Contents

7. Cryptology

This chapter introduces a number of cryptographic protocols and their underlying problems and algorithms. A typical cryptographic scenario is shown in Figure 7.1 (the design of Alice and Bob is due to Crépeau). Alice and Bob wish to exchange messages over an insecure channel, such as a public telephone line or via email over a computer network. Erich is eavesdropping on the channel. Knowing that their data transfer is being eavesdropped, Alice and Bob encrypt their messages using a cryptosystem.

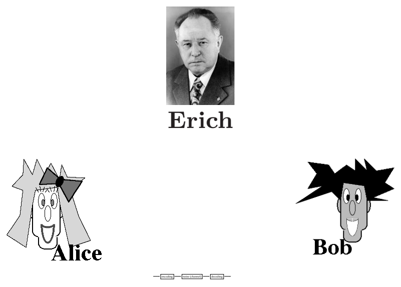

Figure 7.1 A typical scenario in cryptography.

In Section 7.1, various symmetric cryptosystems are presented. A cryptosystem is said to be symmetric if one and the same key is used for encryption and decryption. For symmetric systems, the question of key distribution is central: How can Alice and Bob agree on a joint secret key if they can communicate only via an insecure channel? For example, if Alice chooses some key and encrypts it like a message using a symmetric cryptosystem to send it to Bob, then which key should she use to encrypt this key?

This paradoxical situation is known as ***the secret-key agreement problem,*** and it was considered unsolvable for a long time. Its surprisingly simple, ingenious solution by Whitfield Diffie and Martin Hellman in 1976 is a milestone in the history of cryptography. They proposed a protocol that Alice and Bob can use to exchange a few messages after which they both can easily determine their joint secret key. Eavesdropper Erich, however, does not have clue about their key, even if he was able to intercept every single bit of their transferred messages. Section 7.2 presents the

Diffie-Hellman secret-key agreement protocol.

It may be considered an irony of history that this protocol, which finally solved the long-standing secret-key agreement problem that is so important in symmetric cryptography, opened the door to ***public-key cryptography*** in which there is no need to distribute joint secret keys via insecure channels. In 1978, shortly after Diffie and Hellman had published their pathbreaking work in 1976, Rivest, Shamir, and Adleman developed their famous RSA system, the first ***public-key cryptosystem*** in the open literature. Section 7.3 describes the RSA cryptosystem and the related digital signature scheme. Using the latter protocol, Alice can sign her message to Bob such that he can verify that she indeed is the sender of the message. Digital signatures prevent Erich from forging Alice's messages.

The security of the Diffie-Hellman protocol rests on the assumption that computing discrete logarithms is computationally intractable. That is why modular exponentiation (the inverse function of which is the discrete logarithm) is considered to be a candidate of a one-way function. The security of RSA similarly rests on the assumption that a certain problem is computationally intractable, namely on the assumption that factoring large integers is computationally hard. However, the authorised receiver Bob is able to efficiently decrypt the ciphertext by employing the factorisation of some integer he has chosen in private, which is his private "trapdoor" information.

Section 7.4 introduces a secret-key agreement protocol developed by Rivest and Sherman, which is based on so-called strongly noninvertible associative one-way functions. This protocol can be modified to yield a digital signature scheme as well.

Section 7.5 introduces the fascinating area of interactive proof systems and zero-knowledge protocols that has practical applications in cryptography, especially for authentication issues. In particular, a zero-knowledge protocol for the graph isomorphism problem is presented. On the other hand, this area is also central to complexity theory and will be revisited in Chapter 8, again in connection to the graph isomorphism problem.

7.1. Foundations

Cryptography is the art and science of designing secure cryptosystems, which are used to encrypt texts and messages so that they be kept secret and unauthorised decryption is prevented, whereas the authorised receiver is able to efficiently decrypt the ciphertext received. This section presents two classical symmetric cryptosystems. In subsequent sections, some important asymmetric cryptosystems and cryptographic protocols are introduced. A "protocol" is dialog between two (or more) parties, where a "party" may be either a human being or a computing machine. Cryptographic protocols can have various cryptographic purposes. They consist of algorithms that are jointly executed by more than one party.

Cryptanalysis is the art and science of (unauthorised) decryption of ciphertexts and of breaking existing cryptosystems. ***Cryptology*** captures both these fields, cryptography and cryptanalysis. In this chapter, we focus on ***cryptographic algorithms.*** Algorithms of cryptanalysis, which are used to break cryptographic protocols and systems, will be mentioned as well but will not be investigated in detail.

7.1.1. Cryptography

Figure 7.1 shows a typical scenario in cryptography: Alice and Bob communicate over an insecure channel that is eavesdropped by Erich, and thus they encrypt their messages using a cryptosystem.

Definition 7.1 (Cryptosystem). *A **cryptosystem** is a quintuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ with the following properties:*

1. *$\mathcal{P}$, $\mathcal{C}$, and $\mathcal{K}$ are finite sets, where $\mathcal{P}$ is the **plaintext space,** $\mathcal{C}$ is the **ciphertext space,** and $\mathcal{K}$ is the **key space.** The elements of $\mathcal{P}$ are called the **plaintexts,** and the elements of $\mathcal{C}$ are called the **ciphertexts.** A **message** is a string of plaintext symbols.*
2. *$\mathcal{E} = \{E_k \mid k \in \mathcal{K}\}$ is a family of functions $E_k : \mathcal{P} \to \mathcal{C}$, which are used for encryption. $\mathcal{D} = \{D_k \mid k \in \mathcal{K}\}$ is a family of functions $D_k : \mathcal{C} \to \mathcal{P}$, which are used for decryption.*
3. *For each key $e \in \mathcal{K}$, there exists some key $d \in \mathcal{K}$ such that for each plaintext $p \in \mathcal{P}$,*
$$D_d(E_e(p)) = p \ . \tag{7.1}$$

*A cryptosystem is said to be **symmetric** (or private-key) if either $d = e$ or if d at least can "easily" be determined from e. A cryptosystem is said to be **asymmetric** (or public-key) if $d \neq e$ and it is "computationally infeasible" to determine the private key d from the corresponding public key e.*

At times, we may use distinct key spaces for encryption and decryption, with the above definition modified accordingly.

We now introduce some easy examples of classical symmetric cryptosystems. Consider the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$, which will be used both for the plaintext space and for the ciphertext space. We identify Σ with $\mathbb{Z}_{26} = \{0, 1, \ldots, 25\}$ so as to be able to perform calculations with letters as if they were numbers. The number 0 corresponds to the letter A, the 1 corresponds to B, and so on. This coding of plaintext or ciphertext symbols by nonnegative integers is not part of the actual encryption or decryption.

