phi(d) = n$.

Proof Let $f(n) = \sum_{d|n} \phi(d)$, where d runs on the set of the positive divisors of n. We have to show that $f(n) = n$ and, in order to do that, we first show that f is a multiplicative function, i.e., that $f(n_1 n_2) = f(n_1) f(n_2)$ whenever $\gcd(n_1, n_2) = 1$. Observe that the divisors of $n_1 n_2$ can be written in a unique way in the form $d_1 d_2$ where $d_1 | n_1$ and $d_2 | n_2$ and so, bearing in mind that ϕ is multiplicative we have:

$$f(n_1 n_2) = \sum_{d_1|n_1, d_2|n_2} \phi(d_1)\phi(d_2) = \left(\sum_{d_1|n_1} \phi(d_1)\right)\left(\sum_{d_2|n_2} \phi(d_2)\right) = f(n_1) f(n_2)$$

Now let $n = \prod_{i=1}^r p_i^{e_i}$ be the prime factorization of n. Then, by the multiplicativity of f we have that $f(n) = \prod_{i=1}^r f(p_i^{e_i})$, so that it suffices to prove that f is the identity on prime powers, i.e., that $f(p^e) = p^e$ whenever p is prime. But, as we have seen, $\phi(p^i) = p^i - p^{i-1}$ for $i \geq 1$ and hence we have:

$$f(p^e) = \sum_{i=0}^{e} \phi(p^i) = 1 + \sum_{i=1}^{e} (p^i - p^{i-1}) = p^e,$$

which completes the proof. □

2.7.2 Modular Exponentiation

One of the key ideas underlying the birth of public-key cryptography in the 1970s was to use as encryption function a trapdoor one-way function, which can informally be described as a function that is easy to compute but hard to invert (namely, a one-way function, which will be formally defined later) and, moreover, has a trapdoor, i.e., information whose knowledge allows the efficient inversion of the function. The first candidate for such a function was proposed in 1978 by Rivest, Shamir and Adleman as the basis of what would become the RSA cryptosystem. The underlying hypothesis on which RSA is based is the following:

- *Modular exponentiation with fixed exponent and modulus is a trapdoor one-way function*

Let us analyze the meaning of this assertion in some detail. The idea is that, if n is a large integer (where the meaning of 'large' will be made precise when discussing RSA) which is the product of two distinct primes p and q of approximately the same size, and e is a positive exponent such that $\gcd(e, (p-1)(q-1)) = 1$ (where the latter condition just means that $e \in \mathbb{Z}^*_{\phi(n)}$), then the function

$$\begin{array}{ccc} \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ m & \mapsto & m^e \bmod n, \end{array}$$

is trapdoor one-way. This means that the following conditions should hold:

1. Computing $c = m^e \bmod n$ is easy (in the complexity-theoretic sense of the term).
2. The inverse problem of *extracting* eth *roots modulo* n, that is, 'given a random $c \in \mathbb{Z}_n$ and a positive integer $e \in \mathbb{Z}^*_{\phi(n)}$ (we may also assume that $1 < e < \phi(n) - 1$ to exclude trivial cases), compute m such that $c = m^e \bmod n$' is hard.
3. If the prime factorization of n is known, then the computation of eth roots modulo n is easy (so that the factorization of n would be the trapdoor).

As we shall soon see, conditions 1 and 3 above do indeed hold and the only thing that remains to be proved in order to ensure that modular exponentiation is a trapdoor one-way function is condition 2. Because of this, the assertion that the function is one-way is so far only a conjecture and, more generally, the existence of one-way functions—which, if true, would imply the truth of the $\mathcal{P} \neq \mathcal{NP}$ conjecture—has not been proved. But it is widely believed that some candidates such as modular exponentiation are indeed trapdoor one-way and this is the reason why they are currently used in cryptography.

We leave for later the discussion of conditions 2 and 3 above and we shall now examine condition 1, which is easily shown to hold. To prove 1 we have to show that modular powers can be efficiently computed and, as we will see when discussing RSA, this also implies 3. More precisely, in terms of complexity theory we will show that modular exponentiation can be done in polynomial time. It is not completely obvious how to do this and, to analyze the difficulties that may arise, let us try to compute an instance with Maple.

Example 2.14 Let us try to do a modular exponentiation by means of the naive algorithm consisting in multiplying the basis by itself $e - 1$ times (where e is the exponent):

```
> 21293^458976453419787627567139383 9 mod 29848327924656663721619585710 3;
Error, numeric exception: overflow
```

Here, Maple was trying to compute $21293^{458976453419787627567139383 9}$ before reducing it modulo $n = 29848327924656663721619585710 3$. But this power is an enormous number because, as can be seen by taking the logarithm, the size of a power grows proportionally to the exponent, or, in other words, exponentially. Thus this number does not fit in the computer's memory producing the overflow error. This problem is easily corrected because all operations can be carried out modulo n, i.e., after each multiplication, the remainder of the result divided by n can be computed and the multiplication result can be replaced by this remainder, so that no numbers greater than $n - 1$ are ever multiplied. Let us try this in Maple:

```
> z := 1:
  for i to 458976453419787627567139383 9 do
    z := 21293*z mod 29848327924656663721619585710 3
  end do:
  z;
Warning,  computation interrupted
```

Now memory was not a problem but we had to stop the computation because Maple would be unable to complete it in our lifetime. Actually, we were asking Maple to carry out 458976453419787627567139383 8 multiplications and the same number of divisions by 29848327924656663721619585710 3. Even assuming a computer powerful enough for Maple to carry out one billion of these multiplication/division pairs per second, this computation would take:

```
> 458976453419787627567139383 8./(365*24*3600*10^9);
                                1.455404786*10^11
```

years. Taking into account that the age of the universe is variously estimated at between 12 and 14 billion years (i.e., less than $14 \cdot 10^9$ years), we see that the entire life of the universe would not be enough for this computation. Moreover, it should be taken into account that the involved numbers in the example are relatively small and, at any rate, much smaller than the numbers used in cryptographic applications. The naive exponentiation algorithm requires a number of multiplications (and divisions) that is just one less than the exponent. Thus the total number of operations is proportional to the exponent (and not to its size!) and hence the algorithm runs in exponential time.

Fortunately, we do not have to use the algorithm in the preceding example since there is a much more efficient algorithm which runs in polynomial time, with the number of operations bounded by a multiple of the number of binary digits in the exponent. This algorithm is variously known as *fast exponentiation*, *modular exponentiation*, *binary exponentiation*, *square and multiply algorithm* … and the basic idea is the following. Suppose that we want to compute $m^e \bmod n$ and that the binary expansion of the exponent is $e = (e_k e_{k-1} \dots e_1 e_0)_2 = \sum_{i=0}^{k} e_i 2^i$ where $e_i \in \mathbb{Z}_2$. Then

$m^e \bmod n = m^{\sum_{i=0}^{k} e_i 2^i} \bmod n = \prod_{i=0}^{k} (m^{2^i})^{e_i} \bmod n = \prod_{0 \le i \le k,\ e_i = 1} m^{2^i} \bmod n$. This suggests the following algorithm:

Algorithm 2.4. Modular exponentiation algorithm.

Input: $n \ge 2$, $m \in \mathbb{Z}_n$, and $e \ge 0$.
Output: $m^e \bmod n$.

1. `Compute` the binary expansion of e, $e = (e_k e_{k-1} \dots e_1 e_0)_2 = \sum_{i=0}^{k} e_i 2^i$.
2. `Compute` the powers $m^{2^i} \bmod n$ for $0 \le i \le k$ by successive squarings (note that $m^{2^{i+1}} = (m^{2^i})^2$).
3. `Compute` $m^e \bmod n$ as the product of those $m^{2^i} \bmod n$ for which the corresponding bit is $e_i = 1$.
4. `return` $m^e \bmod n$.

As we are going to see, this algorithm is implemented in Maple and allows the computation of extremely large exponentiations with great efficiency. But, before discussing Maple's built-in function for this purpose, we are going to exhibit a straightforward implementation that makes it easy to understand the simplicity and elegance of the algorithm. The algorithm can be rendered in Maple as follows:

```
> ModExp := proc(m::nonnegint, e::nonnegint, n::posint)
  local z, y, t;
  if m = 0 and e = 0 then
    error "0^0 is undefined"
  end if;
  z := m;
  y := 1;
  t := e;
  while 0 < t do
    if irem(t, 2, 't') = 1 then
      y := irem(y*z, n)
    end if;
    z := irem(z*z, n)
  end do;
  y
  end proc:
```

In this function the operations described in Algorithm 2.4 are carried out in a slightly different order. In the while loop, the binary expansion of e is being computed by means of successive divisions by 2 and, after computing each bit, a multiplication by m followed by a squaring is done if the corresponding bit is 1, while if the bit is 0 only the squaring is performed; note that in this process the bits of e are being scanned in the same order as they are generated, i.e., from the least significant to the most significant one or, in other words, from 'right to left'. It is then clear that, discounting the initial iteration of the loop which is trivial as at most it only requires multiplying by 1 and squaring 1, if e has $k+1$ bits and exactly $r+1$ of these bits are equal to 1, then the total number of modular operations (either multiplications or squarings) performed is $k+r$. This amounts to a total of $k+r$ integer multiplications of numbers less than n and the same number of divisions to reduce modulo n. To estimate the complexity of this algorithm note that the number of iterations of the

while loop is the number of bits of e which is $O(\text{len}(e))$. Each modular operation requires $O(\ln^2 n) = O(\text{len}(n)^2)$ bit operations (since $m < n$). Furthermore, we will usually assume that $e < n$ (since, by Euler's theorem, if $\gcd(m, n) = 1$ then $m^e \bmod n = m^{e \bmod \phi(n)} \bmod n$) and hence $O(\text{len}(e)) = O(\text{len}(n))$, so that the running time of the algorithm is $O(\text{len}(n)^3)$.

We are now going to explore a couple of variants of the binary exponentiation algorithm. Since they are quite simple, we do not describe them in detail, and we just give Maple implementations from which the structure of these variants is easily deduced. We start with a recursive version of the algorithm which can be easily implemented in Maple as follows:

```
> RecModExp := proc(m::nonnegint, e::nonnegint, n::posint)
  if e = 0 then
    if m <> 0 then
      return 1
    else
      error "0^0 is undefined"
    end if
  end if;
  if m = 0 then
    return 0
  end if
  if irem(e,2)=0 then
    return RecModExp(m^2 mod n, iquo(e,2),n)
  end if;
  (m*RecModExp(m^2 mod n, iquo(e,2),n)) mod n;
  end proc:
```

A slightly different version of the algorithm is the 'left to right' version in which the bits of the exponent are scanned from the most significant to the least significant one (note that Maple uses the little-endian convention so that when using Maple's `convert/base` function the bits of the list produced by it are actually scanned from right to left). A Maple procedure implementing this variant is given in the next function `ExpMod`, where we assume that $n > 2$ and $m < n$. Here we use Maple's powerful left-fold operator `foldl` to perform the sequence of squarings and multiplications (see Maple's help for a description of this function) but it is easy to replace the use of `foldl` by a '`for` loop' performing the same operations:

```
> ExpMod := proc(m::nonnegint, e::nonnegint, n::posint)
  local bits, x, y;
  if e = 0 then
    if m <> 0 then
      return 1
    else
      error "0^0 is not defined"
    end if
  end if;
  bits := op(ListTools:-Reverse(convert(e, base, 2)[1 .. -2]));
  foldl((x, y) -> modp(modp(x^2, n)*m^y, n), m, bits)
  end proc:
```

Exercise 2.21 Write a version of the function `ExpMod` that, instead of using the command `foldl`, uses a `for` loop.

Of course, Maple also has a built-in function implementing the binary exponentiation algorithm, namely `Power`, the inert power function, which is also represented by

the 'infix operator' &^. As was to be expected, this function is very fast and we will use it extensively in the sequel as a component of many of the algorithms implemented.

Example 2.15 Let us consider again the exponentiation we tried to compute in Example 2.14. Now we see that any of the variants of the binary method is able to carry out this computation quickly:

```
> 21293&^458976453419787627567139383 9 mod 29848327924656663721619585710 3;
  ModExp(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
  RecModExp(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
  ExpMod(21293, 458976453419787627567139383 9, 29848327924656663721619585710 3);
                    157396720784600298623131794157
                    157396720784600298623131794157
                    157396720784600298623131794157
                    157396720784600298623131794157
```

2.7.2.1 Speeding up Modular Exponentiation with the Chinese Remainder Theorem

As we shall see, modular exponentiation is the basic operation one has to perform when doing RSA encryption or decryption. In the case of decryption, the modulus is a product of two known large primes and the speed of the binary exponentiation algorithm can be significantly increased. Indeed, given a modulus whose prime factorization is known, one can apply the Chinese remainder theorem to recover the final result from the smaller exponentiations corresponding to the different prime-power factors of the modulus. We next describe how to do this in Maple when the modulus is—as in the RSA case—a product of two different primes.

Suppose that we want to compute $x = m^e \bmod n$, where $n = pq$, with p and q distinct primes not dividing m. Let $x_p = x \bmod p$ and $x_q = x \bmod q$. Then we have that $x_p = (m^e \bmod n) \bmod p = (\text{since } p|n)\ m^e \bmod p = (\text{by Corollary 2.3})\ m^{e \bmod p-1} \bmod p$ and, similarly, $x_q = m^e \bmod q = m^{e \bmod q-1} \bmod q$. Thus the Chinese remainder theorem implies that $x = m^e \bmod n$ is the unique solution modulo n of the system of congruences:

$$\left.\begin{aligned} x &\equiv x_p \pmod{p} \\ x &\equiv x_q \pmod{q} \end{aligned}\right\} \tag{2.5}$$

The computations of x_p and x_q require $O(\text{len}(p)^3)$ and $O(\text{len}(q)^3)$ bit operations respectively. If we assume that p and q have approximately the same length, we will have $\text{len}(p) \approx \text{len}(q) \approx \frac{1}{2}\text{len}(n)$ and hence each of these two computations requires approximately one-eight of the time required by the direct computation of $x = m^e \bmod n$ in time $O(\text{len}(n)^3)$. Thus the time taken by the computations of x_p and x_q will be approximately one-fourth of the time required by computing x directly. Since the time taken to apply the CRT is $O(\text{len}(n)^2)$, which is faster, we see that the use of the CRT makes the process almost four times faster when n is large (in other words, the computation that uses the CRT will require time roughly

proportional to $4\text{len}(p)^2 + 2\text{len}(p)^3$ while the standard computation will require time proportional to $8\text{len}(p)^3$).

This algorithm is implemented in Maple as follows:

```
> ChineseModExp :=
  (m,e,p,q) -> chrem([Power(m,e mod (p-1)) mod p,Power(m,e mod (q-1)) mod q],[p,q]):
```

Example 2.16 We perform an experiment to test whether the running times of the two exponentiation algorithms—with and without the CRT—in Maple behave as expected. For this we will use two 1024-bit primes and an exponent chosen as follows:

```
> p := nextprime(2^1023):
  q := prevprime(2^1024):
  n := p*q:
  e := rand(1..(p-1)*(q-1))():
  e1:= e mod (p-1):
  e2:= e mod (q-1):
```

We remark that the primes `p` and `q` above are decidedly non-random and are not suitable for cryptographic use but they are OK for the purposes of this experiment. The next function performs `k` modular exponentiations (where `k` is a positive integer) and outputs the CPU time spent. Note that the bases `m` are chosen by means of Maple's default pseudo-random algorithm[7] in the range `2..n-2`. In fact, we do not seed this generator (this would be accomplished, for example, by the use of `randomize()`) so that, after calling `restart`, the same sequence of bases is always obtained. This way, these bases are far from random but we can use the same ones to compare the speed of the two methods.

```
> ChineseTimeTest := proc(k)
  local i, m, x, t;
  t := time();
  for i to k do
    m := (rand(2 .. n-2))();
    x := chrem([Power(m, e1) mod p, Power(m, e2) mod q], [p, q]);
  end do;
  time()-t
  end proc:
```

The next function is similar to the preceding one and records the time spent by Maple's modular exponentiation algorithm without using the CRT. As mentioned before, after a `restart` and the recomputation of the primes and the exponent, the same numbers will be used in both tests.

```
> TimeTest := proc(k)
  local i, m, x, t;
  t := time();
  for i to k do
    m := (rand(2 .. n-2))();
    x := Power(m, e) mod n;
  end do;
  time()-t
  end proc:
```

[7] Pseudo-random number generators and, in particular, those used by Maple, are discussed in Chap. 3.