Messages are elements of Σ^*, where Σ^* denotes the set of strings over Σ. If some message $m \in \Sigma^*$ is subdivided into blocks of length n and is encrypted blockwise, as it is common in many cryptosystems, then each block of m is viewed as an element of $\mathbb{Z}_{26}^n$.

Example 7.1 (Shift Cipher) The first example is a monoalphabetic symmetric cryptosystem. Let $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbb{Z}_{26}$. The ***shift cipher*** encrypts messages by shifting every plaintext symbol by the same number k of letters in the alphabet modulo 26. Shifting each letter in the ciphertext back using the same key k, the original plaintext is recovered. For each key $k \in \mathbb{Z}_{26}$, the encryption function E_k and the decryption function D_k are defined by:

$$\begin{aligned} E_k(m) &= (m + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26 \ , \end{aligned}$$

where addition and subtraction by k modulo 26 are carried out characterwise.

m	S H I F T E A C H L E T T E R T O T H E L E F T
c	R G H E S D Z B G K D S S D Q S N S G D K D E S

Figure 7.2 Example of an encryption by the shift cipher.

Figure 7.2 shows an encryption of the message m by the shift cipher with key $k = 25$. The resulting ciphertext is c. Note that the particular shift cipher with key $k = 3$ is also known as the ***Caesar cipher,*** since the Roman Emperor allegedly used this cipher during his wars to keep messages secret.[1] This cipher is a very simple substitution cipher in which each letter is substituted by a certain letter of the alphabet.

Since the key space is very small, the shift cipher can easily be broken. It is already vulnerable by attacks in which the attacker knows the ciphertext only, simply by checking which of the 26 possible keys reveals a meaningful plaintext, provided that the ciphertext is long enough to allow unique decryption.

The shift cipher is a monoalphabetic cryptosystem, since every plaintext letter is replaced by one and the same letter in the ciphertext. In contrast, a polyalphabetic cryptosystem can encrypt the same plaintext symbols by different ciphertext symbols, depending on their position in the text. Such a polyalphabetic cryptosystem that is based on the shift cipher, yet much harder to break, was proposed by the French diplomat Blaise de Vigenère (1523 until 1596). His system builds on previous work by the Italian mathematician Leon Battista Alberti (born in 1404), the German abbot Johannes Trithemius (born in 1492), and the Italian scientist Giovanni Porta (born in 1675). It works like the shift cipher, except that the letter that encrypts some plaintext letter varies with its position in the text.

Example 7.2[Vigenère Cipher] This symmetric polyalphabetic cryptosystem uses a so-called ***Vigenère square,*** a matrix consisting of 26 rows and 26 columns, see Figure 7.3. Every row has the 26 letters of the alphabet, shifted from row to row by one position. That is, the single rows can be seen as a shift cipher obtained by the keys $0, 1, \ldots, 25$. Which row of the Vigenère square is used for encryption of some plaintext symbol depends on its position in the text.

Messages are subdivided into blocks of a fixed length n and are encrypted blockwise, i.e., $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbb{Z}_{26}^n$. The block length n is also called the *period* of the system. In what follows, the ith symbol in a string w is denoted by w_i.

For each key $k \in \mathbb{Z}_{26}^n$, the encryption function E_k and the decryption function D_k, both mapping from $\mathbb{Z}_{26}^n$ to $\mathbb{Z}_{26}^n$, are defined by:

$$\begin{aligned} E_k(b) &= (b + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26, \end{aligned}$$

where addition and subtraction by k modulo 26 are again carried out characterwise. That is, the key $k \in \mathbb{Z}_{26}^n$ is written letter by letter above the symbols of the block $b \in \mathbb{Z}_{26}^n$ to be encrypted. If the last plaintext block has less than n symbols, one uses less key symbols

[1] Historic remark: Gaius Julius Caesar reports in his book *De Bello Gallico* that he sent an encrypted message to Q. Tullius Cicero (the brother of the famous speaker) during the Gallic Wars (58 until 50 B.C.). The system used was monoalphabetic and replaced Latin letters by Greek letters; however, it is not explicitly mentioned there if the cipher used indeed was the shift cipher with key $k = 3$. This information was given later by Suetonius.

0	A	B	C	D	E	F	G	$\boxed{\textbf{H}}$	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	$\boxed{\textbf{T}}$	U	V	W	X	Y	Z	$\boxed{\textbf{A}}$	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figure 7.3 Vigenère square: Plaintext "H" is encrypted as "A" by key "T".

k	T O N Y T O N Y T O N Y T O N Y T O N Y T O N Y T O N Y
m	H U N G A R I A N I S A L L G R E E K T O G E R M A N S
c	A I A E T F V Y G W F Y E Z T P X S X R H U R P F O A Q

Figure 7.4 Example of an encryption by the Vigenère cipher.

accordingly. In order to encrypt the ith plaintext symbol b_i, which has the key symbol k_i sitting on top, use the ith row of the Vigenère square as in the shift cipher.

For example, choose the block length $n = 4$ and the key $k =$ TONY. Figure 7.4 shows the encryption of the message m, which consists of seven blocks, into the ciphertext c by the Vigenère cipher using key k.

To the first plaintext letter, "H", the key symbol "T" is assigned. The intersection of the "H" column with the "T" row of the Vigenère square yields "A" as the first letter of the ciphertext, see Figure 7.3.

There are many other classical cryptosystems, which will not be described in detail here. There are various ways to classify cryptosystems according to their properties or to the specific way they are designed. In Definition 7.1, the distinction between *symmetric* and *asymmetric* cryptosystems was explained. The two examples above (the shift cipher and the Vigenère cipher) demonstrated the distinction between *monoalphabetic* and *polyalphabetic* systems. Both are ***substitution ciphers,*** which may be contrasted with ***permutation ciphers*** (a.k.a. ***transposition ciphers***) in which the plaintext letters are not substituted by certain ciphertext letters but go to another position in the text remaining otherwise unchanged.

Moreover, ***block ciphers*** such as the Vigenère system can be contrasted with ***stream ciphers,*** which produce a continuous stream of key symbols depending on the plaintext context to be encrypted. One can also distinguish between different types of block ciphers. An important type are the ***affine linear block ciphers,*** which are defined by affine linear encryption functions $E_{(A,\vec{b})}$ and decryption functions $D_{(A^{-1},\vec{b})}$, both mapping from $\mathbb{Z}_m^n$ to $\mathbb{Z}_m^n$. That is, they are of the following form:

$$\begin{aligned} E_{(A,\vec{b})}(\vec{x}) &= A\vec{x} + \vec{b} \bmod m \,, \\ D_{(A^{-1},\vec{b})}(\vec{y}) &= A^{-1}(\vec{y} - \vec{b}) \bmod m \,. \end{aligned} \tag{7.2}$$

Here, $(A, \vec{b})$ and $(A^{-1}, \vec{b})$ are the keys used for encryption and decryption, respectively; A is a $(n \times n)$ matrix with entries from $\mathbb{Z}_m$; A^{-1} is the inverse matrix for A; $\vec{x}$, $\vec{y}$, and $\vec{b}$ are vectors in $\mathbb{Z}_m^n$, and all arithmetics is carried out modulo m. Some mathematical explanations are in order (see also Definition 7.2 in Subsection 7.1.3): An $(n \times n)$ matrix A over the ring $\mathbb{Z}_m$ has a multiplicative inverse if and only if $\gcd(\det A, m) = 1$. The *inverse matrix for A* is defined by $A^{-1} = (\det A)^{-1} A_{\mathtt{adj}}$, where $\det A$ is the determinant of A, and $A_{\mathtt{adj}} = ((-1)^{i+j} \det A_{j,i})$ is the *adjunct matrix for A*. The *determinant* $\det A$ of A is recursively defined: For $n = 1$ and $A = (a)$, $\det A = a$; for $n > 1$ and each $i \in \{1, 2, \ldots, n\}$, $\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \det A_{i,j}$, where $a_{i,j}$ denotes the (i, j) entry of A and the $(n-1) \times (n-1)$ matrix $A_{i,j}$ results from A by cancelling the ith row and the jth column. The determinant of a matrix and thus its inverse (if it exists) can be computed efficiently, see Problem 7-3.

For example, the Vigenère cipher is an affine linear cipher whose key contains the unity matrix as its first component. If $\vec{b}$ in (7.2) is the zero vector, then it is a *linear block cipher*. A classical example is the ***Hill cipher,*** invented by Lester Hill in 1929. Here, the key space is the set of all $(n \times n)$ matrices A with entries in $\mathbb{Z}_m$ such that $\gcd(\det A, m) = 1$. This condition guarantees the invertibility of those matrices that are allowed as keys, since the inverse matrix A^{-1} is used for decryption of the messages encrypted by key A. For each key A, the Hill cipher is defined by the encryption function $E_A(\vec{x}) = A\vec{x} \bmod m$ and the decryption function $D_{A^{-1}}(\vec{y}) = A^{-1}\vec{y} \bmod m$. Thus, it is the most general linear cipher. The permutation cipher also is linear, and hence is a special case of the Hill cipher.

7.1.2. Cryptanalysis

Cryptanalysis aims at breaking existing cryptosystems and, in particular, at determining the decryption keys. In order to characterise the security or vulnerability of the cryptosystem considered, one distinguishes different types of attacks according to the information available for the attacker. For the shift cipher, ***ciphertext-only*** attacks were already mentioned. They are the weakest type of attack, and a cryptosystem that does not resist such attacks is not of much value.

Affine linear block ciphers such as the Vigenère and the Hill cipher are vulnerable to attacks in which the attacker knows the plaintext corresponding to some ciphertext obtained and is able to conclude the keys used. These attacks are called ***known-plaintext attacks.*** Affine linear block ciphers are even more vulnerable to ***chosen-plaintext attacks,*** in which the attacker can choose some plaintext and is then able to see which ciphertext corresponds to the plaintext chosen. Another type of attack is in particular relevant for asymmetric cryptosystems: In an ***encryption-key attack,*** the attacker merely knows the public key but does not know any ciphertext yet, and seeks to determine the private key from this information. The difference is that the attacker now has plenty of time to perform computations, whereas in the other types of attacks the ciphertext was already sent and much less time is available to decrypt it. That is why keys of much larger size are required in public-key cryptography to guarantee the security of the system used. Hence, asymmetric cryptosystems are much less efficient than symmetric cryptosystems in many practical applications.

For the attacks mentioned above, the method of frequency counts is often useful. This method exploits the redundancy of the natural language used. For example, in many natural languages, the letter “E” occurs statistically significant most frequently. On average, the “E” occurs in long, “typical” texts with a percentage of 12.31% in English, of 15.87% in French, and even of 18.46% in German. In other languages, different letters may occur most frequently. For example, the “A” is the most frequent letter in long, “typical” Finnish texts, with a percentage of 12.06%.

That the method of frequency counts is useful for attacks on monoalphabetic cryptosystems is obvious. For example, if in a ciphertext encrypting a long German text by the shift cipher, the letter occurring most frequently is “Y”, which is rather rare in German (as well as in many other languages), then it is most likely that “Y” encrypts “E”. Thus, the key used for encryption is “U” ($k = 20$). In addition to counting the frequency of single letters, one can also count the frequency with which certain pairs of letters (so-called digrams) and triples of letters (so-called trigrams) occur, and so on. This kind of attack also works for polyalphabetic systems, provided the period (i.e., the block length) is known.

Polyalphabetic cryptosystems with an unknown period, however, provide more security. For example, the Vigenère cipher resisted each attempt of breaking it for a long time. No earlier than in 1863, about 300 years after its discovery, the German cryptanalyst Friedrich Wilhelm Kasiski found a method of breaking the Vigenère cipher. He showed how to determine the period, which initially is unknown, from repetitions of the same substring in the ciphertext. Subsequently, the ciphertext can be decrypted by means of frequency counts. Singh writes that the British eccentric

Charles Babbage, who was considered a genius of his time by many, presumably had discovered Kasiski's method even earlier, around 1854, although he didn't publish his work.

The pathbreaking work of Claude Shannon (1916 until 2001), the father of modern coding and information theory, is now considered a milestone in the history of cryptography. Shannon proved that there exist cryptosystems that guarantee perfect secrecy in a mathematically rigorous sense. More precisely, a cryptosystem $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ***guarantees perfect secrecy*** if and only if $|\mathcal{P}| = |\mathcal{C}| = |\mathcal{K}|$, the keys in $\mathcal{K}$ are uniformly distributed, and for each plaintext $p \in \mathcal{P}$ and for each ciphertext $c \in \mathcal{C}$ there exists *exactly one* key $k \in \mathcal{K}$ with $E_k(p) = c$. That means that such a cryptosystem often is not useful for practical applications, since in order to guarantee perfect secrecy, every key must be at least as long as the message to be encrypted and can be used only once.

7.1.3. Algebra, number theory, and graph theory

In order to understand some of the algorithms and problems to be presented later, some fundamental notions, definitions, and results from algebra and, in particular, from number theory, group theory, and graph theory are required. This concerns both the cryptosystems and zero-knowledge protocols in Chapter 7 and some of the problems to be considered in upcoming Chapter 8. The present subsection may as well be skipped for now and revisited later when the required notions and results come up. In this section, most of the proofs are omitted.

Definition 7.2 (Group, ring, and field).