Now we measure the time taken by `200` iterations of the test by the Chinese method:

```
> ChineseTimeTest(200);
                                  2.000
```

Next, after restarting Maple's kernel and recomputing all the parameters, we do the timing for the ordinary exponentiation:

```
> TimeTest(200);
                                  6.922
```

We see that the method that uses the CRT was, for these numbers, almost 3.5 times faster than the ordinary method. A quick check that both methods give the same result when these parameters are used is the following:

```
> a := rand(2..n-1)():
  evalb(Power(a,e) mod n = chrem([Power(a,e1) mod p,Power(a,e2) mod q],[p,q]));
                                  true
```

Exercise 2.22 Write a Maple procedure that extends the above method to any positive modulus whose prime factorization is known. The procedure should compute the exponentiation modulo each prime power dividing the modulus and then use the CRT to compute the final result.

2.7.3 Finding Generators in $\mathbb{Z}_p^*$

The binary exponentiation method, together with Proposition 2.6, gives an efficient algorithm to test whether a given integer, whose prime factors are known, is the order of an element of a group. In particular, this can be applied to find a generator of a cyclic group of order n (given the factorization of n). One of the more important classes of cyclic groups is given by the following proposition whose proof will be given later on, in Corollary 2.6:

Proposition 2.9 *If p is prime then the group $\mathbb{Z}_p^*$ is cyclic of order $p-1$.*

When the group $\mathbb{Z}_n^*$ is cyclic, a generator of this group is also called a *primitive root* modulo n (more generally, any integer a such that $\gcd(a,n)=1$ and $a \bmod n$ is a generator of $\mathbb{Z}_n^*$ is also called a primitive root modulo n). We will now consider the problem of finding a primitive root modulo a prime.

Example 2.17 Let us consider the following prime:

```
> nextprime(1000);
                                  1009
```

If $p = 1009$ then $p-1$ has the following factorization:

```
> ifactor(1008);
                              (2)^4 (3)^2 (7)
```

Using this information we try to find a generator of $\mathbb{Z}^*_{1009}$, i.e., an element of $\mathbb{Z}^*_{1009}$ of order 1008. By Proposition 2.6 we have that, since the quotients of 1008 by its three prime factors are 504, 336 and 144, we need to find an element g in the group such that $g^{504} \neq 1$, $g^{336} \neq 1$ and $g^{144} \neq 1$. We start by trying the element $2 \in \mathbb{Z}^*_{1009}$:

```
> 2&^504 mod 1009;
                              1
```

This already tells us that 2 is not a primitive root modulo 1009. The same happens with the successive values 3, 5, …(some values such as 4 need not be tested because once we know that 2 is not a generator we also know that $4 = 2^2$ is not a generator either) until 11 for which we find:

```
> 11&^504 mod 1009;
                           1008
> 11&^336 mod 1009;
                            374
> 11&^144 mod 1009;
                            935
```

Thus we see that 11 satisfies the conditions of Proposition 2.6 and hence is a primitive root modulo 1009.

The preceding example suggests the following algorithm to find a generator of $\mathbb{Z}^*_p$ assuming that the prime factors of $p-1$ are known:

- Choose an element $x \in \mathbb{Z}^*_p$ at random and check whether it satisfies $x^{(p-1)/q} \neq 1$ for each prime factor q of $p-1$.
- If for some prime factor q this does not hold, discard x and choose a new random element, repeating the process until a generator is found.

This method gives a probabilistic algorithm which runs in expected polynomial time on $\text{len}(p)$. The algorithm performs a sequence of Bernoulli trials (choosing elements of $\mathbb{Z}^*_p$ at random) with success probability $\phi(p-1)/(p-1)$ (since, by Theorem 2.16, there are $\phi(p-1)$ generators out of $p-1$ elements in the group) and the expected value of the random variable that estimates the number of trials for the first success is $(p-1)/\phi(p-1)$ by Proposition 2.2. Since checking whether a candidate element is a primitive root requires one modular exponentiation for each prime factor of $p-1$ and the number of these prime factors is clearly $O(\text{len}(p))$, we see that each trial requires polynomial time on $\text{len}(p)$ and hence to show that the generator-finding algorithm runs in expected polynomial time it suffices to see that the expected number of trials is also polynomial in $\text{len}(p)$. This follows from a theorem from analytic number theory due to Rosser and Schoenfeld (see, e.g., [7, Theorem 8.8.7] or also [60, Exercise 4.1]) which shows that, for $n \geq 3$, $n/\phi(n) < e^{\gamma}(\ln\ln n) + 3/(\ln\ln n)$, where γ is Euler's constant (cf. [7, p. 26]). Thus the expected number of trials is $(p-1)/\phi(p-1) = O(\ln\ln p)$.

Remarks 2.3

1. The algorithm just described requires knowledge of the prime factorization of $p-1$. If this factorization is not known, then we do not have an efficient algorithm to find a primitive root modulo p.
2. Note also that the previous algorithm does not make specific use of the fact that the cyclic group is $\mathbb{Z}_p^*$ and works also in expected polynomial time on any cyclic group of order n where the group operation can be efficiently computed.
3. The previous algorithm is probabilistic but, so far, no deterministic polynomial-time algorithm is known to find a primitive root modulo p. Such an algorithm exists if we assume the Extended Riemann Hypothesis (ERH), a widely believed but unproven conjecture that we describe in Chap. 6. If the ERH holds then, by a result of Shoup, the least primitive root modulo p is $O(\ln^6 p)$, and hence it suffices to check the least elements of $\mathbb{Z}_p^*$ in order until this primitive root is found. See [60, Research Problem 2.39] for a discussion and references.
4. There is a special case where finding a generator of a group of order n is extremely easy, namely, when the group order n is prime (which is actually the case for some groups of cryptographic interest such as elliptic curve groups of prime order). Then it follows from Corollary 2.1 that any element of G distinct from 1 is a generator.

Exercise 2.23 Write a Maple procedure that computes the smallest primitive root modulo a prime $p > 2$. Use this procedure to gather some statistical data about the size of this primitive root and examine these data in relation to a conjecture (mentioned in [60, Research Problem 2.39]) that the least primitive root is $O((\ln p)(\ln\ln p))$.

There is a variant of the preceding algorithm which is given in [180, 11.1], where it is shown that it runs in expected time $O((\text{len}(p))^4)$. Suppose that $p-1 := \prod_{i=1}^{r} q_i^{e_i}$. The algorithm proceeds as before but, instead of looking directly for a generator of $\mathbb{Z}_p^*$, it proceeds to find elements (which are chosen at random) of order $q_i^{e_i}$ for each $i = 1, \ldots, r$. Once these r elements are found, their product is a generator of $\mathbb{Z}_p^*$.

This algorithm works, more generally, for any finite cyclic group, and may be summarized as follows:

Algorithm 2.5. Computation of a generator of a finite cyclic group.

Input: A finite cyclic group G of order n and the prime factorization $n = \prod_{i=1}^{r} p_i^{e_i}$.
Output: A generator g of G.

1. `Choose`, for each $i \in \{1, \ldots, r\}$, elements in G at random until finding $x_i \in G$ such that $x_i^{n/p_i} \neq 1$.
2. `Set`, for each $i \in \{1, \ldots, r\}$, $y_i := x_i^{n/p_i^{e_i}}$.
3. `return` $g := \prod_{i=1}^{r} y_i$.

In order to see that the element g returned by this algorithm is indeed a generator of G, we can argue as follows. Consider an element $x_i \in G$ obtained in the first step of the algorithm and let n_i be the order of x_i. By Corollary 2.1, n_i divides n and, since $x_i^{n/p_i} \neq 1$, we see from Proposition 2.4 that n_i is a multiple of $p_i^{e_i}$. Now let $y_i = x_i^{n/p_i^{e_i}}$. By Proposition 2.5 the order of this element is equal to $\frac{n_i}{\gcd(n_i, n/p_i^{e_i})}$. As we have just seen, n_i is a multiple of $p_i^{e_i}$ and the cofactor of the latter is removed when dividing by the gcd, so the order of y_i is exactly $p_i^{e_i}$. Finally, since the integers $p_i^{e_i}$ are pairwise relatively prime, the order of the product g of the y_i is equal to $\prod_{i=1}^{r} p_i^{e_i} = n$ (and hence g is a generator of G) as a consequence of the following exercise:

Exercise 2.24 Let $(G, \cdot)$ be an abelian group and $x_1, x_2 \in G$ elements whose orders o_1, o_2 satisfy $\gcd(o_1, o_2) = 1$. Show that the order of $x_1 x_2$ is the product $o_1 o_2$. (Hint: Prove that the subgroup $\langle x_1 \rangle \cap \langle x_2 \rangle \subseteq G$ is the identity subgroup by checking that its order must divide the orders of x_1 and x_2. If $x_1 = x_2 = 1$ then there is nothing to prove, otherwise assume that $1 < d < o_1 o_2$ is the order of $x_1 x_2$ and show that $x_1^d, x_2^d \in \langle x_1 \rangle \cap \langle x_2 \rangle = 1$. Deduce that then d divides both o_1 and o_2, giving a contradiction.)

An implementation of the generator-finding Algorithm 2.5 in Maple, for the case of the cyclic group $\mathbb{Z}_p^*$, where p is prime, is given in the function `findgenerator` below. The first input parameter is for the prime p and there is an optional parameter `factors`, which is used for the factors of $p-1$ in the format $[[p_1, e_1], [p_2, e_2], \ldots, [p_r, e_r]]$ where the p_i are the prime divisors of $p-1$ and the e_i the corresponding exponents. The default value is precisely this list which, if not passed to the function, is computed by Maple by means of the built-in function `ifactors` applied to $p-1$.

The option of providing the factorization of $p-1$ as input is given because this will allow Maple to compute a primitive root modulo p if this factorization is known, even in cases when `ifactors` is not able to compute it. The output is a generator of $\mathbb{Z}_p^*$.

```
> findgenerator := proc(p, factors := ifactors(p-1)[2])
  local n, l, i, found, e, x, g;
  if `not`(isprime(p)) then
    error "%1 is not prime", p
  end if;
  n := nops(factors);
  RandomTools:-MersenneTwister:-SetState();
  l := [];
  for i to n do
    found := false;
    e := (p-1)/factors[i][1];
    while not found do
      x := RandomTools:-MersenneTwister:-GenerateInteger(':-range' = 2 .. p-2);
      found := evalb(x&^e mod p <> 1)
    end do;
    e := (p-1)/(factors[i][1])^(factors[i][2]);
    l := [op(l), x&^e mod p]
  end do;
  g := 1;
  for i to n do
    g := (g*l[i]) mod p
```

```
end do;
g
end proc:
```

Example 2.18 We use `findgenerator` to compute a primitive root modulo a 1024-bit prime. Let

```
> q1 := 561378849237823437209639626695358421330239881486344495906838845417463221 45\
  87113127165567687732559098437668155861260540297330848898848261719781963459449179:

  q2 := 536764790187229183471215219686142391714385392956660627307022842352548950 63\
  28707875866844215864177767975538660113907335828570679570907425483448984476864331:
```

`q1` and `q2` are (most likely) primes as can be seen by applying Maple's function `isprime` which will be discussed later; if this function returns `true` then the tested number is, with high probability, prime. Also, we check by means of the binary logarithm that both numbers have 511 bits:

```
> isprime~([q1, q2]);
  ilog2~([q1, q2]);
                                [true, true]
                                [510, 510]
```

Now, we set `p := 4*q1*q2+1` and we check that this number—which we do not print—is a 1024-bit prime:

```
> p := 4*q1*q2+1:
  isprime(p);
  ilog2(p);
                                      true
                                      1023
```

Next we compute a primitive root modulo `p`, i.e., a generator of the cyclic group $\mathbb{Z}_p^*$:

```
> findgenerator(p, [[2, 2], [q1, 1], [q2, 1]])

507158581783015838745660768924712556263357818335236570888284120213718059075471704 2\
  56887456925344543886456402017227468479254316677005127839147120046714053127092723\
  78225388310351970876146651714842638782711686064194343918066971518084609972932972\
  01647349636437715506447502018143966841099854369119418199173984027 2
```

Note that a generator of $\mathbb{Z}_p^*$ was easily found because we knew the factorization $p - 1 = 4 \cdot q1 \cdot q2$ but we would not be able to find such a generator given only p. Indeed, Maple has a function, `numtheory:-primroot`, that computes the first primitive root modulo an integer $n > 1$ such that $\mathbb{Z}_n^*$ is cyclic but this function is unable to compute a primitive root modulo the prime p above within reasonable time. The reason is that Maple does not know the factorization of $p - 1$ and so it tries to obtain it but the prime factors of $p - 1$ are too large.

Exercise 2.25 Write a Maple program that computes a prime p and the factorization of $p - 1$, given the sizes of the prime factors of $(p - 1)/2$, by pseudo-randomly choosing these factors.

Even if n is not prime, it may be the case that the group $\mathbb{Z}_n^*$ is cyclic and then, as already mentioned, a *primitive root* modulo n is a generator of $\mathbb{Z}_n^*$, i.e., an element

of order $\phi(n)$ in this group. But it may also happen that there are no primitive roots modulo n. For example:

Exercise 2.26 Prove that there are no primitive roots modulo 12 by showing that the three nonidentity elements of $\mathbb{Z}_{12}^*$ have order 2.

In fact, $n = 8$ is the smallest value for which there are no primitive roots, for it can be shown that an integer n has a primitive root if and only if $n = 2, 4, p^t, 2p^t$, where p is an odd prime. The existence of primitive roots when $n = p$ is prime will be proved in Corollary 2.6 as a consequence of a more general result about finite fields. Based on this result, we are going to show that, in fact, there are primitive roots modulo any odd prime power. We start by dealing with a squared prime, and we leave the proof of this case as an exercise:

Exercise 2.27 Show that, if p is an odd prime, then $\mathbb{Z}_{p^2}^*$ is cyclic, i.e., there exists a primitive root modulo p^2. (Hint: Let g be a primitive root modulo p. Then $g + p$ is also a primitive root modulo p and, if g is not a primitive root modulo p^2, i.e., if $g^{p-1} \equiv 1 \pmod{p^2}$, then $(g + p)^{p-1} \not\equiv 1 \pmod{p^2}$, so that $g + p$ is a primitive root modulo p^2.)

Next we show that, for any odd prime p, there exists a primitive root modulo p^t for each positive integer t.