- *A* ***group*** $\mathfrak{G} = (S, \circ)$ *is defined by some nonempty set S and a two-ary operation $\circ$ on S that satisfy the following axioms:*

  - Closure: $(\forall x \in S)\,(\forall y \in S)\,[x \circ y \in S]$.
  - Associativity: $(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S)\,[(x \circ y) \circ z = x \circ (y \circ z)]$.
  - Neutral element: $(\exists e \in S)\,(\forall x \in S)\,[e \circ x = x \circ e = x]$.
  - Inverse element: $(\forall x \in S)\,(\exists x^{-1} \in S)\,[x \circ x^{-1} = x^{-1} \circ x = e]$.

  The element e is called the ***neutral element of the group $\mathfrak{G}$****. The element x^{-1} is called the* ***inverse element for x.*** *$\mathfrak{G}$ is said to be a* ***monoid*** *if $\mathfrak{G}$ satisfies associativity and closure under $\circ$, even if $\mathfrak{G}$ has no neutral element or if not every element in $\mathfrak{G}$ has an inverse. A group $\mathfrak{G} = (S, \circ)$ (respectively, a monoid $\mathfrak{G} = (S, \circ)$) is said to be* ***commutative*** *(or abelian) if and only if $x \circ y = y \circ x$ for all $x, y \in S$. The number of elements of a finite group $\mathfrak{G}$ is said to be* ***the order $\mathfrak{G}$*** *and is denoted by $|\mathfrak{G}|$.*

- *$\mathfrak{H} = (T, \circ)$ is said to be a* ***subgroup of a group $\mathfrak{G} = (S, \circ)$*** *(denoted by $\mathfrak{H} \leq \mathfrak{G}$) if and only if $T \subseteq S$ and $\mathfrak{H}$ satisfies the group axioms.*

- *A* ***ring*** *is a triple $\mathfrak{R} = (S, +, \cdot)$ such that $(S, +)$ is an abelian group and $(S, \cdot)$ is a monoid and the distributive laws are satisfied:*

$$(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S) \quad :$$

$$(x \cdot (y + z) = (x \cdot y) + (x \cdot z)) \wedge ((x + y) \cdot z = (x \cdot z) + (y \cdot z)) .$$

*A ring $\mathfrak{R} = (S, +, \cdot)$ is said to be **commutative** if and only if the monoid $(S, \cdot)$ is commutative. The neutral element group $(S, +)$ is called the **zero element** (the zero, for short) of the ring $\mathfrak{R}$. A neutral element of the monoid $(S, \cdot)$ is called the **one element** (the one, for short) of the ring $\mathfrak{R}$.*

- *Let $\mathfrak{R} = (S, +, \cdot)$ be a ring with one. An element x of $\mathfrak{R}$ is said to be **invertible** (or a unity of $\mathfrak{R}$) if and only if it is invertible in the monoid $(S, \cdot)$.*
- *A **field** is a commutative ring with one in which each nonzero element is invertible.*

Example 7.3[Group, ring, and field]

- Let $k \in \mathbb{N}$. The set $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$ is a finite group with respect to addition modulo k, with neutral element 0. With respect to addition and multiplication modulo k, $\mathbb{Z}_k$ is a commutative ring with one, see Problem 7-1. If p is a *prime number*, then $\mathbb{Z}_p$ is a field with respect to addition and multiplication modulo p.
- Let $\gcd n, m$ denote the greatest common divisor of two integers m and n. For $k \in \mathbb{N}$, define the set $\mathbb{Z}_k^* = \{i \mid 1 \leq i \leq k-1 \text{ and } \gcd i, k = 1\}$. With respect to multiplication modulo k, $\mathbb{Z}_k^*$ is a finite group with neutral element 1.

If the operation $\circ$ of a group $\mathfrak{G} = (S, \circ)$ is clear from the context, we omit stating it explicitly. The group $\mathbb{Z}_k^*$ from Example 7.3 will play a particular role in Section 7.3, where the RSA cryptosystem is introduced. The *Euler function* φ gives the order of this group, i.e., $\varphi(k) = |\mathbb{Z}_k^*|$. The following properties of φ follow from the definition:

- $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ for all $m, n \in \mathbb{N}$ with $\gcd m, n = 1$, and
- $\varphi(p) = p - 1$ for all prime numbers p.

The proof of these properties is left to the reader as Exercise 7.1-3. In particular, we will apply the following fact in Subsection 7.3.1, which is a consequence of the properties above.

Claim 7.3 *If $n = p \cdot q$ for prime numbers p and q, then $\varphi(n) = (p-1)(q-1)$.*

Euler's Theorem below is a special case (namely, for the group $\mathbb{Z}_n^*$) of Lagrange's Theorem, which says that for each group element a of a finite multiplicative group $\mathfrak{G}$ of order $|\mathfrak{G}|$ and with neutral element e, $a^{|\mathfrak{G}|} = e$. The special case of Euler's theorem, where n is a prime number not dividing a, is known as Fermat's Little Theorem.

Theorem 7.4 (Euler). *For each $a \in \mathbb{Z}_n^*$, $a^{\varphi(n)} \equiv 1 \bmod n$.*

Corollary 7.5 (Fermat's Little Theorem). *If p is a prime number and $a \in \mathbb{Z}_p^*$, then $a^{p-1} \equiv 1 \bmod p$.*

In Section 8.4, algorithms for the graph isomorphism problem will be presented. This problem, which also is related to the zero-knowledge protocols to be introduced in Subsection 7.5.2, can be seen as a special case of certain group-theoretic problems. In particular, *permutation groups* are of interest here. Some examples for illustration will be presented later.

Definition 7.6 (Permutation group).

- *A* ***permutation*** *is a bijective mapping of a set onto itself. For each integer* $n \geq 1$, *let* $[n] = \{1, 2, \ldots, n\}$. *The set of all permutations of* $[n]$ *is denoted by* $\mathfrak{S}_n$. *For algorithmic purposes, permutations* $\pi \in \mathfrak{S}_n$ *are given as pairs* $(i, \pi(i))$ *from* $[n] \times [n]$.
- *If one defines the composition of permutations as an operation on* $\mathfrak{S}_n$, *then* $\mathfrak{S}_n$ *becomes a group. For two permutations* π *and* τ *in* $\mathfrak{S}_n$, *their* composition $\pi\tau$ *is defined to be that permutation in* $\mathfrak{S}_n$ *that results from first applying* π *and then* τ *to the elements of* $[n]$, *i.e.,* $(\pi\tau)(i) = \tau(\pi(i))$ *for each* $i \in [n]$. *The neutral element of the permutation group* $\mathfrak{S}_n$ *is the* ***identical permutation,*** *which is defined by* $\mathrm{id}(i) = i$ *for each* $i \in [n]$. *The subgroup of* $\mathfrak{S}_n$ *that contains* id *as its only element is denoted by* $\mathbf{id}$.
- *For any subset* $\mathfrak{T}$ *of* $\mathfrak{S}_n$, *the* ***permutation group*** $\langle\mathfrak{T}\rangle$ ***generated by*** $\mathfrak{T}$ *is defined as the smallest subgroup of* $\mathfrak{S}_n$ *containing* $\mathfrak{T}$. *Subgroups* $\mathfrak{G}$ *of* $\mathfrak{S}_n$ *are represented by their generating sets, sometimes dubbed the generators of* $\mathfrak{G}$. *In* $\mathfrak{G}$, *the* ***orbit of an element*** $i \in [n]$ *is defined as* $\mathfrak{G}(i) = \{\pi(i) \mid \pi \in \mathfrak{G}\}$.
- *For any subset* T *of* $[n]$, *let* $\mathfrak{S}_n^T$ *denote the subgroup of* $\mathfrak{S}_n$ *that maps every element of* T *onto itself. In particular, for* $i \leq n$ *and a subgroup* $\mathfrak{G}$ *of* $\mathfrak{S}_n$, *the* ***(pointwise) stabiliser of*** $[i]$ ***in*** $\mathfrak{G}$ *is defined by*

  $$\mathfrak{G}^{(i)} = \{\pi \in \mathfrak{G} \mid \pi(j) = j \text{ for each } j \in [i]\} .$$

  Observe that $\mathfrak{G}^{(n)} = \mathbf{id}$ *and* $\mathfrak{G}^{(0)} = \mathfrak{G}$.
- *Let* $\mathfrak{G}$ *and* $\mathfrak{H}$ *be permutation groups with* $\mathfrak{H} \leq \mathfrak{G}$. *For* $\tau \in \mathfrak{G}$, $\mathfrak{H}\tau = \{\pi\tau \mid \pi \in \mathfrak{H}\}$ *is said to be a* ***right coset of*** $\mathfrak{H}$ ***in*** $\mathfrak{G}$. *Any two right cosets of* $\mathfrak{H}$ *in* $\mathfrak{G}$ *are either identical or disjoint. Thus, the permutation group* $\mathfrak{G}$ *is partitioned by the right cosets of* $\mathfrak{H}$ *in* $\mathfrak{G}$:

  $$\mathfrak{G} = \mathfrak{H}\tau_1 \cup \mathfrak{H}\tau_2 \cup \cdots \cup \mathfrak{H}\tau_k \ . \tag{7.3}$$

  Every right coset of $\mathfrak{H}$ *in* $\mathfrak{G}$ *has the cardinality* $|\mathfrak{H}|$. *The set* $\{\tau_1, \tau_2, \ldots, \tau_k\}$ *in (7.3) is called the* ***complete right transversal of*** $\mathfrak{H}$ ***in*** $\mathfrak{G}$.

The notion of pointwise stabilisers is especially important for the design of algorithms solving problems on permutation groups. The crucial structure exploited there is the so-called ***tower of stabilisers*** of a permutation group $\mathfrak{G}$:

$$\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G} \ .$$

For each i with $1 \leq i \leq n$, let $\mathfrak{T}_i$ be the complete right transversal of $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$. Then, $\mathfrak{T} = \bigcup_{i=1}^{n-1} \mathfrak{T}_i$ is said to be a ***strong generator of*** $\mathfrak{G}$, and we have $\mathfrak{G} = \langle\mathfrak{T}\rangle$. Every $\pi \in \mathfrak{G}$ then has a unique factorisation $\pi = \tau_1\tau_2\cdots\tau_n$ with $\tau_i \in \mathfrak{T}_i$. The following basic algorithmic results about permutation groups will be useful later in Section 8.4.

Theorem 7.7 *Let a permutation group $\mathfrak{G} \leq \mathfrak{S}_n$ be given by a generator. Then, we have:*

1. *For each $i \in [n]$, the orbit $\mathfrak{G}(i)$ of i in $\mathfrak{G}$ can be computed in polynomial time.*
2. *The tower of stabilisers* $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ *can be computed in time polynomially in n, i.e., for each i with $1 \leq i \leq n$, the complete right transversals $\mathfrak{T}_i$ of $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ and thus a strong generator of $\mathfrak{G}$ can be determined efficiently.*

The notions introduced in Definition 7.6 for permutation groups are now explained for concrete examples from graph theory. In particular, we consider the automorphism group and the set of isomorphisms between graphs. We start by introducing some useful graph-theoretical concepts.

Definition 7.8 (Graph isomorphism and graph automorphism). *A graph G consists of a finite set of vertices, $V(G)$, and a finite set of edges, $E(G)$, that connect certain vertices with each other. We assume that no multiple edges and no loops occur. In this chapter, we consider only undirected graphs, i.e., the edges are not oriented and can be seen as unordered vertex pairs. The* disjoint union $G \cup H$ of two graphs G and H *is defined as the graph with vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$, where we assume that that the vertex sets $V(G)$ and $V(H)$ are made disjoint (by renaming if necessary).*

Let G and H be two graphs with the same number of vertices. An ***isomorphism between G and H*** *is an edge-preserving bijection of the vertex set of G onto that of H. Under the convention that $V(G) = \{1, 2, \ldots, n\} = V(H)$, G and H are isomorphic ($G \cong H$, for short) if and only if there exists a permutation $\pi \in \mathfrak{S}_n$ such that for all vertices $i, j \in V(G)$,*

$$\{i, j\} \in E(G) \iff \{\pi(i), \pi(j)\} \in E(H) . \tag{7.4}$$

An ***automorphism of G*** *is an edge-preserving bijection of the vertex set of G onto itself. Every graph has the trivial automorphism* id. *By* Iso(G, H) *we denote the set of all isomorphisms between G and H, and by* Aut(G) *we denote the set of all automorphisms of G. Define the problems* ***graph automorphism*** (`GI`*, for short) and* ***graph automorphism*** (`GA`*, for short) by:*

$$\begin{aligned} \mathtt{GI} &= \{(G, H) \mid G \text{ and } H \text{ are isomorphic graphs}\} ; \\ \mathtt{GA} &= \{G \mid G \text{ has a nontrivial automorphism}\} . \end{aligned}$$

For algorithmic purposes, graphs are represented either by their vertex and edge lists or by an adjacency matrix, which has the entry 1 at position (i, j) if $\{i, j\}$ is an edge, and the entry 0 otherwise. This graph representation is suitably encoded over the alphabet $\Sigma = \{0, 1\}$. In order to represent pairs of graphs, we use a standard bijective pairing function $(\cdot, \cdot)$ from $\Sigma^* \times \Sigma^*$ onto Σ^* that is polynomial-time computable and has polynomial-time computable inverses.