Theorem 2.19 *Let p be an odd prime and t a positive integer. Then $\mathbb{Z}_{p^t}^*$ is a cyclic group.*

Proof By the preceding exercise we know that the result is true for $t = 2$ and that, in fact, there exists a generator g of $\mathbb{Z}_{p^2}$ such that $g \bmod p$ is also a generator of $\mathbb{Z}_p$. We claim that $g \in \mathbb{Z}_{p^t}^*$ is a generator for all $t \geq 2$. Since g is a generator of $\mathbb{Z}_{p^2}$, we know that $g^{p-1} \not\equiv 1 \pmod{p^2}$ and we will now show that $g^{p^{t-2}(p-1)} \not\equiv 1 \pmod{p^t}$. We know that this is true for $t = 2$ and, by induction, we assume that $t \geq 3$ and that

$$g^{p^{t-3}(p-1)} \not\equiv 1 \pmod{p^{t-1}}.$$

Since p does not divide g, p^{t-2} does not divide g either and so, by Euler's theorem we have that

$$g^{p^{t-3}(p-1)} \equiv g^{\phi(p^{t-2})} \equiv 1 \pmod{p^{t-2}}.$$

Thus there exists $r \in \mathbb{Z}$ such that $g^{p^{t-3}(p-1)} = 1 + rp^{t-2}$, where p does not divide r because of the induction hypothesis. Now, we raise the terms of this equality to the pth power and, using the binomial expansion for $g^{p^{t-2}(p-1)} = (1 + rp^{t-2})^p$, we obtain:

$$g^{p^{t-2}(p-1)} = 1 + prp^{t-2} + \binom{p}{2}(rp^{t-2})^2 + \cdots + (rp^{t-2})^p \equiv 1 + rp^{t-1} \pmod{p^t}.$$

Since p does not divide r we see that indeed $g^{p^{t-2}(p-1)} \not\equiv 1 \pmod{p^t}$. Now, to prove that g is a primitive root modulo p^t, let us call n its order in $\mathbb{Z}^*_{p^t}$ and observe that, by Corollary 2.1, $n|\phi(p^t) = p^{t-1}(p-1)$. On the other hand, we have that $g^n \equiv 1 \pmod{p^t}$ and hence that $g^n \equiv 1 \pmod{p}$. Since g is a primitive root modulo p (i.e., $g \bmod p \in \mathbb{Z}^*_p$ is a generator), its order in $\mathbb{Z}^*_p$ is equal to $p-1$ and divides n. Thus $(p-1)|n|p^{t-1}(p-1)$ and hence $n = p^u(p-1)$ with $u \leq t-1$. Then, if g is not a generator of $\mathbb{Z}^*_{p^t}$, its order must be a proper divisor of $|\mathbb{Z}^*_{p^t}| = p^{t-1}(p-1)$ and hence a divisor of $p^{t-2}(p-1)$. Therefore in this case we have that $u \leq t-2$ and $g^{p^{t-2}(p-1)} = (g^n)^{p^{t-2-u}} \equiv 1 \pmod{p^t}$, which is a contradiction and completes the proof. □

Exercise 2.28

(i) Modify the function `findgenerator` in order to make it able to find a primitive root modulo any integer $n \geq 2$ such that $\mathbb{Z}^*_n$ is cyclic. Use the format `proc(n,o:=numtheory:-phi(n), factors:=ifactors(o)[2])` for the input parameter declaration and make sure that the pseudo-randomly chosen elements x belong to $\mathbb{Z}^*_n$ by checking that `igcd(x, n) = 1`. The rest of the function code should be practically the same as that of `findgenerator`, just dropping the primality check and replacing `p - 1` by `o` throughout.

(ii) Use the new function to compute a primitive root modulo $2q^3$, where q is the prime $q = 3944809387454309923$. Test the result obtained by means of Maple's function `numtheory:-primroot`, by checking that if `g` is the primitive root obtained, then:
`> evalb(numtheory:-primroot(g - 1, 2 * q^3) = g);`
gives `true` as output (meaning that g is the smallest primitive root greater than $g-1$).

(iii) Define a Maple function, based on the previous one, such that the output is the smallest primitive root modulo n which is greater than a given positive integer $t < n$ (if such a primitive root exists). Compare the output of this function with that of Maple's `numtheory:-primroot`.

2.8 Finite Fields

Finite fields play an important role in cryptography and in this section we show how to construct them. We start with a small example.

2.8.1 A Field of 4 Elements

We have seen that $\mathbb{Z}_n$ is a field if and only if n is prime. This guarantees, for each prime p, the existence of a finite field of p elements (the number of elements of a

finite field is also called the *order* of the field). This result, however, leaves open the question whether there are other finite fields. Although $\mathbb{Z}_4$ is not a field, one can easily check that if we consider the set $\{0, 1, 2, 3\}$ with addition and multiplication defined by the following tables:

$\oplus$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

then we have a field of order 4 (note that we have not used the notation $\mathbb{Z}_4$ because we reserve it for the ring of residue classes modulo 4, which, as mentioned, is not a field). Thus we see that, in addition to the fields $\mathbb{Z}_p$ with p prime (the finite *prime fields*) there are other fields. But if we attempt in a similar way to find a field on the set $\{0, 1, 2, 3, 4, 5\}$ we fail: this set cannot be given a field structure, or, in other words, there are no fields with exactly 6 elements. With a little patience one might carry out this experiment but it is really not necessary because, as the next exercise shows, there are no fields of order 6 (for more details one may consult an introductory algebra text such as [45], [48] or, more specifically, a textbook on finite fields such as [131]).

Exercise 2.29

(i) Show that if $\mathbb{F}$ is a finite field, then the least positive integer m such that $\underbrace{1+1+\cdots+1}_{m \text{ times}} = 0$ is a prime number p (called the *characteristic* of $\mathbb{F}$).

(ii) Show that the set of the elements of a finite field $\mathbb{F}$ that can be obtained by adding 1 a finite number of times is a field with the operations induced by those of $\mathbb{F}$ (i.e., a subfield). Show that this subfield is isomorphic to $\mathbb{Z}_p$, where p is the characteristic of $\mathbb{F}$, so that it is a prime field called the prime subfield of $\mathbb{F}$.

(iii) Show that $\mathbb{F}$ has a natural structure of a vector space over its prime subfield so that if the dimension of this vector space is n, then $\mathbb{F}$ is isomorphic to $\mathbb{Z}_p^n$ as vector spaces (similarly to the way each finite-dimensional real vector space is isomorphic to some $\mathbb{R}^n$). Deduce that $|\mathbb{F}| = p^n$, where p is the characteristic of $\mathbb{F}$, and hence that the order of a finite field is always a prime power.

2.8.1.1. The `BitXor` Operation

The field of order 4 we have just constructed has characteristic 2 and, as we shall soon see, addition in these fields is given by the bitwise Xor operation, which we are going to implement in a Maple function called `BitXor` and plays an important role in several cryptographic algorithms such as the one-time pad (Chap. 3) and the Advanced Encryption Standard (Chap. 4). To give a Maple implementation we start

by considering bit lists, for which the operation is simply the sum modulo 2 of the corresponding bits:

```
> BitXorList := proc()
    add(_passed[i], i = 1 .. _npassed) mod 2
  end proc:
```

Next we are going to implement the `BitXor` operation for integers in the 0..255 range (in Maple v12 and above this operation is also performed by the built-in function `Bits:-Xor`). To make the operation faster, we will use a lookup table constructed with the help of the preceding function and the tables `bitstobytetable` and `bytetobitstable` given in Appendix A.

```
> bitXortable := Array(0 .. 255, 0 .. 255, (i, j) ->
              bitstobytetable[BitXorList(bytetobitstable[i], bytetobitstable[j])]):
```

Then the `BitXor` operation with two integers in the 0..255 range as arguments is the following:

```
> BitXor := (a, b) -> bitXortable[a, b]:
```

The addition operation defined on the set {0, 1, 2, 3} by the table given above is just bitwise Xor as can easily be checked; this is the reason why we used the $\oplus$ sign to denote this operation although generically we denote the addition in a finite field by the symbol '$+$' and the multiplication by '$\cdot$'.

Example 2.19 We use the previously defined Maple function `BitXor` to check that the addition table of the 4-element field above corresponds to the bitwise Xor operation. In the following Maple function, we define the table of an operation as a matrix where the (i, j)th entry contains the result of operating the ith element with the jth element (where the elements are given in a list):

```
> tabl := (lis, operation) -> Matrix(nops(lis), (i,j) -> operation(lis[i], lis[j])):
```

Next, we compute the table corresponding to `BitXor` and we see that it agrees with the one given above:

```
> tabl([0, 1, 2, 3], BitXor);
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

2.8.1.2 The Field Multiplication

It is not so easy to see where the multiplication table of the 4-element field comes from, but it is a particular case of a more general construction that will allow us to construct finite fields of p^n elements, for any prime power p^n. Before giving this general construction, let us use Maple to compute the multiplication table above by means of simple polynomial operations. Recall the definition of a polynomial:

Definition 2.24 Let R be a commutative ring with identity $1 \neq 0$. A *polynomial* in one variable over R, is a formal expression

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where n is a non-negative integer, x is the *variable* (also called an *indeterminate*) and the $a_i \in R$, $i = 1, \ldots, n$, are the *coefficients*. The set of all the polynomials in the variable x over R is denoted by $R[x]$.

We are going to identify the set $\{0, 1, 2, 3\}$ with the set $\{0, 1, x, x+1\}$ of polynomials of $\mathbb{Z}_2[x]$, with the identification done by means of the bijection that maps the latter to the former by just substituting $x = 2$:

```
> subs(x = 2, [0, 1, x, x+1]);
                                [0, 1, 2, 3]
```

Then we define a multiplication in this set of polynomials as follows:

```
> mult4 := (f,g) -> Rem(f*g, x^2+x+1, x) mod 2:
```

The preceding operation takes two polynomials in the variable x, multiplies them, and computes the remainder of dividing their product by the polynomial $x^2 + x + 1$ (see Sect. 2.8.2 for the definition of these operations), where all the polynomials are considered with coefficients in $\mathbb{Z}_2$. Now we use the function `tabl` above to compute the table of this operation in matrix form:

```
> tabl([0, 1, x, x+1], mult4);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & x & x+1 \\ 0 & x & x+1 & 1 \\ 0 & x+1 & 1 & x \end{bmatrix}$$

If we want to see this multiplication viewed as defined on the set $\{0, 1, 2, 3\}$, we substitute $x = 2$ into the preceding matrix:

```
> subs(x = 2, %);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 3 & 1 \\ 0 & 3 & 1 & 2 \end{bmatrix}$$

We see that the operation defined with the help of the polynomials is indeed the same as the multiplication we had previously defined on $\{0, 1, 2, 3\}$.

Exercise 2.30 Use Maple to define this multiplication explicitly on the set $\{0, 1, 2, 3\}$ and build the corresponding table using `tabl`. In order to do it, define the map $\{0, 1, 2, 3\} \to \{0, 1, x, x+1\}$ whose inverse is given by `f -> subs(x = 2, f)` (this map can be defined more generally for any positive integer: the coefficients of the polynomial corresponding to one of these integers are just its binary digits).

Then use these maps to define the multiplication as a function that assigns to a pair of elements of {0, 1, 2, 3} another element of this set (their product). Use `tabl` to compute the table of this multiplication and verify that it is the one given above.

The multiplication `mult4` defined above looks very similar to the multiplication in $\mathbb{Z}_n$, except that now we multiply polynomials and take the remainder of dividing by x^2+x+1 which plays here the role of the modulus n. The similarity goes further because the four polynomials $0, 1, x, x+1$ are the possible remainders of dividing a polynomial by x^2+x+1 just like the elements of $\mathbb{Z}_n$ are the possible remainders of integer division by n (we are assuming that all these polynomials have coefficients in $\mathbb{Z}_2$). We have seen before that the order of a finite field is always a prime power and, by generalizing this construction we are going to show that, for each prime power p^n there is a field of order p^n (which, moreover, is unique up to isomorphism).

2.8.2 The Polynomial Ring

Notations and terminology. We will adopt the standard terminology and notational conventions for polynomials. For example, if there is no ambiguity about the name of the variable, we will write f instead of $f(x)$ to denote a polynomial. The zero polynomial is the polynomial whose coefficients are all equal to 0 and is denoted simply by 0. Moreover, we follow the convention that a term of the form $a_i x^i$ with $a_i = 0$ need not be written down, so a generic nonzero polynomial $f = \sum_{i=0}^{n} a_i x^i \in R[x]$ can always be written as a sum of polynomials—called *monomials*—of the form $a_i x^i$ with $a_i \neq 0$. If $f \neq 0$, then $f = \sum_{i=0}^{n} a_i x^i$ with $a_n \neq 0$ and a_n is called the *leading coefficient* of f and n the *degree* of f, denoted $n = \deg(f)$. If $f = 0$ then, by convention, we write $\deg(f) = -\infty$ (and we also agree to the conventions that $-\infty < n$, $-\infty + -\infty = -\infty$ and $-\infty + n = -\infty$, for each integer n). a_0 is the *constant term* of f and if $f = a_0$ or, equivalently, if $\deg(f) \leq 0$, f is called a *constant polynomial*. The polynomials whose leading coefficient is 1 are called *monic polynomials*. We see that f is a monomial precisely when all the coefficients of f, except the leading coefficient, are equal to 0.

Let R be a commutative ring with identity and $f, g \in R[x]$. We can write $f = \sum_{i=0}^{n} a_i x^i$, $g = \sum_{i=0}^{n} b_i x^i$ and define their sum:

$$f + g = \sum_{i=0}^{n} (a_i + b_i) x^i .$$

The product of $f = \sum_{i=0}^{n} a_i x^i$ and $g = \sum_{i=0}^{m} b_i x^i$ is the polynomial:

$$fg = \sum_{k=0}^{n+m} c_k x^k, \text{ where } c_k = \sum_{i=0}^{k} a_i b_{k-i}, \text{ for } 0 \leq k \leq n+m.$$

We see that the sum of two polynomials is computed just by adding in R the coefficients corresponding to the monomials of the same degree. Also, if polynomials are viewed as a sum of monomials, then the product is obtained by distributivity as the sum of all the products of monomials, where two monomials are multiplied by adding their degrees and multiplying their coefficients. It is straightforward to see that $R[x]$ with the operations just defined is a ring called the *polynomial ring* in the variable x over R. Note that the zero element of $R[x]$ is precisely the 0 polynomial, i.e., the polynomial all of whose coefficients are 0. Similarly the identity of this ring is denoted by 1 and is the polynomial whose only nonzero coefficient is $a_0 = 1$.

Exercise 2.31 Show that polynomials of $R[x]$ can be defined as infinite sequences $\{a_i\}_{i\in\mathbb{N}}$ of elements of R such that all but a finite number of the a_i are zero (as usual, $\mathbb{N}$ denotes the set of natural numbers, i.e., the set of non-negative integers). Show how to define an addition and a multiplication in the set of these sequences which make it a ring isomorphic to $R[x]$.

In the following, the coefficient ring R will always be a field k and, in fact, it will usually be a finite prime field $k = \mathbb{Z}_p$. For polynomial rings over fields, a straightforward comparison of the leading coefficients shows that:

Proposition 2.10 *Let $f, g \in k[x]$, where k is a field. Then we have:*

(i) $\deg(f + g) \leq \max\{\deg(f), \deg(g)\}$.
(ii) $\deg(fg) = \deg(f) + \deg(g)$.

Proof (i) is clear. For (ii) observe first that if any of the involved polynomials is equal to 0 then the result holds by our previous convention regarding sums including the term $-\infty$. Otherwise, let a_n and b_m be the leading coefficients of f and g. Then, since $k[x]$ has no zero divisors we have that $a_n b_m \neq 0$ and this is both the coefficient of the monomial of degree $n + m$ and the leading coefficient of fg, so that fg has degree $n + m$. □

As a consequence of part (ii) of the above proposition we see that the ring $k[x]$ is an integral domain. This is one property that $k[x]$ shares with $\mathbb{Z}$ and in fact, from the point of view of divisibility, the ring $k[x]$ behaves in a way very similar to $\mathbb{Z}$. From a more general point of view whose details we will omit, both rings are *Euclidean domains* which implies the existence of division with remainder and that the Euclidean algorithm applies. We introduce some terminology entirely similar to that used for $\mathbb{Z}$. If $f, g \in k[x]$ then we say that g divides f (denoted by $g|f$) if there exists a polynomial $q \in k[x]$ such that $f = gq$. In this case, we also say that g is a divisor of f and that f is a multiple of g. Note that the units (i.e., the invertible elements) of $k[x]$ are the divisors of the polynomial 1, i.e., all nonzero constant polynomials.

Theorem 2.20 *Let $f, g \in k[x]$, $g \neq 0$. Then there exist uniquely determined polynomials $q, r \in k[x]$ such that:*

$$f = gq + r \quad \text{and} \quad \deg(r) < \deg(g).$$

Proof The case $f = 0$ is trivial, just set $q = r = 0$. Thus we will assume in the sequel that $f \neq 0$. Now, if $\deg(f) < \deg(g)$, then we set $q = 0$ and $r = f$. Therefore we may also assume that $\deg(g) \leq \deg(f)$. The proof of the existence of q and r proceeds by induction on the degree of f. If $\deg(f) = 0$ then $\deg(g) = 0$ too, so that both f and g are nonzero constant polynomials and hence units in $k[x]$, and we set $q = fg^{-1}$ and $r = 0$.

Suppose now that $\deg(f) = n > 0$, where $n \geq m = \deg(g)$, and let

$$f = \sum_{i=0}^{n} a_i x^i, \text{ and } g = \sum_{i=0}^{m} b_i x^i.$$

Then we set $f_1 = f - a_n b_m^{-1} x^{n-m} g$ and since $\deg(f_1) < \deg(f)$, we may use the induction hypothesis to conclude that there are polynomials $q_1, r \in k[x]$ such that $f_1 = q_1 g + r$, with $\deg(r) < \deg(g)$. Then, replacing this value of f_1 in the previous equation we obtain:

$$f = (a_n b_m^{-1} x^{n-m} + q_1)g + r,$$

so that $q = a_n b_m^{-1} x^{n-m} + q_1$ and r satisfy the assertion of the theorem.

To prove uniqueness, let $f = gq + r = gq' + r'$, with $\max\{\deg(r), \deg(r')\} < \deg(g)$. Then $g(q - q') = r' - r$ and so $\deg(g) + \deg(q - q') = \deg(r - r') \leq \max\{\deg(r), \deg(r')\} < \deg(g)$ (by Proposition 2.10 and the hypothesis). This inequality can only hold if $\deg(q - q') = -\infty$, i.e., we have that $q = q'$. Thus $r' - r = g(q - q') = g0 = 0$ and we also have $r' = r$. □

The preceding proof gives an algorithm to perform the division of polynomials:

Algorithm 2.6. Polynomial division.

Input: Polynomials $f, g \in k[x]$, such that $g \neq 0$.
Output: $q, r \in k[x]$ such that $f = gq + r$ and $\deg(r) < \deg(g)$.

1. `Set` $r := f$ and $q := 0$.
2. `Compute` the quotient and the remainder:
   `while` $\deg(r) \geq \deg(g)$ `do`
   $a :=$ leading coefficient of r
   $b :=$ leading coefficient of g
   $h := ab^{-1}x^{\deg(r)-\deg(g)}$
   $r := r - hg$
   $q := q + h$
   `end do`
3. `return`: q, r.

The remainder of dividing f by g is denoted by f mod g. In Maple, division of polynomials is implemented in the functions `Rem` (we already used this function when we defined the multiplication `mult4`) and `Quo`. These functions, when used

in conjunction with `mod` (or with `modp1`) implement division in the polynomial ring $\mathbb{Z}_p[x]$, where p is a prime number.

Example 2.20 Suppose that we want to compute the quotient and the remainder of the division of $x^{13}+x^{10}+x^9+x^7+x^4+x^2+1 \in \mathbb{Z}_2[x]$ by $x^8+x^4+x^3+x+1 \in \mathbb{Z}_2[x]$:

```
> f := x^13 + x^10 + x^9 + x^7 + x^4 + x^2 + 1:
  g := x^8 + x^4 + x^3 + x + 1:
```

Then we may use `Rem` which will give the remainder as output and, if provided with a fourth optional argument '`q`', will assign the value of the quotient to the variable `q`, so that:

```
> Rem(f, g, x, 'q') mod 2;
  q;
                                    x^7 + x
                                x^5 + x^2 + 1
```

This shows that, in this case, $r = x^7 + x$ and $q = x^5 + x^2 + 1$. Alternatively, we can use `Quo`, in which case the output of the function is the quotient and the remainder is optionally assigned to a variable name (`r` in our example) if it is passed as fourth argument to the function:

```
> Quo(f, g, x, 'r') mod 2;
  r;
                                x^5 + x^2 + 1
                                  x^7 + x
```

The divisibility theory of $k[x]$ closely mimics that of $\mathbb{Z}$ and also in this case the greatest common divisor of polynomials can be defined and computed by means of the Euclidean algorithm. If $f, g \in k[x]$ are polynomials, not both 0, then the *greatest common divisor* of f and g is the unique monic polynomial $d \in k[x]$ such that $d|f$, $d|g$, and if $h \in k[x]$ is such that $h|f$, $h|g$, then $h|d$. Moreover, d can be expressed in the form $d = sf + tg$ with $s, t \in k[x]$ (see, e.g., [131, Theorem 1.55]).

Remark 2.6 The requirement that the gcd be a monic polynomial is purely technical and its purpose is to make the gcd unique. Otherwise the gcd would be defined only up to units of $k[x]$, i.e., up to multiplication by nonzero elements of k.

We shall not go into the details of the Euclidean algorithm for polynomials because it is practically the same as the Euclidean algorithm for integers, just replacing integer division by polynomial division. Starting with f and g the analogous sequence of divisions is performed until reaching a zero remainder. Then the previous remainder, multiplied by the inverse in k of its leading coefficient (in order to make the polynomial monic) is the gcd of f and g. There is also a version of the extended Euclidean algorithm for $k[x]$, which computes, in addition to $d = \gcd(f, g)$, polynomials $s, t \in k[x]$ such that $d = sf + tg$. As in the integer case, when $d = 1$, f and g are said to be *relatively prime*.

Example 2.21 We use Maple to compute the greatest common divisor of the two polynomials of $\mathbb{Z}_2[x]$ we used in Example 2.20:

```
> Gcd(f, g) mod 2;
                                    1
```

We see that these polynomials are relatively prime. We use the extended Euclidean algorithm to compute $s, t \in k[x]$ such that $1 = sf + tg$:

```
> Gcdex(f, g, x, 's', 't') mod 2;
  s, t;
                                    1
   x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, x^11 + x^10 + x^9 + x^6 + x^4 + x^3 + x^2 + x
```

We see that, $s = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$, $t = x^{11} + x^{10} + x^9 + x^6 + x^4 + x^3 + x^2 + x$ in this case. We may check that they satisfy $1 = sf + tg$:

```
> Expand(s*f+t*g) mod 2;
                                    1
```

An element $\alpha \in k$ is said to be a *zero* of $f \in k[x]$ whenever $f(\alpha) = 0$ (where, as usual, $f(\alpha)$ denotes the element of k obtained by substituting $x = \alpha$ into the polynomial and carrying out the field operations, so that α is a root of the equation $f(x) = 0$ and we also say that α is a root of f). There is a simple test to determine whether an element of k is a zero of f in terms of divisibility:

Theorem 2.21 *Let $f \in k[x]$ and $\alpha \in k$. Then α is a zero of f if and only if $(x-\alpha)|f$.*

Proof We divide f by $x - \alpha$ and we have $f = (x - \alpha)q + r$. Then $f(\alpha) = r$ and hence $f(\alpha) = 0$ (i.e., α is a zero of f) if and only if $r = 0$ (i.e., $(x - \alpha)|f$). □

Example 2.22 Since 1 is a zero of $x^2 + 1 \in \mathbb{Z}_2[x]$ and also of $x^3 + 1 \in \mathbb{Z}_2[x]$, we see that $x + 1 \in \mathbb{Z}_2[x]$ divides both polynomials. Indeed, one easily checks that $x^2 + 1 = (x + 1)^2 \in \mathbb{Z}_2[x]$ and also that $x^3 + 1 = (x + 1)(x^2 + x + 1) \in \mathbb{Z}_2[x]$ (note that no minus signs appear here because $-1 = 1$ in $\mathbb{Z}_2$). In fact, 1 is a zero of the polynomial $x^n + 1 \in \mathbb{Z}_2[x]$ for all $n > 0$ and it is easily seen that $x^n + 1 = (x + 1)(x^{n-1} + x^{n-2} + \cdots + 1) \in \mathbb{Z}_2[x]$.

Corollary 2.4 *A nonzero polynomial $f \in k[x]$ has at most* $\deg(f)$ *zeros.*

Proof We use induction on $n = \deg(f)$. If $n = 0$ then f is a nonzero element of k and hence f has no zeros. Suppose now that $n > 0$. If f has no zeros then the assertion is true so we may assume that f has a zero α. By Theorem 2.21 we have that $f = (x - \alpha)q$ where $\deg(q) = n - 1$. The induction hypothesis then implies that q has at most $n - 1$ zeros from which it follows that f has at most n zeros. □

If α is a zero of $f \in k[x]$, then the maximum value of t such that $f = (x - \alpha)^t g$, with $g \in k[x]$, is called the *multiplicity* of α in f and we often say that α is a *multiple zero* (or a *multiple root*) when $t > 1$. It is not difficult to see that Corollary 2.4 is also valid if one counts multiplicities, i.e., the sum of the multiplicities of the zeros of a polynomial cannot exceed its degree.

Exercise 2.32 Consider the polynomial ring $\mathbb{Z}_2[x]$ and give examples that show that a polynomial of degree 2 can have two zeros, one zero (which then will have multiplicity 2) or no zeros.

2.8.3 The Field of p^n Elements

We are now ready to construct, given a prime power p^n, a field of p^n elements. The construction will be very similar to the one used to obtain $\mathbb{Z}_p$ from the integers but, in this case, the starting point will be $\mathbb{Z}_p[x]$ instead.

So, suppose that $f \in \mathbb{Z}_p[x]$ is a polynomial with $\deg(f) = n > 0$. We define a set $\mathbb{Z}_p[x]_f = \{g \in \mathbb{Z}_p[x] | \deg(g) < \deg(f)\}$ which is the analogue of $\mathbb{Z}_q$, for $q \in \mathbb{Z}$, $q > 1$. Indeed, while the latter set consists of the possible remainders of integer division by q, the former consists of all the possible remainders of polynomial division by f. Note that $\mathbb{Z}_p[x]_f = \{a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 | a_i \in \mathbb{Z}_p\}$, so that $|\mathbb{Z}_p[x]_f| = p^n$. We define two binary operations on $\mathbb{Z}_p[x]_f$ by $g + h = (g + h) \bmod f$, $gh = (gh) \bmod f$, i.e., the operations in $\mathbb{Z}_p[x]_f$ are the same as the corresponding operations in $\mathbb{Z}_p[x]$ followed by taking the remainder of division by f in $\mathbb{Z}_p[x]$.[8] In fact, adding two polynomials of $\mathbb{Z}_p[x]_f$ gives the same result as adding them in $\mathbb{Z}_p[x]$, so that the 'mod f' is not necessary in the case of addition but we have written the operation this way to emphasize the fact that these operations are defined exactly like the operations of $\mathbb{Z}_q$: in order to add/multiply two polynomials in $\mathbb{Z}_p[x]_f$, we just add/multiply them in $\mathbb{Z}_p[x]$ and then we reduce modulo f.

It is easily seen that $\mathbb{Z}_p[x]_f$ with these two operations is a commutative ring with identity. The proof is straightforward and $\mathbb{Z}_p[x]_f$ inherits the ring properties from $\mathbb{Z}_p[x]$ similarly to the way $\mathbb{Z}_q$ inherits them from $\mathbb{Z}$. But observe that $\mathbb{Z}_p[x]_f$ need not be a field:

Example 2.23 Let $p = 2$ and $f = x^2 + 1 \in \mathbb{Z}_2[x]$. Then $\mathbb{Z}_2[x]_f$ is not a field because the element $x + 1 \in \mathbb{Z}_2[x]_f$ is a zero divisor and hence it does not have an inverse. Indeed, we see that in $\mathbb{Z}_2[x]_f$, $(x+1)(x+1) = (x^2+1) \bmod (x^2 + 1) = 0$.

Remarks 2.4

1. The ring $\mathbb{Z}_p[x]_f$ is constructed from $\mathbb{Z}_p[x]$ just as $\mathbb{Z}_q$ is constructed from $\mathbb{Z}$. This ring can also be built in a way similar to the construction of $\mathbb{Z}/n\mathbb{Z}$ from $\mathbb{Z}$, i.e., as a quotient ring whose elements are congruence classes modulo f (also called a residue class ring). These classes are defined, using the divisibility properties of $\mathbb{Z}_p[x]$, in a similar way to congruence classes modulo a positive integer in $\mathbb{Z}$: given $g, h \in \mathbb{Z}_p[x]$, we say that $g \equiv h \pmod f$ if and only if $f | g - h$. Then, if we denote the congruence class of $g \in \mathbb{Z}_p[x]$ by $[g]$, we set $\mathbb{Z}_p[x]/(f) = \{[g] | g \in \mathbb{Z}_p[x]\} = \{g + (f) | g \in \mathbb{Z}_p[x]\}$, where $(f) = \{hf | h \in \mathbb{Z}_p[x]\}$ is the

[8] We are committing here the usual abuse of notation and using the same generic symbols for different additions and different multiplications.

ideal of $\mathbb{Z}_p[x]$ generated by f and $g+(f)=\{g+k|k\in(f)\}$. The rings $\mathbb{Z}_p[x]_f$ and $\mathbb{Z}_p[x]/(f)$ are canonically isomorphic, with isomorphism given by the map $g\mapsto[g]$ from the former to the latter. Thus both rings are essentially the same and we will usually work with the former since its elements (polynomials) require less notation.

2. Note that, although we will only need to work with the rings $\mathbb{Z}_p[x]$ in order to build the fields of order p^n, all the preceding constructions and results are valid, more generally, for any polynomial ring $k[x]$, with k a field.

We need something else in order to obtain a field from this construction. Example 2.23 shows that, as in the case of $\mathbb{Z}_q$, $\mathbb{Z}_p[x]_f$ may fail to be a field due to the existence of zero divisors. In fact, the non-existence of zero divisors is, in this case, not only necessary but also sufficient for $\mathbb{Z}_p[x]_f$ to be a field. Indeed, as the following exercise shows, any finite integral domain is a field (see also [131, Theorem 1.31]).

Exercise 2.33 Show that a finite integral domain R is a field. To prove it, consider a nonzero element $a\in R$ and show that the map from R to itself given by multiplication with a is injective. Show that this map is also surjective (and hence a bijection) and use it to prove that a is an invertible element of R. (The same argument shows, more generally, that any non-zero divisor in a finite commutative ring with 1 is invertible.)

It is clear then that, for $\mathbb{Z}_p[x]_f$ to be a field, the product in $\mathbb{Z}_p[x]$ of two polynomials of $\mathbb{Z}_p[x]_f$ cannot be equal to f. Thus, as in the construction of $\mathbb{Z}_q$, we see that f must satisfy a property similar to primality. This property is the following:

Definition 2.25 A polynomial $f\in\mathbb{Z}_p[x]$ is said to be *irreducible* if $\deg(f)>0$ and f cannot be written as a product $f=gh$, where $g,h\in\mathbb{Z}_p[x]$ and $\deg(g)$, $\deg(h)>0$.

Examples 2.4

1. Constant polynomials are not irreducible but all polynomials of degree 1 are irreducible by Proposition 2.10.
2. If $\deg(f)=2$ or 3 then f is irreducible if and only if f does not have zeros in k. Indeed, using Proposition 2.10 we see that the irreducibility of f is, in this case equivalent to not having any degree 1 factor, i.e., to the non-existence of a polynomial g such that $\deg(g)=1$ and $g|f$. But, by Theorem 2.21 this is, in turn, equivalent to f not having zeros in k.
3. The only irreducible polynomial of degree 2 in $\mathbb{Z}_2[x]$ is x^2+x+1 because neither 0 nor 1 is a zero while, on the other hand, the remaining three polynomials of degree 2 (namely, x^2, x^2+x, x^2+1) all have zeros in $\mathbb{Z}_2$. Similarly, it can easily be checked that the only irreducible polynomials of degree 3 in $\mathbb{Z}_2[x]$ are x^3+x+1, x^3+x^2+1.

Now, suppose that we want to find all the irreducible polynomials of a given degree over $\mathbb{Z}_p$. If the degree is greater than 3, then we cannot use the criterion given in Example 2.4 but we can check whether a polynomial is irreducible or not by dividing it by the irreducible polynomials of smaller degree (or we can generate

all the reducible polynomials of a given degree by multiplying all polynomials of smaller degree). This method is rather inefficient when the degree is large, and there are more powerful methods to check the irreducibility of a polynomial, but we shall not go into the details as they are not needed for our purposes. We will show instead how we can compute the irreducible polynomials (of a given degree) over a prime field using Maple.

The Maple function that tests whether a polynomial over a prime field is irreducible is `Irreduc` which, when used together with the operator `mod` in the form `Irreduc(f) mod p` (where f is the polynomial to be tested and p the prime that defines the field of coefficients) returns `true` if the polynomial $f \in \mathbb{Z}_p[x]$ is irreducible and `false` otherwise.

Example 2.24 Let us check the irreducibility of a couple of polynomials of $\mathbb{Z}_2[x]$. We have:

```
> Irreduc(x^8 + x^4 + x^3 + x + 1) mod 2;
                                  true
```

while, on the other hand:

```
> Irreduc(x^8 + x^4 + x^2 + x + 1) mod 2;
                                  false
```

This shows that the first of these polynomials of $\mathbb{Z}_2[x]$ is irreducible while the second is not. We can also obtain the irreducible factors of the latter polynomial by computing:

```
> Factor(x^8 + x^4 + x^2 + x + 1) mod 2;
                    (x^4 + x^3 + x^2 + x + 1)(x^4 + x^3 + 1)
```

This also shows that not having any zeros in the coefficient field is not a sufficient condition for a polynomial to be irreducible. It is easy to check that the polynomial $x^8 + x^4 + x^2 + x + 1 \in \mathbb{Z}_2[x]$ has no zeros in $\mathbb{Z}_2$ but, as we have just seen, this polynomial is not irreducible.

Next we give a procedure to compute all the irreducible polynomials of a given degree over a prime field. In practice, one is usually interested in the monic irreducible polynomials because any irreducible polynomial has a unique associated monic irreducible polynomial which is obtained just by multiplying the original polynomial by the inverse of its leading coefficient. This is a similar situation to the one arising when we consider the primes, i.e., the "positive irreducibles" of $\mathbb{Z}$ (the negative irreducibles are obtained from the primes just as the non-monic irreducible polynomials are obtained from the monic ones: by multiplying by a unit of the ring).

The next function takes as input a prime p which defines the coefficient field, a positive integer n which specifies the degree, and an optional parameter `monic` which is `true` by default. The output is the list of all the monic irreducible polynomials of $\mathbb{Z}_p[x]$ of degree n or, if the argument `false` is passed to `monic`, the list of all irreducible polynomials.