Example 7.4[Graph isomorphism and graph automorphism] The graphs G and H shown in Figure 7.5 are isomorphic.

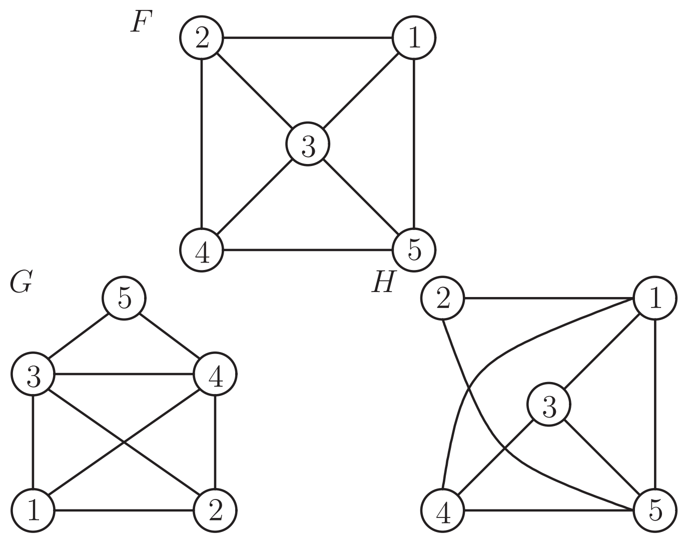

Figure 7.5 Three graphs: G is isomorphic to H, but not to F.

An isomorphism $\pi : V(G) \to V(H)$ preserving adjacency of the vertices according to (7.4) is given, e.g., by $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{smallmatrix}\right)$, or in cycle notation by $\pi = (1\,3)(2\,4\,5)$. There are three further isomorphisms between G and H, i.e., $|\mathrm{Iso}(G,H)| = 4$, see Exercise 7.1-4. However, neither G nor H is isomorphic to F. This is immediately seen from the fact that the sequence of *vertex degrees* (the number of edges fanning out of each vertex) of G and H, respectively, differs from the sequence of vertex degrees of F: For G and H, this sequence is $(2,3,3,4,4)$, whereas it is $(3,3,3,3,4)$ for F. A nontrivial automorphism $\varphi : V(G) \to V(G)$ of G is given, e.g., by $\varphi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 5 \end{smallmatrix}\right)$, or $\varphi = (1\,2)(3\,4)(5)$; another one by $\tau = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 4 & 3 & 5 \end{smallmatrix}\right)$, or $\tau = (1)(2)(3\,4)(5)$. There are two more automorphisms of G, i.e., $|\mathrm{Aut}(G)| = 4$, see Exercise 7.1-4.

The permutation groups $\mathrm{Aut}(F)$, $\mathrm{Aut}(G)$, and $\mathrm{Aut}(H)$ are subgroups of $\mathfrak{S}_5$. The tower $\mathrm{Aut}(G)^{(5)} \leq \mathrm{Aut}(G)^{(4)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)}$ of stabilisers of $\mathrm{Aut}(G)$ consists of the subgroups $\mathrm{Aut}(G)^{(5)} = \mathrm{Aut}(G)^{(4)} = \mathrm{Aut}(G)^{(3)} = \mathbf{id}$, $\mathrm{Aut}(G)^{(2)} = \mathrm{Aut}(G)^{(1)} = \langle\{\mathrm{id}, \tau\}\rangle$, and $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. In the automorphism group $\mathrm{Aut}(G)$ of G, the vertices 1 and 2 have the orbit $\{1,2\}$, the vertices 3 and 4 have the orbit $\{3,4\}$, and vertex 5 has the orbit $\{5\}$.

$\mathrm{Iso}(G,H)$ and $\mathrm{Aut}(G)$ have the same number of elements if and only if G and H are isomorphic. To wit, if G and H are isomorphic, then $|\mathrm{Iso}(G,H)| = |\mathrm{Aut}(G)|$ follows from $\mathrm{Aut}(G) = \mathrm{Iso}(G,G)$. Otherwise, if $G \not\cong H$, then $\mathrm{Iso}(G,H)$ is empty but $\mathrm{Aut}(G)$ contains always the trivial automorphism id. This implies (7.5) in Lemma 7.9 below, which we will need later in Section 8.4. For proving (7.6), suppose that G and H are connected; otherwise, we simply consider instead of G and H the co-graphs $\overline{G}$ and $\overline{H}$, see Exercise 7.1-5. An automorphism of $G \cup H$ that switches the vertices of G and H, consists of an isomorphism in $\mathrm{Iso}(G,H)$ and an isomorphism in $\mathrm{Iso}(H,G)$. Thus, $|\mathrm{Aut}(G \cup H)| = |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| + |\mathrm{Iso}(G,H)|^2$, which implies (7.6) via (7.5).

Lemma 7.9 *For any two graphs G and H, we have*

$$|\mathrm{Iso}(G,H)| = \begin{cases} |\mathrm{Aut}(G)| = |\mathrm{Aut}(H)| & \text{if } G \cong H \\ 0 & \text{if } G \ncong H \end{cases} ; \tag{7.5}$$

$$|\mathrm{Aut}(G \cup H)| = \begin{cases} 2 \cdot |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \text{if } G \cong H \\ |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \text{if } G \ncong H \end{cases} . \tag{7.6}$$

If G and H are isomorphic graphs and if $\tau \in \mathrm{Iso}(G,H)$, then $\mathrm{Iso}(G,H) = \mathrm{Aut}(G)\tau$. Thus, $\mathrm{Iso}(G,H)$ is a right coset of $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Since any two right cosets are either identical or disjoint, $\mathfrak{S}_n$ can be partitioned into right cosets of $\mathrm{Aut}(G)$ according to (7.3):

$$\mathfrak{S}_n = \mathrm{Aut}(G)\tau_1 \cup \mathrm{Aut}(G)\tau_2 \cup \cdots \cup \mathrm{Aut}(G)\tau_k , \tag{7.7}$$

where $|\mathrm{Aut}(G)\tau_i| = |\mathrm{Aut}(G)|$ for all i, $1 \leq i \leq k$. The set $\{\tau_1, \tau_2, \ldots, \tau_k\}$ of permutations in $\mathfrak{S}_n$ thus is a complete right transversal of $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Denoting by $\pi(G)$ the graph $H \cong G$ that results from applying the permutation $\pi \in \mathfrak{S}_n$ to the vertices of G, we have $\{\tau_i(G) \mid 1 \leq i \leq k\} = \{H \mid H \cong G\}$. Since there are exactly $n! = n(n-1)\cdots 2 \cdot 1$ permutations in $\mathfrak{S}_n$, it follows from (7.7) that

$$|\{H \mid H \cong G\}| = k = \frac{|\mathfrak{S}_n|}{|\mathrm{Aut}(G)|} = \frac{n!}{|\mathrm{Aut}(G)|} .$$

This proves the following corollary.