```
> IrreduciblePolynomialsList := proc(p, n::posint, monic := true)
  local coeffs, pols, interv;
  global x;
  x := 'x';
  if monic then
    interv := [$p^n..2*p^n - 1]
  else
    interv := [$p^n..p^(n+1)-1]
  end if;
  coeffs := map(a -> convert(a, base, p), interv);
  pols := map(l -> sort(sum(l[j]*x^(j-1), j = 1 .. n+1)), coeffs);
  #in Maple v.13 and above the preceding line may be replaced by the following:
  #pols := map(l -> PolynomialTools:-FromCoefficientList(l,x), coeffs);
  select(f -> Irreduc(f) mod p, pols)
  end proc:
```

Example 2.25 Let us compute the monic irreducible polynomials of degree 4 in $\mathbb{Z}_2[x]$ (note that when $p = 2$ all nonzero polynomials are monic):

```
> IrreduciblePolynomialsList(2, 4);
              [x^4 + x + 1, x^4 + x^3 + 1, x^4 + x^3 + x^2 + x + 1]
```

Similarly, the irreducible polynomials of degree 8 in $\mathbb{Z}_2[x]$ are:

```
> IrreduciblePolynomialsList(2, 8);
[x^8 + x^4 + x^3 + x + 1, x^8 + x^4 + x^3 + x^2 + 1, x^8 + x^5 + x^3 + x + 1,
  x^8 + x^5 + x^3 + x^2 + 1, x^8 + x^5 + x^4 + x^3 + 1,
  x^8 + x^5 + x^4 + x^3 + x^2 + x + 1, x^8 + x^6 + x^3 + x^2 + 1,
  x^8 + x^5 + x^3 + x + 1, x^8 + x^6 + x^5 + x + 1, x^8 + x^6 + x^5 + x^2 + 1,
  x^8 + x^6 + x^5 + x^3 + 1, x^8 + x^6 + x^5 + x^4 + 1,
  x^8 + x^6 + x^5 + x^4 + x^2 + x + 1, x^8 + x^6 + x^5 + x^4 + x^3 + x + 1,
  x^8 + x^7 + x^2 + x + 1, x^8 + x^7 + x^3 + x + 1, x^8 + x^7 + x^3 + x^2 + 1,
  x^8 + x^7 + x^4 + x^3 + x^2 + x + 1, x^8 + x^7 + x^5 + x + 1,
  x^8 + x^7 + x^5 + x^3 + 1, x^8 + x^7 + x^5 + x^4 + 1,
  x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + 1, x^8 + x^7 + x^6 + x + 1,
  x^8 + x^7 + x^6 + x^3 + x^2 + x + 1, x^8 + x^7 + x^6 + x^4 + x^2 + x + 1,
  x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + 1, x^8 + x^7 + x^6 + x^5 + x^2 + x + 1,
  x^8 + x^7 + x^6 + x^5 + x^4 + x + 1, x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + 1,
  x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1]
```

Let us now consider an example in characteristic $\neq 2$; the monic irreducible polynomials of degree 4 over $\mathbb{Z}_3[x]$ are:

```
> IrreduciblePolynomialsList(3, 4);
[x^4 + x + 2, x^4 + 2x + 2, x^4 + x^2 + 2, x^4 + x^2 + x + 1, x^4 + x^2 + 2x + 1,
  x^4 + 2x^2 + 2, x^4 + x^3 + 2, x^4 + x^3 + 2x + 1, x^4 + x^3 + x^2 + 1,
  x^4 + x^3 + x^2 + x + 1, x^4 + x^3 + x^2 + 2x + 2, x^4 + x^3 + 2x^2 + 2x + 2,
  x^4 + 2x^3 + 2, x^4 + 2x^3 + x + 1, x^4 + 2x^3 + x^2 + 1,
  x^4 + 2x^3 + x^2 + x + 2, x^4 + 2x^3 + x^2 + 2x + 1, x^4 + 2x^3 + 2x^2 + x + 2]
```

These results may be compared with the listing in [131, Table C].

In the previous example we have considered irreducible polynomials of low degree but, as we will see in Chap. 5, an irreducible polynomial of degree 128 is used in the CMAC authentication scheme. The list of irreducible polynomials of degree 128 in $\mathbb{Z}_2[x]$ is way too long to be displayed here, but we will give a procedure to find a single irreducible polynomial of a given degree. This procedure takes as input the prime p and the degree n and returns the first irreducible polynomial of $\mathbb{Z}_p[x]$ in the enumeration implicitly defined in `IrreduciblePolynomialsList`, i.e.,

the irreducible polynomial f such that the value of $f(p)$ is the lowest among all the values $g(p)$ with g irreducible of degree n. Of course, this procedure can easily be modified to find not just one but any given number of irreducible polynomials instead (provided that the specified number is less than the total number of irreducible polynomials of the given degree). The function that finds the first irreducible polynomial is the following:

```
> IrreduciblePolynomial := proc(p, n::posint)
  local a, coeffs, f, found;
  global x;
  a := p^n;
  x := 'x';
  found := false;
  while not found do
    coeffs := convert(a, base, p);
    f := sum(coeffs[j]*x^(j-1), j = 1 .. n+1);
    found := Irreduc(f) mod p;
    a := a+1
  end do;
  sort(f)
  end proc:
```

Example 2.26 Let us find the first irreducible polynomial of degree 128 in $\mathbb{Z}_2[x]$, which is the one used in the CMAC scheme:

```
> IrreduciblePolynomial(2, 128);
                          x^128 + x^7 + x^2 + x + 1
```

Of course, using Maple it is easy to find irreducible polynomials of even larger degree.

Exercise 2.34

(i) Write a Maple procedure that finds the first ten monic irreducible polynomials in $\mathbb{Z}_p[x]$ of a given degree n (or the whole list of monic irreducible polynomials if there are fewer than ten).

(ii) Write a Maple procedure that finds the "last" irreducible polynomial in $\mathbb{Z}_p[x]$ of degree n, i.e., the irreducible polynomial f of degree n for which $f(p)$ is largest.

As can be easily guessed from our previous comments, irreducible polynomials allow us to build finite fields of order p^n:

Theorem 2.22 *Let $f \in \mathbb{Z}_p[x]$. Then the ring $\mathbb{Z}_p[x]_f$ is a field if and only if f is irreducible.*

Proof The proof is the same as the one that $\mathbb{Z}_q$ is a field if and only if q is prime, except that this time we work with polynomials of $\mathbb{Z}_p[x]$ and with the (extended) Euclidean algorithm for polynomials. We have already noted that the irreducibility of f is a necessary condition for $\mathbb{Z}_p[x]_f$ to be a field because if f is not irreducible, then any factor g of f such that $0 < \deg(g) < \deg(f)$ is a zero divisor in $\mathbb{Z}_p[x]_f$. Conversely, if f is irreducible and g a nonzero polynomial in $\mathbb{Z}_p[x]_f$, then

$\gcd(f, g) = 1$ and the extended Euclidean algorithm allows us to compute polynomials $s, t \in \mathbb{Z}_p[x]$ such that $1 = sf + tg$. Then $1 \equiv tg \pmod{f}$ and hence $t \bmod f$ is the inverse of g in $\mathbb{Z}_p[x]_f$. □

The following table summarizes the parallelism between the constructions of $\mathbb{Z}_q$ and $\mathbb{Z}_2[x]_f$:

Ring	$\mathbb{Z}$	$\mathbb{Z}_p[x]$
Element	$q \in \mathbb{Z}, q \geq 2$	$f \in \mathbb{Z}_p[x], \deg(f) = n > 0$
Quotient ring	$\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$	$\mathbb{Z}_p[x]_f = \{g \in \mathbb{Z}_p[x] \mid \deg(g) < n\}$
Field	$\mathbb{Z}_q$ field iff q prime	$\mathbb{Z}_p[x]_f$ field iff f irreducible

Remarks 2.5 Theorem 2.22 shows that for each prime p and each irreducible polynomial of degree n in $\mathbb{Z}_p[x]$ there exists a field of order p^n. It can be shown that, for each prime p and each positive integer n, there is always an irreducible polynomial of degree n in $\mathbb{Z}_p[x]$, so that there is always a field of order p^n. Moreover, every finite field has order p^n for some prime p and some integer $n > 0$ as a consequence of Exercise 2.29. Note that there may be several distinct irreducible polynomials of degree n in $\mathbb{Z}_p[x]$, and each of them gives rise to a different field $\mathbb{Z}_p[x]_f$. However, all these fields are isomorphic and this allows us to speak of *the* finite field of order p^n which is denoted by $\mathbb{F}_{p^n}$ or by $GF(p^n)$ (where the initials GF stand for "Galois Field"). We emphasize, however, that in cryptographic applications, the irreducible polynomial used to construct the field is explicitly defined in order to obtain exactly the same multiplication.

Examples 2.5

1. Let $f = x \in \mathbb{Z}_p[x]$. Then f is irreducible and $\mathbb{Z}_p[x]_f$ is a field of order p. In this case $\mathbb{Z}_p[x]_f$ consists of the polynomials of degree < 1, i.e., the constant polynomials. The polynomial operations on these constant polynomials are the same as the operations of these elements when regarded as elements of $\mathbb{Z}_p$. Thus we see that, in this case, $\mathbb{Z}_p[x]_f = \mathbb{Z}_p$. As mentioned above, often the notation $\mathbb{F}_p$ is used (instead of $\mathbb{Z}_p$) to denote this field.
2. Let $f = x^2 + x + 1 \in \mathbb{Z}_2[x]$. Then the field $\mathbb{F}_4 = \mathbb{Z}_2[x]_f$ is simply the field of 4 elements constructed at the beginning of this section and the multiplication is given by the previously defined Maple function `mult4`.
3. Let us use Maple to build the multiplication table of the finite field of 9 elements. We first find an irreducible polynomial of degree 2 in $\mathbb{Z}_3[x]$:

```
> IrreduciblePolynomial(3, 2);
                                x^2 + 1
```

The multiplication in $\mathbb{Z}_3[x]_f$, where $f = x^2 + 1$, will be given by the following function:

```
> mult9 := (f, g) -> Rem(f*g, x^2+1, x) mod 3:
```

We need to build the list of elements of $\mathbb{Z}_3[x]_f$, i.e., the list of polynomials of $\mathbb{Z}_3[x]$ of degree less than 2. The next function builds the list of polynomials of degree $< n$ in $\mathbb{Z}_p[x]$:

```
> polylist := proc(p, n)
  local coeffs;
  global x;
  x := 'x';
  coeffs := map(a -> convert(a, base, p), [$0 .. p^n-1]);
  map(l -> sort(sum(l[j]*x^(j-1), j = 1 .. nops(l))), coeffs)
  end proc:
```

The elements of $\mathbb{Z}_3[x]_f$ are then the following:

```
> polylist(3, 2);
              [0, 1, 2, x, x + 1, x + 2, 2 x, 2 x + 1, 2 x + 2]
```

Now, we use the already defined procedure `tabl` to build the multiplication table of the 9-element field. As before, the elements in the preceding list correspond, in the given order, to the files and columns of the following matrix, in which the element in the (i, j) position is the result of multiplying the ith and the jth element of the list:

```
> tabl(%, mult9)
```

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & x & x+1 & x+2 & 2x & 2x+1 & 2x+2 \\
0 & 2 & 1 & 2x & 2x+2 & 2x+1 & x & x+2 & x+1 \\
0 & x & 2x & 2 & x+2 & 2x+2 & 1 & x+1 & 2x+1 \\
0 & x+1 & 2x+2 & x+2 & 2x & 1 & 2x+1 & 2 & x \\
0 & x+2 & 2x+1 & 2x+2 & 1 & x & x+1 & 2x & 2 \\
0 & 2x & x & 1 & 2x+1 & x+1 & 2 & 2x+2 & x+2 \\
0 & 2x+1 & x+2 & x+1 & 2 & 2x & 2x+2 & x & 1 \\
0 & 2x+2 & x+1 & 2x+1 & x & 2 & x+2 & 1 & 2x
\end{bmatrix}$$

There are several natural ways to represent the elements of the field $\mathbb{F}_{p^n}$, some of which are more convenient than the polynomials we have used to define the field structure. If to each polynomial of degree $< n$ we associate the list (or the string) of its coefficients, then we have a list of base-p digits that define a number in the interval $[0, p^n - 1]$ of which these digits are the p-adic expansion. Thus we see that we have natural bijections:

$$\mathbb{Z}_p[x]_f \longleftrightarrow \mathbb{Z}_p^n \longleftrightarrow \{0, 1, \ldots, p^n - 1\} = \mathbb{Z}_{p^n},$$

whose composition is given by $\sum_{i=0}^{n-1} a_i x^i \mapsto \sum_{i=0}^{n-1} a_i p^i$, i.e., it assigns to each polynomial its value for $x = p$. In this way we can deal with finite fields in a very convenient form that represents the elements of $\mathbb{F}_{p^n}$ as the integers in the interval $[0, p^n - 1]$. One should be careful, however, for, as we have seen in the case of $\mathbb{F}_4$, neither the sum nor the multiplication in this field are the same as the corresponding operations of $\mathbb{Z}_{p^n}$ when $n > 1$. In this case, $\mathbb{Z}_{p^n}$ is not a field and the bijections defined above are not (ring) isomorphisms. Indeed, the sum in $\mathbb{F}_{p^n}$ is given by the "digitwise" sum in $\mathbb{Z}_p$, i.e., to sum two elements regarded as integers, they are first expressed as sequences of digits in base p and then the corresponding digits are

added mod p. In other words, the sum consists in computing the p-adic expansion of the integers and then adding the base-p digits without carry. On the other hand, multiplying two integers in $[0, p^n - 1]$ consists simply of taking the polynomials associated to these integers and multiplying them modulo f (so that, in contrast with addition, multiplication depends on the particular choice of f). The matrix that defines the multiplication table of $\mathbb{F}_9$ that we have just seen can be put in integer form as follows (assuming that the result of computing the previous table in polynomial form is the last output produced by Maple):