Corollary 7.10 *If G is a graph with n vertices, then there are exactly $n!/|\mathrm{Aut}(G)|$ graphs isomorphic to G.*

For the graph G in Figure 7.5 from Example 7.4, there thus exist exactly $5!/4 = 30$ isomorphic graphs. The following lemma will be used later in Section 8.4.

Lemma 7.11 *Let G and H be two graphs with n vertices. Define the set*

$$A(G,H) = \{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\} .$$

Then, we have

$$|A(G,H)| = \begin{cases} n! & \text{if } G \cong H \\ 2n! & \text{if } G \ncong H \end{cases} .$$

Proof If F and G are isomorphic, then $|\mathrm{Aut}(F)| = |\mathrm{Aut}(G)|$. Hence, by Corollary 7.10, we have

$$|\{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\}| = \frac{n!}{|\mathrm{Aut}(F)|} \cdot |\mathrm{Aut}(F)| = n! .$$

Analogously, one can show that $|\{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\}| = n!$. If G and H are isomorphic, then

$$\{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\} = \{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\} .$$

c_1	W K L V V H Q W H Q F H L V H Q F U B S W H G E B F D H V D U V N H B
c_2	N U O S J Y A Z E E W R O S V H P X Y G N R J B P W N K S R L F Q E P

Figure 7.6 Two ciphertexts encrypting the same plaintext, see Exercise 7.1-1.

It follows that $|A(G,H)| = n!$. If G and H are nonisomorphic, then $\{(F,\varphi) \mid F \cong G$ and $\varphi \in \mathrm{Aut}(F)\}$ and $\{(F,\varphi) \mid F \cong H$ and $\varphi \in \mathrm{Aut}(F)\}$ are disjoint sets. Thus, $|A(G,H)| = 2n!$. ■

Exercises

7.1-1 Figure 7.6 shows two ciphertexts, c_1 and c_2. It is known that both encrypt the same plaintext m and that one was obtained using the shift cipher, the other one using the Vigenère cipher. Decrypt both ciphertexts. *Hint.* After decryption you will notice that the plaintext obtained is a true statement for one of the two ciphertexts, whereas it is a false statement for the other ciphertext. Is perhaps the method of frequency counts useful here?

7.1-2 Prove that $\mathbb{Z}$ is a ring with respect to ordinary addition and multiplication. Is it also a field? What can be said about the properties of the algebraic structures $(\mathbb{N},+)$, $(\mathbb{N},\cdot)$, and $(\mathbb{N},+,\cdot)$?

7.1-3 Prove the properties stated for Euler's φ function:

a. $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ for all $m, n \in \mathbb{N}$ with $\gcd(m,n) = 1$.

b. $\varphi(p) = p - 1$ for all prime numbers p.

Using these properties, prove Proposition 7.3.

7.1-4 Consider the graphs F, G, and H from Figure 7.5 in Example 7.4.

a. Determine all isomorphisms between G and H.

b. Determine all automorphisms of F, G, and H.

c. For which isomorphisms between G and H is $\mathrm{Iso}(G,H)$ a right coset of $\mathrm{Aut}(G)$ in $\mathfrak{S}_5$, i.e., for which $\tau \in \mathrm{Iso}(G,H)$ is $\mathrm{Iso}(G,H) = \mathrm{Aut}(G)\tau$? Determine the complete right transversals of $\mathrm{Aut}(F)$, $\mathrm{Aut}(G)$, and $\mathrm{Aut}(H)$ in $\mathfrak{S}_5$.

d. Determine the orbit of all vertices of F in $\mathrm{Aut}(F)$ and the orbit of all vertices of H in $\mathrm{Aut}(H)$.

e. Determine the tower of stabilisers of the subgroups $\mathrm{Aut}(F) \leq \mathfrak{S}_5$ and $\mathrm{Aut}(H) \leq \mathfrak{S}_5$.

f. How many graphs with 5 vertices are isomorphic to F?

7.1-5 The *co-graph* $\overline{G}$ of a graph G is defined by the vertex set $V(\overline{G}) = V(G)$ and the edge set $E(\overline{G}) = \{\{i,j\} \mid i,j \in V(\overline{G})$ and $\{i,j\} \notin E(G)\}$. Prove the following claims: *a.* $\mathrm{Aut}(G) = \mathrm{Aut}(\overline{G})$.

b. $\mathrm{Iso}(G,H) = \mathrm{Iso}(\overline{G},\overline{H})$.

c. $\overline{G}$ is connected if G is not connected.

Step	Alice	Erich	Bob
1	Alice and Bob agree upon a large prime number p and a primitive root r of p; both p and r are public		
2	chooses a large, secret number a at random and computes $\alpha = r^a \bmod p$		chooses a large, secret number b at random and computes $\beta = r^b \bmod p$
3		$\alpha \Rightarrow$ $\Leftarrow \beta$	
4	computes $k_A = \beta^a \bmod p$		computes $k_B = \alpha^b \bmod p$

Figure 7.7 The Diffie-Hellman secret-key agreement protocol.

7.2. Diffie and Hellman's secret-key agreement protocol

The basic number-theoretic facts presented in Subsection 7.1.3 will be needed in this and the subsequent sections. In particular, recall the multiplicative group $\mathbb{Z}_k^*$ from Example 7.3 and Euler's φ function. The arithmetics in remainder class rings will be explained at the end of this chapter, see Problem 7-1.

Figure 7.7 shows the Diffie-Hellman secret-key agreement protocol, which is based on exponentiation with base r and modulus p, where p is a prime number and r is a primitive root of p. A *primitive root of a number* n is any element $r \in \mathbb{Z}_n^*$ such that $r^d \not\equiv 1 \bmod n$ for each d with $1 \leq d < \varphi(n)$. A primitive root r of n generates the entire group $\mathbb{Z}_n^*$, i.e., $\mathbb{Z}_n^* = \{r^i \mid 0 \leq i < \varphi(n)\}$. Recall that for any prime number p the group $\mathbb{Z}_p^*$ has order $\varphi(p) = p - 1$. $\mathbb{Z}_p^*$ has exactly $\varphi(p-1)$ primitive roots, see also Exercise 7.2-1.

Example 7.5 Consider $\mathbb{Z}_5^* = \{1, 2, 3, 4\}$. Since $\mathbb{Z}_4^* = \{1, 3\}$, we have $\varphi(4) = 2$, and the two primitive roots of 5 are 2 and 3. Both 2 and 3 generate all of $\mathbb{Z}_5^*$, since:

$$\begin{array}{llll} 2^0 = 1; & 2^1 = 2; & 2^2 = 4; & 2^3 \equiv 3 \bmod 5\ ; \\ 3^0 = 1; & 3^1 = 3; & 3^2 \equiv 4 \bmod 5; & 3^3 \equiv 2 \bmod 5\ . \end{array}$$

Not every number has a primitive root; 8 is the smallest such example. It is known that a number n has a primitive root if and only if n either is from $\{1, 2, 4\}$, or has the form $n = q^k$ or $n = 2q^k$ for some odd prime number q.