```
> subs(x = 3, %);
```

$$\begin{bmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&1&2&3&4&5&6&7&8 \\ 0&2&1&6&8&7&3&5&4 \\ 0&3&6&2&5&8&1&4&7 \\ 0&4&8&5&6&1&7&2&3 \\ 0&5&7&8&1&3&4&6&2 \\ 0&6&3&1&7&4&2&8&5 \\ 0&7&5&4&2&6&8&3&1 \\ 0&8&4&7&3&2&5&1&6 \end{bmatrix}$$

Looking at this matrix we see, for example, that the product of 3 by 5 is 8 in $\mathbb{F}_9$ which corresponds to the fact that the polynomial x (which corresponds to the integer 3) multiplied by the polynomial $x + 2$ (corresponding to 5) gives as a result in $\mathbb{Z}_3[x]$ the polynomial $x^2 + 2x$ and the remainder of dividing this polynomial by the irreducible $x^2 + 1$ that we used to define the field is $2x + 2$ which corresponds to the integer 8.

Exercise 2.35 Consider $\mathbb{Z}_3[x]_f$ with $f = x^3 + 2x + 1 \in \mathbb{Z}_3[x]$. Prove that f is irreducible and hence that it defines the field $\mathbb{F}_{3^3}$. Compute the product of 25 by 11 when these integers are seen as elements of this field.

2.8.4 The Field of 256 Elements

The field $\mathbb{F}_{2^8}$ is especially important for cryptography because it is the one on whose arithmetic the *Advanced Encryption Standard* is based. We are going to describe this field in detail with the help of Maple. Although we have not used it so far, Maple has a specific package for the arithmetic of finite fields, namely, the package `GF` whose calling sequence is

```
> GF(p, n, f):
```

where p is a prime number, n a positive integer and f an irreducible polynomial of degree n over $\mathbb{Z}_p$, with the last parameter being optional (if not specified, Maple itself chooses the irreducible polynomial of degree n to be used).

We are going to use this package, and the "first" irreducible polynomial of degree 8 over $\mathbb{Z}_2[x]$, $x^8 + x^4 + x^3 + x + 1$ (already mentioned in a previous example), to

define the field $\mathbb{F}_{2^8}$. This particular polynomial is especially convenient because it is the one used in the specification of the Advanced Encryption Standard (cf. Chap. 4). Thus we initialize this field as:

```
> F256 := GF(2, 8, x^8+x^4+x^3+x+1):
```

If the field is defined by the command `F := GF(p, n)`, i.e., if the irreducible polynomial is not specified, then Maple chooses an irreducible polynomial which need not be the "first" one. The irreducible polynomial used by Maple in this case is given by the command `F[extension]`.

Exercise 2.36 Define $G := GF(2, 8)$ and find the irreducible polynomial used by Maple to construct this field.

The package `GF` has several commands to convert field elements between the different formats they can take. The elements of $\mathbb{F}_{2^8}$ are binary polynomials of degree less than 8 and, as we have mentioned before, there is a natural bijective correspondence between them and the integers in the 0..255 range, and also with the set of bit strings or lists of length 8 which are the binary representations of these integers. These bit strings are commonly referred to as *bytes* so we freely use this term—as we have already been doing—to denote the integers in the 0..255 range when regarded as elements of this field. The Maple functions that convert between these integers and polynomials modulo 2 are `F256:-input` and `F256:-output` (assuming that we defined the field `F256` with the command above) so that, for example, in Maple v13 or later we have[9]:

```
> F256:-input~([$127 .. 132)]);
    [(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1) mod 2, x^7 mod 2, (x^7 + 1) mod 2,
      (x^7 + x) mod 2, (x^7 + x + 1) mod 2, (x^7 + x^2) mod 2]
```

Now we can go back from binary polynomials to integers as follows:

```
> F256:-output~(%)
                        [127, 128, 129, 130, 131, 132]
```

Note that the outputs of the function `F256:-input` are polynomials modulo 2 (or, more generally, modulo p), and for doing arithmetic with these polynomials one should use the `modp1` function of Maple. For example, let

```
> g := F256:-input(7);
                                (x^2 + x + 1) mod 2
```

Then, if we want to check whether g is irreducible `Irreduc(g) mod 2` would fail, as g is already a binary polynomial, so we would use `modp1` instead:

```
> modp1(Irreduc(g), 2);
                                      true
```

Maple also has commands `F256:-ConvertIn` and `F256:-ConvertOut` to convert from modulo 2 polynomials to ordinary polynomials and vice versa. For example:

[9] In Maple versions prior to v13 the elementwise operator ~ is not available and so one would use `map` instead to make these conversion functions act on the elements of a list.

```
> F256:-ConvertOut(g);
  F256:-ConvertIn(%);
                                  x^2 + x + 1
                              (x^2 + x + 1) mod 2
```

Maple performs the `F256` arithmetic by means of the functions `F256:-'+'`, `F256:-'-'`, `F256:-'*'`, `F256:-'^'`, `F256:-inverse`, and `F256:-'/'`. Field elements must be passed to these functions as binary polynomials. For the addition we will not make use of these functions and, for efficiency, we will instead use the previously given function `BitXor`. The multiplication can then be defined as:

```
> mult256 := (a, b) -> F256:-output(F256:-'*'(F256:-input(a), F256:-input(b))):
```

For example,

```
> mult256(89, 175);
                                      198
```

We will see that this operation is heavily used in the Advanced Encryption Standard and hence, for efficiency, it is convenient to have it implemented by means of a lookup table. This can be done in the same way as in our implementation of `BitXor`.

Exercise 2.37 Construct a Maple 2-dimensional `(0..255, 0..255) Array` containing the multiplication table of `F256` and use it to define a more efficient multiplication function.

Another operation that will be used in the sequel is the computation of inverses in `F256`, which we shall also implement by means of a lookup table. For now, note that the function `(F256:-output@F256:-inverse@F256:-input)` gives the inverse of a nonzero byte (as an integer in the 1..255 range). For example, the inverse of 73 in this field is 100 because:

```
> (F256:-output@F256:-inverse@F256:-input)(73);
                                      100
```

2.8.5 The Multiplicative Group of a Finite Field

Let $\mathbb{F}_q$, with $q = p^n$, be a finite field with q elements. We are going to see that the group of units of this field, i.e., the multiplicative abelian group $\mathbb{F}_q^* = \mathbb{F}_q - \{0\}$ of all the nonzero elements of $\mathbb{F}_q$, is cyclic. First we show:

Theorem 2.23 *For every divisor d of $q - 1$ there are $\phi(d)$ elements of order d in $\mathbb{F}_q^*$.*

Proof Let $a \in \mathbb{F}_q^*$ be an element of order d, i.e., an element such that $a^d = 1$ and no lower power with positive exponent is equal to 1. Then the powers $a, a^2, \ldots, a^d = 1 \in \mathbb{F}_q$ are all distinct and are zeros of the polynomial $x^d - 1 \in \mathbb{F}_q[x]$. Since by Corollary 2.4 this polynomial has at most d zeros, we see that its zeros are all powers

of a. Therefore, all elements of order d, being zeros of $x^d - 1$, are powers of a. By Proposition 2.5, a power a^t of a has order d if and only if $\gcd(t, d) = 1$ and hence we have shown that if a has order d then the elements of order d are all the powers a^t with $\gcd(t, d) = 1$. Since there are $\phi(d)$ values of t such that $0 < t \leq d$ and $\gcd(t, d) = 1$, this means that if there is at least one element of order d, then there are exactly $\phi(d)$ such elements. To complete the proof of the theorem it remains to show that an $a \in \mathbb{F}_q^*$ of order d must exist. If we denote by $f(d)$ the number of elements of order d we know that either $f(d) = \phi(d)$ or $f(d) = 0$. But there are $q - 1$ elements in $\mathbb{F}_q^*$ and all of them have order equal to some divisor of $q - 1$, so that $\sum_{d|q-1} f(d) = q - 1$. On the other hand, by Proposition 2.8 we also have that $\sum_{d|q-1} \phi(d) = q - 1$. Thus $f(d)$ is always equal to $\phi(d)$ and never 0, which completes the proof. □

Corollary 2.5 *The multiplicative group $\mathbb{F}_q^*$ of a finite field is cyclic and has $\phi(q-1)$ generators.*

We can now see that, for every prime p, there are primitive roots modulo p:

Corollary 2.6 *If p is a prime number, then $\mathbb{Z}_p^*$ is a cyclic group of order $p-1$ which has $\phi(p-1)$ generators.*

Exercise 2.38 Show that if p is a prime number, then the only square roots of 1 modulo p are 1 and -1.

We have seen in Sect. 2.7 an algorithm to compute a generator of $\mathbb{Z}_p^*$ (for p prime) and its Maple implementation. That algorithm works in any finite cyclic group and the corresponding Maple function can easily be adapted to find a generator of the multiplicative group of a finite field. However, it is not necessary to do this because Maple already has this function implemented in the package `GF`. Using as example our previously defined 256-element field `F256`, the Maple functions to test whether a given field element is a generator and to pseudo-randomly choose a generator are `F256:-isPrimitiveElement` and `F256:-PrimitiveElement()`, respectively.

Example 2.27 Suppose we want to check whether the element corresponding to the polynomial x is a generator. We compute:

```
> F256:-isPrimitiveElement(F256:-ConvertIn(x));
                                    false
```

Thus this element is not a generator. Let's now check the element corresponding to $x + 1$:

```
> a := F256:-ConvertIn(x+1);
                                a:=(x + 1) mod 2

> F256:-isPrimitiveElement(a);
                                    true
```

We see that this is a generator of the multiplicative group of the field. To check in a more explicit way that this is indeed the case, we would apply Proposition 2.6 as follows:

```
> ifactor(255);
                                    (3) (5) (17)
> F256:-`^`(a, 85);
                        (x^7 + x^5 + x^4 + x^3 + x^2 + 1) mod 2
> F256:-`^`(a, 51);
                                 (x^3 + x^2) mod 2
> F256:-`^`(a, 15);
                              x^5 + x^4 + x^2 + 1) mod 2
```

We see that when `a` is raised to the exponents obtained by dividing the order of the group, 255, by each of its prime factors, an element different from the identity is obtained. Thus the order of `a` must be equal to the order of the multiplicative group, i.e., 255, and hence `a` is a generator. If we want to obtain a generator, we just compute:

```
> randomize():
  F256:-PrimitiveElement();
                                  (x^7 + x^4) mod 2
```

Exercise 2.39 Write a Maple function that tests whether a given element of a finite field defined by means of the `GF` package is a generator of the multiplicative group of the field and another one that finds a generator by pseudo-randomly selecting elements of the field and applying the previous function to check whether they are generators. (Hint: One may have a look at Maple's code for the corresponding functions by printing the module `GF`; this requires executing `interface (verboseproc = 2)` and then `print(GF)`.)

2.9 Quadratic Residues and Modular Square Roots

In this section we study quadratic residues and some related algorithms that have interesting cryptographic applications.

2.9.1 Quadratic Residues and the Legendre and Jacobi Symbols

Definition 2.26 Let a be an integer and $n > 1$. We say that a is a *quadratic residue* modulo n if $\gcd(a, n) = 1$ and the congruence $x^2 \equiv a \pmod{n}$ has a solution. In other words, a is a quadratic residue if and only if $a \bmod n$ is a square in $\mathbb{Z}_n^*$. If a is not a quadratic residue modulo n then we say that a is a *quadratic non-residue* modulo n.

Remark 2.7 Whether an integer a is a quadratic residue or a quadratic non-residue depends only on the congruence class of a modulo n and the quadratic residues correspond to the squares in the group $\mathbb{Z}_n^*$ (if $a \in (\mathbb{Z}_n^*)^2 = \{x^2 | x \in \mathbb{Z}_n^*\}$, we will not distinguish between saying that a is a square in $\mathbb{Z}_n^*$ or that a is a quadratic residue). For example, in $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ we see that the squares are 1 and 4. In fact, we have that in this group $1 = 1^2 = 4^2 = 11^2 = 14^2$ and $4 = 2^2 = 7^2 = 8^2 = 13^2$ so that each of these squares has exactly four square roots.

Example 2.28 We do a slightly larger example using Maple (much larger examples are possible but it is not practical to print the results!). Taking into account that if $b^2 = a$ then also $(-b)^2 = a$ (where $-b$ is the opposite of b in $\mathbb{Z}_n$) we see that in order to compute all the squares in $\mathbb{Z}_n^*$ (and hence all the quadratic residues modulo n), we only have to compute the squares of the elements $1, 2, \ldots, \lfloor n/2 \rfloor$ (because the remaining elements of $\mathbb{Z}_n^*$ are opposites of some of these and, in fact, as seen in the preceding remark, there can even be different elements among them with the same square). Then the function that computes the quadratic residues in $\mathbb{Z}_n^*$ is the following:

```
> qr := n -> sort(ListTools:-MakeUnique(
         map(x -> x&^2 mod n, select(x -> evalb(igcd(x,n)=1), [$1..floor(n/2)])))):
```

which, when applied to $n = 187 = 11 \cdot 17$, gives:

```
> qr(187);
[1, 4, 9, 15, 16, 25, 26, 36, 38, 42, 47, 49, 53, 59, 60, 64, 67, 69, 70, 81,
  86, 89, 93, 100, 103, 104, 111, 115, 135, 137, 144, 152, 155, 157, 166, 168,
  169, 174, 179, 185]
```

We see that in $\mathbb{Z}_{187}^*$ there are exactly 40 squares:

```
> nops(%);
                                    40
```

while the order of the group $\mathbb{Z}_{187}^*$ is:

```
> numtheory:-phi(187);
                                   160
```

As in the previous remark we see that exactly one-fourth of the elements in the group are squares and this has not happened by chance. It is a consequence of the fact that, as we will see later, when n is, as in this case, a product of two distinct odd primes, each square has four square roots. For example, in $\mathbb{Z}_{187}^*$, the four square roots of 1 are 1, 67, 120, 186 (see Exercise 2.40 below).

Exercise 2.40 Write a Maple procedure that uses brute force to find all the square roots of a quadratic residue in the preceding example (and in similar examples; we will later give a more efficient algorithm to extract modular square roots). Use it to compute the four square roots of 1 and the four square roots of 115 in $\mathbb{Z}_{187}^*$.

We will denote by $\mathcal{QR}_n$ and $\mathcal{QN}_n$ the sets of quadratic residues and quadratic non-residues in $\mathbb{Z}_n^*$, respectively; note that $\mathcal{QR}_n$ is the same as the set of the squares $(\mathbb{Z}_n^*)^2$ which is, clearly, a subgroup of $\mathbb{Z}_n^*$:

Exercise 2.41 Let $n > 1$ be an integer. Prove that $\mathcal{QR}_n$ is a subgroup of $\mathbb{Z}_n^*$.

We will be especially interested in the set of squares $\mathcal{QR}_p$ of $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ for an odd prime $p > 2$. Observe that 1 is always a quadratic residue and, moreover, modulo a prime p its only square roots are ± 1. This is because they are zeros of the polynomial $x^2 - 1 \in \mathbb{Z}_p[x]$ which, as we have seen, are at most two. More generally, the same argument shows that an element of $\mathbb{Z}_p^*$ has either two square roots (if it is a quadratic residue) or none (if it is a quadratic non-residue). Next we show that $\mathcal{QR}_p$ and $\mathcal{QN}_p$ have the same number of elements:

Proposition 2.11 *Let p be an odd prime. Then, $|\mathcal{QR}_p| = |\mathcal{QN}_p| = (p-1)/2$.*

Proof If $a \in \mathbb{Z}_p^*$ is a square then $a = b^2$ for some $b \in \mathbb{Z}_p^*$ and, as we have already remarked, a has precisely two square roots that must be b and $-b$. If we consider the squaring function $sq : \mathbb{Z}_p^* \to \mathcal{QR}_p$, given by $sq(x) = x^2$, we see that each quadratic residue is mapped onto by two elements of $\mathbb{Z}_p^*$, from which the result follows. □

Remark 2.8 If $g \in \mathbb{Z}_p^*$ is a generator, then the squares in $\mathbb{Z}_p^*$ are precisely the elements that can be written in the form g^i with i even (for, in that case, $g^i = (\pm g^{i/2})^2$). In particular, we see that a generator of $\mathbb{Z}_p^*$ is always a quadratic non-residue.

A convenient notation used to identify when an integer is a quadratic residue modulo p is given by the Legendre symbol which we now define:

Definition 2.27 Let a be an integer and $p > 2$ a prime. The *Legendre symbol* $\left(\frac{a}{p}\right)$ is defined by:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p|a \\ +1 & \text{if } a \text{ is a quadratic residue mod } p \\ -1 & \text{if } a \text{ is a quadratic non-residue mod } p. \end{cases}$$

Example 2.29 We will see how the Legendre symbol can be efficiently computed; Maple does it with the function `numtheory:-legendre` which can be used to find quadratic residues and quadratic non-residues modulo primes. More generally, the function `numtheory:-quadres` can be used to search for quadratic residues even when the modulus is not prime: like the Legendre symbol this function takes the value $+1$ on quadratic residues and the value -1 on quadratic non-residues. Let us consider the prime 101 and compute the quadratic residues in $\mathbb{Z}_{101}^*$:

```
> qr101 := select(x -> evalb(numtheory:-quadres(x, 101) = 1), {$1 .. 100})
{1, 4, 5, 6, 9, 13, 14, 16, 17, 19, 20, 21, 22, 23, 24, 25, 30, 31, 33, 36,
  37, 43, 45, 47, 49, 52, 54, 56, 58, 64, 65, 68, 70, 71, 76, 77, 78, 79, 80,
  81, 82, 84, 85, 87, 88, 92, 95, 96, 97, 100}
```

We check that exactly one half of the elements of $\mathbb{Z}_{101}^*$, namely 50 elements, are quadratic residues:

```
> nops(qr101);
                                    50
```

Next, we compute a generator of $\mathbb{Z}_{101}^*$:

```
> findgenerator(101);
                                    42
```

We compute all the even-exponent powers of the generator $42 \in \mathbb{Z}_{101}^*$:

```
> evenpowers42mod101 := map(x -> 42&^x mod 101, {seq(2*i, i = 1 .. 50)});
{1, 4, 5, 6, 9, 13, 14, 16, 17, 19, 20, 21, 22, 23, 24, 25, 30, 31, 33, 36,
  37, 43, 45, 47, 49, 52, 54, 56, 58, 64, 65, 68, 70, 71, 76, 77, 78, 79, 80,
  81, 82, 84, 85, 87, 88, 92, 95, 96, 97, 100}
```

Finally, we check that these powers coincide with the quadratic residues in $\mathbb{Z}_{101}^*$:

```
> evalb(qr101 = evenpowers42mod101);
                                   true
```

The following result gives a simple way to compute the Legendre symbol:

Proposition 2.12 (Euler's criterion) *Let p be an odd prime and a an integer. Then*

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Proof If $p|a$ then both sides of the congruence are congruent to 0 modulo p. Hence we may suppose that p does not divide a and, by Fermat's little theorem (Theorem 2.2) we have that $a^{p-1} \equiv 1 \pmod{p}$ so that $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. Let g be a generator of $\mathbb{Z}_p^*$ and $a = g^i$. Since a is a quadratic residue if and only if i is even, we see that the left side of the congruence in the statement of the proposition is equal to 1 if and only if i is even. The right side of this congruence is then equal to $g^{i(p-1)/2}$, which is 1 if and only if $(p-1)$ divides $i(p-1)/2$, i.e., if and only if i is even. Thus both sides of the congruence are ± 1 and each side is 1 if and only if i is even, which completes the proof. □

Remark 2.9 Euler's criterion provides a polynomial-time algorithm to compute the Legendre symbol and hence to decide whether an element of $\mathbb{Z}_p^*$ is a quadratic residue or not. The algorithm performs a modular exponentiation modulo p with an exponent less than p and hence its complexity is $O(\text{len}(p)^3)$. However, we shall see later that there is a faster algorithm to compute the Legendre symbol.

The next proposition collects several basic properties of the Legendre symbol:

Proposition 2.13 *Let p be an odd prime. Then the Legendre symbol satisfies:*

(i) *If $a \equiv b \pmod{p}$ then $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.*

(ii) $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$.

(iii) If p does not divide b then $\left(\frac{ab^2}{p}\right) = \left(\frac{a}{p}\right)$.

(iv) $\left(\frac{1}{p}\right) = 1$ *and* $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$.

Proof As we have already remarked, (i) is a straightforward consequence of the definition. (ii) follows from Euler's criterion since $a^{(p-1)/2} \cdot b^{(p-1)/2} \equiv (ab)^{(p-1)/2} \pmod{p}$. (iii) is an immediate consequence of (ii). For (iv), observe first that $1 = 1^2$ is always a quadratic residue and so $\left(\frac{1}{p}\right) = 1$. Finally, $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$ by Euler's criterion. □

Remarks 2.6

1. As a consequence of part (ii) of the preceding proposition, a product of two elements of $\mathbb{Z}_p^*$ (where p is an odd prime) is a quadratic residue if and only if either both factors are quadratic residues or both are quadratic non-residues. Also, (ii) reduces the computation of the Legendre symbol of a to those of the Legendre symbols of its prime factors but this does not give an efficient method to compute the Legendre symbol because factoring a is usually difficult.
2. The fact that $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$ is obviously equivalent to:

$$\left(\frac{-1}{p}\right) = \begin{cases} 1 & \text{if } p \equiv 1 \pmod 4 \\ -1 & \text{if } p \equiv 3 \pmod 4. \end{cases}$$

Exercise 2.42 Write a Maple function that, given a positive integer n, finds the smallest prime p such that a is a quadratic residue modulo p for all $1 \leq a \leq n$. Use it to find the smallest prime p whose least quadratic non-residue is greater than 50 and compute the value of this quadratic non-residue. Perform a similar computation for other small values of n (it has been conjectured that the least quadratic non-residue modulo p is $O(\ln p \ln\ln p)$ but no deterministic algorithm which is polynomial on the size of p is known to find a quadratic non-residue).

The most efficient method to compute the Legendre symbol $\left(\frac{a}{p}\right)$ is obtained by using a generalization called the *Jacobi symbol*:

Definition 2.28 Let a be an integer and n a positive odd integer with prime factorization $n = \prod_{i=1}^{r} p_i^{e_i}$. Then the *Jacobi symbol* $\left(\frac{a}{n}\right)$ is defined by:

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{r} \left(\frac{a}{p_i}\right)^{e_i},$$

where the symbols on the right side are Legendre symbols and we define $\left(\frac{a}{1}\right) = 1$.

Remarks 2.7

1. If n is itself prime, then the Jacobi symbol $\left(\frac{a}{n}\right)$ is just the Legendre symbol. Thus the Jacobi symbol is a generalization of the Legendre symbol and, like the latter, it clearly depends only on the congruence class of a modulo n.
2. The Jacobi symbol is multiplicative in both arguments, namely $\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$ and $\left(\frac{a}{mn}\right) = \left(\frac{a}{m}\right)\left(\frac{a}{n}\right)$.
3. If a is a quadratic residue modulo n, then it is clear that a is also a quadratic residue modulo p for every prime p that divides n, and hence it follows from the definition that, in this case, $\left(\frac{a}{n}\right) = 1$. But what about the converse? If n is prime we know that the converse is also true but suppose, for example that $n = pq$ is the product of two different odd primes. Then it may happen that $\left(\frac{a}{p}\right) = -1$ and $\left(\frac{a}{q}\right) = -1$, in which case $\left(\frac{a}{n}\right) = 1$ but, clearly, a is not a quadratic residue modulo n (as this would imply that it is a quadratic residue modulo p and also modulo q). A concrete example is $\left(\frac{2}{15}\right) = \left(\frac{2}{3}\right)\left(\frac{2}{5}\right) = (-1)(-1) = 1$ where, as we have seen, 2 is a quadratic non-residue modulo 15.

Exercise 2.43 Let $\mathcal{J}_n = \{a \in \mathbb{Z}_n^* | \left(\frac{a}{n}\right) = 1\}$. Prove that if n is a square, then $\mathcal{J}_n = \mathbb{Z}_n^*$ and otherwise $|\mathcal{J}_n| = \phi(n)/2$, i.e., half of the elements of $\mathbb{Z}_n^*$ are in $\mathcal{J}_n$. (Hint: If n is not a square then there exists a prime factor p of n such that the largest power of p that divides n has an odd exponent. Use the Chinese remainder theorem to find an element which is not a quadratic residue modulo p but is a quadratic residue modulo all the remaining prime divisors of n and show that multiplication by this element gives a bijective map from $\mathcal{J}_n$ to its complement in $\mathbb{Z}_n^*$.)

The Jacobi symbol can be efficiently computed if one knows the factorization of n for, in that case, it reduces to the computation of Legendre symbols which may be done, for example, by means of Euler's criterion. But there is another method that allows the computation of the Jacobi symbol without doing any factoring—except pulling out powers of 2, which is easy—and the amazing thing is that it gives an even more efficient algorithm to compute Legendre symbols. This method is based on the *Law of Quadratic Reciprocity* that was proved by Gauss for the Legendre symbol and holds equally for the Jacobi symbol. There are more than 200 known proofs of quadratic reciprocity (see [125]) and we refer to any book introducing number theory—like, for example, [77]—for some of them; among the books related to cryptography we refer to [118, 180] for proofs.

Theorem 2.24 *Let m and n be positive odd integers. Then the following conditions hold:*

(i) $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2} = \begin{cases} 1 \quad if\ n \equiv 1 \pmod 4 \\ -1\ if\ n \equiv 3 \pmod 4. \end{cases}$

(ii) $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8} = \begin{cases} 1 \quad if\ n \equiv \pm 1 \pmod 8 \\ -1\ if\ n \equiv \pm 3 \pmod 8. \end{cases}$

(iii) (Law of Quadratic Reciprocity)

$$\left(\frac{m}{n}\right) = (-1)^{(n-1)(m-1)/4}\left(\frac{n}{m}\right) = \begin{cases} -\left(\frac{n}{m}\right) & if\ m \equiv n \equiv 3 \pmod 4 \\ \left(\frac{n}{m}\right) & otherwise. \end{cases}$$

Exercise 2.44 Use quadratic reciprocity to characterize the odd primes p such that 5 is a quadratic residue modulo p.

Remarks 2.8

1. When $n = p$ is prime, part (i) of Theorem 2.24 tells us that whether -1 is a square modulo p (or, equivalently, $p - 1 \in \mathbb{Z}_p^*$ is a square) depends only on the congruence class of p modulo 4 and, similarly, part (ii) says that the question whether 2 is a square depends only on the congruence class of p modulo 8.
2. We mentioned above that quadratic reciprocity can be used to compute the Legendre symbol and hence to determine whether an integer a is a square modulo a prime p. This is done by using part (iii) of Theorem 2.24 to repeatedly flip the Jacobi symbol, reducing the top number modulo the bottom number after each flip (and pulling out powers of 2 by means of part (ii) of the theorem whenever the top number obtained after the reduction is even). As in the Euclidean algorithm, these numbers decrease rapidly in size which gives a very efficient algorithm. In the next example we show how this works.

Example 2.30 We determine whether 41069779796933 is a quadratic residue modulo 673275095033 (in this example both numbers are prime). The steps applied are the following:

- If both numbers are odd and the top number is greater than the bottom one, we replace the top number by the result of reducing it modulo the bottom number.
- If the top number is even, we use part (ii) of Theorem 2.24 to pull out the largest power of 2 that divides it.
- If both numbers are odd and the top number is smaller than the bottom one, we use part (iii) of Theorem 2.24 to interchange the position of both numbers.

We then have the following computation, where we only point out the use of each of the previous three steps the first time that they are used:

$$\left(\frac{41069779796933}{673275095033}\right) = \text{(reducing the top number modulo the bottom number)}$$
$$= \left(\frac{673274094953}{673275095033}\right) = \text{(by (iii) since the top number is congruent to 1 modulo 4)}$$
$$= \left(\frac{673275095033}{673274094953}\right) = \left(\frac{1000080}{673274094953}\right) = \left(\frac{2^4 \cdot 62505}{673274094953}\right)$$
$$= \text{(by (ii))}\left(\frac{62505}{673274094953}\right) = \left(\frac{673274094953}{62505}\right) = \left(\frac{49838}{62505}\right) = \left(\frac{2 \cdot 24919}{62505}\right)$$
$$= \left(\frac{24919}{62505}\right) = \left(\frac{62505}{24919}\right) = \left(\frac{12667}{24919}\right) = -\left(\frac{24919}{12667}\right) = -\left(\frac{12252}{12667}\right)$$

$$= -\left(\frac{2^2 \cdot 3063}{12667}\right) = -\left(\frac{3063}{12667}\right) = \left(\frac{12667}{3063}\right) = \left(\frac{415}{3063}\right)$$
$$= -\left(\frac{3063}{415}\right) = -\left(\frac{158}{415}\right) = -\left(\frac{2 \cdot 79}{415}\right) = -\left(\frac{79}{415}\right) = \left(\frac{415}{79}\right) = \left(\frac{20}{79}\right)$$
$$= \left(\frac{2^2 \cdot 5}{79}\right) = \left(\frac{5}{79}\right) = \left(\frac{79}{5}\right) = \left(\frac{4}{5}\right) = 1.$$

Thus we see that 41069779796933 is a quadratic residue modulo 673275095033.

The built-in Maple function `numtheory:-Jacobi` uses this algorithm to compute the Jacobi symbol $\left(\frac{m}{n}\right)$ by calling `numtheory:-Jacobi(m,n)`. In particular, this function also computes Legendre symbols and, as already mentioned, there is also `numtheory:-legendre` which is just the specialization of the preceding function when the second argument is prime. To carry out the preceding computation we can use either of these functions, for example:

```
> numtheory:-Jacobi(41069779796933, 673275095033);
                                    1
```

The algorithm applied in the preceding example is then the following. Note that the algorithm is recursive, i.e., it calls itself:

Algorithm 2.7. `Jacobi` (m, n) **(Computation of the Jacobi symbol)**.

Input: An integer m and an odd positive integer n.
Output: The Jacobi symbol $\left(\frac{m}{n}\right)$.

`if` $n = 1$ `then return` 1.
`Set` $m := m \bmod n$.
`if` $m = 0$ `then return` 0.
`if` $m = 1$ `then return` 1.
`Write` $m = 2^e m_1$ with m_1 odd.
`if` e is even `then`
    $s := 1$
`elif` $n \equiv \pm 1 \pmod 8$ `then`
    $s := 1$
`elif` $n \equiv \pm 3 \pmod 8$ `then`
    $s := -1$
`end if`.
`if` $m_1 \equiv 3 \pmod 4$ `and` $n \equiv 3 \pmod 4$ `then`
    $s := -s$
`end if`.
`return` $s \cdot$ `Jacobi`$(n, m1)$.

Remark 2.10 Except for the removal of powers of 2, the sequence of recursive calls in Algorithm 2.7 is the same as the one used by the Euclidean algorithm to compute $\gcd(m, n)$, and an analysis similar to that of the Euclidean algorithm shows that the complexity is also the same, so that the running time of this algorithm can be estimated as $O(\text{len}(m)\text{len}(n))$ or, assuming that $m < n$, $O(\text{len}(n)^2)$. Thus

we see that this algorithm computes the Legendre symbol more efficiently than Euler's criterion and hence so do Maple's functions `numtheory:-Jacobi` and `numtheory:-legendre`.

As we will see later, the quadratic residues modulo a product of two odd primes are of cryptographic interest. The following result characterizes them:

Proposition 2.14 *Let $n = pq$ be a product of two distinct primes and $x \in \mathbb{Z}_n^*$. Then x is a quadratic residue modulo n if and only if it is both a quadratic residue modulo p and a quadratic residue modulo q.*

Proof As we have already remarked, if x is a quadratic residue modulo n then it is also a quadratic residue modulo p and modulo q. Conversely, suppose that x is a quadratic residue modulo both primes, so that we have congruences $x \equiv a^2 \pmod{p}$ and $x \equiv b^2 \pmod{q}$. Then, the Chinese remainder theorem gives us a unique element $y \in \mathbb{Z}_n^*$ such that

$$\left.\begin{aligned} y &\equiv a \pmod{p} \\ y &\equiv b \pmod{q} \end{aligned}\right\} \tag{2.6}$$

Thus $x \equiv a^2 \equiv y^2 \pmod{p}$ and $x \equiv b^2 \equiv y^2 \pmod{q}$. Hence both p and q divide $x - y^2$ and, since they are different primes, so does their product n. Therefore $x \equiv y^2 \pmod{n}$ and x is indeed a quadratic residue modulo n. □

Remark 2.11 Note that the preceding proposition can also easily be deduced from Theorem 2.14. The proof thus obtained is essentially the same as the one above but it looks even more straightforward.

The following corollary explains something we noticed when computing the quadratic residues modulo 15: when n is a product of two distinct odd primes, exactly one-fourth of the elements of $\mathbb{Z}_n^*$ are quadratic residues:

Corollary 2.7 *If $n = pq$ is a product of two distinct odd primes then each quadratic residue modulo n has exactly four square roots in $\mathbb{Z}_n^*$.*

Proof Let x be a quadratic residue modulo n. Then x is both a quadratic residue modulo p and a quadratic residue modulo q, and hence it has two square roots modulo p and two square roots modulo q. Thus we get four different pairs formed by a square root of x modulo p and a square root of x modulo q. Each of these pairs defines a system of congruences that, by the Chinese remainder theorem, gives rise to a square root of x modulo n. □

2.9.2 Computing Modular Square Roots

We have seen how to determine whether the equation $x^2 \equiv a \pmod{p}$, where p is prime, has a solution, but now we are going to see how to find the solutions, i.e.,

how to compute modular square roots modulo a prime. The easiest case is when $p \equiv 3 \pmod 4$ (p is then called a *Blum prime*). For these primes we have:

Proposition 2.15 *If $p \equiv 3 \pmod 4$ is prime and a is a quadratic residue modulo p, then the square roots of a modulo p are $\pm a^{(p+1)/4} \bmod p$.*

Proof Using Euler's criterion (Proposition 2.12) and the fact that a is a quadratic residue modulo p, we have $(\pm a^{(p+1)/4})^2 \equiv a^{(p+1)/2} \equiv a \cdot a^{(p-1)/2} \equiv a \cdot \left(\frac{a}{p}\right) \equiv a \pmod p$. □

Observe that the argument used to prove the proposition does not work when $p \equiv 1 \pmod 4$ because we made use of the fact that $(p+1)/4$ is an integer, which is no longer true in this case. So we now describe an algorithm that computes square roots modulo any odd prime.

If p is an odd prime, we may write $p - 1 = 2^s t$ with t odd and consider $A = a^t \bmod p$. Then $A^{2^{s-1}} \equiv a^{(p-1)/2} \equiv 1 \pmod p$ by Euler's criterion, so we know from Proposition 2.4 that the order of A is a divisor of 2^{s-1}. Now let $h \in \mathbb{Z}_p^*$ be a quadratic non-residue and $H = h^t \bmod p$. The order of H is a power of 2 by Proposition 2.4 and, since $H^{2^{s-1}} \equiv h^{2^{s-1}t} \equiv h^{(p-1)/2} \equiv -1 \pmod p$ again by Euler's criterion, we see that the order of H is exactly 2^s. The order of its inverse $H^{-1} \in \mathbb{Z}_p^*$ is also 2^s and hence $\langle H \rangle = \langle H^{-1} \rangle$ is the unique (cyclic) subgroup of order 2^s of the cyclic group $\mathbb{Z}_p^*$, and it contains all the elements of $\mathbb{Z}_p^*$ whose order is a power of 2. In particular, $A \in \langle H^{-1} \rangle$ and hence A is a power of H^{-1} in $\mathbb{Z}_p^*$. Moreover, A is an even power of H^{-1} because otherwise it would follow from Proposition 2.5 that A has the same order as H^{-1}, namely 2^s, in contradiction with the fact that, as we have seen, the order of A divides 2^{s-1}. Thus we see that there exists an integer u such that $0 \leq u < 2^{s-1}$ and $A \equiv H^{-2u} \pmod p$. Since $A \equiv a^t \pmod p$, this is equivalent to $a^t H^{2u} \equiv 1 \pmod p$ and, multiplying this congruence by a, we obtain $a \equiv a^{(t+1)} H^{2u} \pmod p$. Now the terms on the right side of this congruence all have even exponents and we find a square root of a modulo p by halving these exponents, namely, by computing $a^{(t+1)/2} H^u \bmod p$ (see [60]).

In order to complete the description of the square roots algorithm we need to show how to find the quadratic non-residue h and how to find the integer u such that $A \equiv H^{-2u} \pmod p$. The first problem is very interesting because no deterministic polynomial-time algorithms to find a quadratic non-residue modulo p are known.[10] It is easy, however, to describe a probabilistic polynomial-time algorithm that does the job: just pick random elements in $\mathbb{Z}_n^*$ and use either Euler's criterion or Algorithm 2.7 to check whether they are quadratic non-residues. Since, as we have seen, exactly one-half of the integers in $\mathbb{Z}_p^*$ are quadratic non-residues, the expected number of trials until finding one of them is 2.

Thus, in order to complete the algorithm, it only remains to find the integer u satisfying the conditions explained above. We show how to do it in Algorithm 2.8:

[10] As in the case of primitive roots, there is a deterministic polynomial-time algorithm under the assumption that the *Extended Riemann Hypothesis* holds.

Algorithm 2.8. Computation of square roots modulo an odd prime.

Input: An odd prime p and an integer a such that $\left(\frac{a}{p}\right) = 1$.
Output: A square root of $a \bmod p$.

1. **Case** $p \equiv 3 \pmod 4$:
   $a := a \bmod p$;
   `return` $a^{(p+1)/4} \bmod p$.

2. **Case** $p \equiv 1 \pmod 4$:
   Find a random quadratic non-residue $h \in \mathbb{Z}_p^*$
   Compute $s \geq 0$ and t odd such that $p - 1 = 2^s t$
   $A := a^t \bmod p$;
   $H := h^t \bmod p$;
   $v := 0$;
   `for` i from 1 to $s - 1$ `do`
      `if` $(AH^v)^{2^{s-1-i}} \equiv -1 \pmod p$ `then`
         $v := v + 2^i$
      `end if`;
   `end do`; #Now $AH^v \equiv 1 \pmod p$, with v even
   `return` $a^{(t+1)/2} H^{v/2} \bmod p$.

This algorithm is essentially due to Tonelli, who published it already in 1891. We next prove the correctness of the algorithm and estimate its complexity:

Proposition 2.16 *Algorithm 2.8 is correct and produces as output a square root of the quadratic residue a modulo p. The expected running time of the algorithm is $O(\text{len}(p)^4)$.*

Proof We have already shown that the algorithm is correct if $p \equiv 3 \pmod 4$, when $a^{(p+1)/4} \bmod p$ is a square root of a modulo p. So suppose that $p \equiv 1 \pmod 4$ in which case we have to show that the value returned after running the `for` loop is indeed a square root of a modulo p. It is clear that $s \geq 2$ and v is initialized as $v := 0$ so the loop is started by computing $A^{2^{s-2}} \bmod p$, which is a square root in $\mathbb{Z}_p^*$ of $A^{2^{s-1}} \bmod p = 1$ and hence is equal to ± 1 (in fact, the loop can also be started at $i = 0$ but this is unnecessary because, since a is a quadratic residue, we already know that $A^{2^{s-1}} \bmod p = 1$). We show by induction on i that, after running the ith iteration of the loop which includes updating the value of v, $(AH^v)^{2^{s-1-i}} \equiv 1 \pmod p$, i.e., the value 1 is maintained for this expression after each iteration of the loop. We may start at $i = 0$, after which we have the value 1 as already remarked. So suppose that we also have the value 1 after the $(i-1)$th iteration of the loop, i.e., $(AH^v)^{2^{s-i}} \equiv 1 \pmod p$. Then in the ith iteration we compute $(AH^v)^{2^{s-i-1}} \bmod p$ which is a square root of the previous value 1 (as v has not been updated yet) and hence it is equal to ± 1. If the value is 1 then we have nothing to prove, so assume that $(AH^v)^{2^{s-i-1}} \equiv -1 \pmod p$. In this case the value of v is updated in the loop to $v := v + 2^i$ and so we have to show that $(AH^{v+2^i})^{2^{s-i-1}} \equiv 1 \pmod p$. This is indeed true because $(AH^{v+2^i})^{2^{s-i-1}} \equiv (AH^v)^{2^{s-i-1}} (H^{2^i})^{2^{s-1-i}} \equiv (-1) H^{2^{s-1}} \equiv$

$(-1)(-1) \equiv 1 \pmod{p}$ (where we used the fact that $H^{2^{s-1}} \equiv -1 \pmod{p}$, as we have seen in our previous discussion). After the last iteration of the loop, corresponding to $i = s - 1$, we have that $AH^v \equiv 1 \pmod{p}$ and hence the final value of v is the integer $2u$ in our previous discussion and $a^{(t+1)/2}H^{v/2} \bmod p$ is indeed a square root of a modulo p.

For the complexity of the algorithm observe that, given a non-square h, outside the `for` loop only a few exponentiations modulo p are computed in time $O(\text{len}(p)^3)$. The loop has $s - 1 = O(\text{len}(p))$ iterations in which, again, modular exponentiations are carried out, so the complexity is $O(\text{len}(p)\text{len}(p)^3) = O(\text{len}(p)^4)$. Finally, as we have remarked, the expected number of trials to find a non-residue is 2 and each trial uses Algorithm 2.7 which has complexity $O(\text{len}(p)^2)$ so the expected running time of this part of the algorithm is $O(\text{len}(p)^2)$. Thus the expected running time of Algorithm 2.8 is $O(\text{len}(p)^4)$. □

Remark 2.12 Observe that in the estimation of the complexity of Algorithm 2.8 we used the fact that $s = O(\text{len}(p))$. Since s is usually much smaller than $\text{len}(p)$ this bound is not very tight and, by considering average instead of worst-case complexity, one obtains the estimate $O(\text{len}(p)^3)$, which better reflects the behavior of the algorithm over many random primes.

Exercise 2.45 Prove that the case $p \equiv 3 \pmod 4$ of Algorithm 2.8 is also handled by the algorithm used when $p \equiv 1 \pmod 4$, so that it was not strictly necessary to include it as a separate case. To see this, show that if $p \equiv 3 \pmod 4$ then $s = 1$ and hence A has order 1, that is $A \equiv 1 \pmod p$. Thus $v = 0$ and the root is obtained as $a^{(t+1)/2} \equiv a^{(p+1)/4} \pmod p$.

Remark 2.13 When $p \equiv 5 \pmod 8$, Algorithm 2.8 can be modified to use $h = 2$ instead of a random non-square. Then we can prove that the algorithm computes a square root x of a as follows: set $x := a^{(p+3)/8} \bmod p$ and if $x^2 \not\equiv a \pmod p$ then $x := x2^{(p-1)/4} \bmod p$. Indeed, it is easy to see that in this case we have $s = 2$ and $t = (p-1)/4$ and, since 2 is a quadratic non-residue modulo p by Theorem 2.24, we may choose $h = 2$. Then, when entering the `for` loop with $v = 0$ and $i = 1$ we have to compute $A = a^t = a^{(p-1)/4} \bmod p$. This can only take the values 1 or -1. The first case occurs when $(a^{(p+3)/8})^2 \equiv a^{(p+3)/4} \equiv a \cdot a^{(p-1)/4} \equiv a \pmod p$, so that $a^{(p+3)/8} \bmod p$ is indeed a square root in this case. The second case makes $v := 2$ and returns the root $x \equiv a^{(t+1)/2}2^{(p-1)/4} \equiv a^{(p+3)/8}2^{(p-1)/4} \pmod p$.

Example 2.31 The algorithm for computing square roots modulo primes is used in several cryptologic algorithms and, in particular, in the quadratic sieve and number field sieve factoring algorithms that we discuss in Chap. 6. We will see as an example how the Fermat number $F_7 = 2^{2^7} + 1$ is factored with the quadratic sieve, for which purpose the square roots of F_7 modulo p are computed for a set of primes modulo which F_7 is a quadratic residue. One of these primes is:

```
> p:= 31873:
```

Let us compute the square roots of F_7 modulo p using Algorithm 2.8 with the help of Maple, i.e., let us solve the equation $x^2 \equiv 2^{2^7} + 1 \pmod{31873}$. We do it step by step. Observe first that:

```
> F_7 := 2^128+1;
  a := F_7 mod p;
                    340282366920938463463374607431768211457
                                      17919
```

and hence computing these square roots amounts to computing the square roots of 17919 in $\mathbb{Z}^*_{31873}$. Note also that

```
> p mod 4;
                                      1
```

so that we are in the "difficult" case of the algorithm and we will need to execute the `for` loop in it. Next we choose a quadratic non-residue modulo p;

```
> randomize():
   lg := 1:
   while lg = 1 do
     h := (rand(1 .. p-1))():
     lg := numtheory:-legendre(h, p)
   end do:
   h;
                                     30863
```

We compute s and t:

```
> ifactor(p-1);
                                  (2)^7 (3) (83)
> s := 7; t := 3*83;
                                       7
                                     249
```

Now we compute A and H:

```
> A := Power(a, t) mod p;
  H := Power(h, t) mod p;
                                     29298
                                     5250
```

We initialize v:

```
> v := 0:
```

We start the `for` loop with `i := 1`:

```
> i := 1: (A*H^v)^(2^(s-1-i)) mod p;
                                      1
```

The condition $(AH^v)^{2^{s-1-i}} \equiv -1 \pmod{p}$ does not hold, thus the value of v remains equal to 0. Similarly, the analogous computation for $i = 2$ and $i = 3$ gives 1 as result, so the value of v remains 0 during the first three iterations of the loop. In the fourth iteration we have:

```
> i := 4: (A*H^v)^(2^(s-1-i)) mod p;
                                     31872
```

Now we obtained $p - 1 = -1 \bmod p$ as result, so we increase the value of v accordingly:

```
> v := v + 2 ^i;
                                        16
```

Similarly, the value of v gets increased for $i = 5$ and $i = 6$:

```
> i := 5:
  (A*H^v)^(2^(s-1-i)) mod p;
                                      31872
> v := v + 2^i;
                                        48

> i := 6:
  (A*H^v)^(2^(s-1-i)) mod p;
                                      31872
> v := v + 2^i;
                                       112
```

At the end of the loop we obtain the value $v = 112$ and hence we should have that $AH^{112} \equiv 1 \pmod{p}$. We check that this is indeed the case:

```
> A*H^112 mod p;
                                         1
```

We can now find a square root of F_7 modulo p, which is:

```
> x := a^((t+1)*(1/2))*H^56 mod p;
                                      23037
```

Therefore, the other square root is:

```
> y := -x mod p;
                                      8836
```

A quick check that these are indeed the square roots of $a = 17919$ in $\mathbb{Z}^*_{31873}$:

```
> modp~([x, y]^~2, p);
                                  [17919, 17919]
```

The previous example serves to illustrate how the algorithm works step by step but Maple has this algorithm built-in in the function `numtheory:-msqrt`, which computes modular square roots whenever possible, even if the modulus is not prime. For the prime case, this function calls the function `numtheory:-_msqrtprime`, which implements Tonelli's algorithm. This function is local to the module `numtheory` but the reader interested in looking at its code can do so with the commands `kernelopts(opaquemodules=false)` followed by `interface (verboseproc=2)` and `print(numtheory:-_msqrtprime)`. Of course, in what follows we will use the function `numtheory:-msqrt` to compute modular square roots whenever necessary.

The algorithm to compute modular square roots can be extended to non-prime moduli. There is a case of special interest for cryptography which is when the modulus $n = pq$ is the product of two distinct primes p, q. We have already discussed the existence and the number of roots in this case. Based on this discussion, the following exercise is straightforward:

Exercise 2.46 Give an algorithm to solve the equation $x^2 \equiv a \pmod{n}$ when n is the product of two distinct primes and a is a quadratic residue modulo n. (Hint: Use Tonelli's algorithm and the Chinese remainder theorem.)

http://www.springer.com/978-3-642-32166-5

Introduction to Cryptography with Maple

Gómez Pardo, J.L.

2013, XXX, 706 p.,
ISBN: 978-3-642-32166-5