Definition 7.12 (Discrete logarithm). *Let p be a prime number, and let r be a primitive root of p. The* modular exponential function with base r and modulus p *is the function* $\texttt{exp}_{r,p}$ *that maps from $\mathbb{Z}_{p-1}$ to $\mathbb{Z}_p^*$, and is defined by* $\texttt{exp}_{r,p}(a) = r^a \bmod p$. *Its inverse function is called the* discrete logarithm, *and maps for fixed p and r the value* $\texttt{exp}_{r,p}(a)$ *to a. We write* $a = \log_r \texttt{exp}_{r,p}(a) \bmod p$.

The protocol given in Figure 7.7 works, since (in the arithmetics modulo p)

$$k_A = \beta^a = r^{ba} = r^{ab} = \alpha^b = k_B\ ,$$

so Alice indeed computes the same key as Bob. Computing this key is not hard, since modular exponentiation can be performed efficiently using the square-and-multiply method from algorithm Square-and-Multiply.

Erich, however, has a hard time when he attempts to determine their key, since the discrete logarithm is considered to be a hard problem. That is why the modular exponential function $\mathtt{exp}_{r,p}$ is a candidate of a *one-way function*, a function that is easy to compute but hard to invert. Note that it is not known whether or not one-way functions indeed exist; this is one of the most challenging open research questions in cryptography. The security of many cryptosystems rests on the assumption that one-way functions indeed exist.

```
Square-and-Multiply($a, b, m$)
1                  ▷ $m$ is the modulus, $b < m$ is the base, and $a$ is the exponent
2  determine the binary expansion of the exponent $a = \sum_{i=0}^{k} a_i 2^i$, where $a_i \in \{0,1\}$
3  compute successively $b^{2^i}$, where $0 \le i \le k$, using that $b^{2^{i+1}} = \left(b^{2^i}\right)^2$
4  compute $b^a = \prod_{i=0,\ a_i=1}^{k} b^{2^i}$ in the arithmetics modulo $m$
5                  ▷ as soon as a factor $b^{2^i}$ in the product and $b^{2^{i+1}}$ are determined,
                   ▷ $b^{2^i}$ can be deleted and need not be stored
6  return $b^a$
```

Why can the modular exponential function $\mathtt{exp}_{r,p}(a) = r^a \bmod p$ be computed efficiently? Naively performed, this computation may require many multiplications, depending on the size of the exponent a. However, using algorithm Square-and-Multiply there is no need to perform $a-1$ multiplications as in the naive approach; no more than $2\log a$ multiplications suffice. The square-and-multiply algorithm thus speeds modular exponentiation up by an exponential factor.

Note that in the arithmetics modulo m, we have

$$b^a = b^{\sum_{i=0}^{k} a_i 2^i} = \prod_{i=0}^{k} \left(b^{2^i}\right)^{a_i} = \prod_{\substack{i=0 \\ a_i=1}}^{k} b^{2^i} .$$

Thus, the algorithm Square-and-Multiply is correct.

Example 7.6[Square-and-Multiply in the Diffie-Hellman Protocol] Alice and Bob have chosen the prime number $p = 5$ and the primitive root $r = 3$ of 5. Alice picks the secret number $a = 17$. In order to send her public number α to Bob, Alice wishes to compute $\alpha = 3^{17} = 129140163 \equiv 3 \bmod 5$. The binary expansion of the exponent is $17 = 1 + 16 = 2^0 + 2^4$. Alice successively computes the values:

$$3^{2^0} = 3;\ 3^{2^1} = 3^2 \equiv 4 \bmod 5;\ 3^{2^2} \equiv 4^2 \equiv 1 \bmod 5;\ 3^{2^3} \equiv 1^2 \equiv 1 \bmod 5;\ 3^{2^4} \equiv 1^2 \equiv 1 \bmod 5 .$$

Then, she computes $3^{17} \equiv 3^{2^0} \cdot 3^{2^4} \equiv 3 \cdot 1 \equiv 3 \bmod 5$. Note that Alice does not have to multiply 16 times but merely performs four squarings and one multiplication to determine $\alpha = 3 \bmod 5$.

Suppose that Bob has chosen the secret exponent $b = 23$. By the same method, he can compute his part of the key, $\beta = 3^{23} = 94143178827 \equiv 2 \bmod 5$. Now, Alice and Bob determine their joint secret key according to the Diffie-Hellman protocol from Figure 7.7; see Exercise 7.2-2.

Note that the protocol is far from being secure in this case, since the prime number $p = 5$ and the secret exponents $a = 17$ and $b = 23$ are much too small. This toy example was chosen just to explain how the protocol works. In practice, a and b should have at least 160 bits each.

If Erich was listening very careful, he knows the values p, r, α, and β after Alice and Bob have executed the protocol. His aim is to determine their joint secret key $k_A = k_B$. This problem is known as the *Diffie-Hellman problem*. If Erich were able to determine $a = \log_r \alpha \bmod p$ and $b = \log_r \beta \bmod p$ efficiently, he could compute the key $k_A = \beta^a \bmod p = \alpha^b \bmod p = k_B$ just like Alice and Bob and thus would have solved the Diffie-Hellman problem. Thus, this problem is no harder than the problem of computing discrete logarithms. The converse question of whether or not the Diffie-Hellman problem is *at least* as hard as solving the discrete logarithm (i.e., whether or not the two problems are equally hard) is still just an unproven conjecture. As many other cryptographic protocols, the Diffie-Hellman protocol currently has no proof of security.

However, since up to date neither the discrete logarithm nor the Diffie-Hellman problem can be solved efficiently, this direct attack is not a practical threat. On the other hand, there do exist other, indirect attacks in which the key is determined not immediately from the values α and β communicated in the Diffie-Hellman protocol. For example, Diffie-Hellman is vulnerable by the "*man-in-the-middle*" attack. Unlike the *passive* attack described above, this attack is *active*, since the attacker Erich aims at actively altering the protocol to his own advantage. He is the "man in the middle" between Alice and Bob, and he intercepts Alice's message $\alpha = r^a \bmod p$ to Bob and Bob's message $\beta = r^b \bmod p$ to Alice. Instead of α and β, he forwards his own values $\alpha_E = r^c \bmod p$ to Bob and $\beta_E = r^d \bmod p$ to Alice, where the private numbers c and d were chosen by Erich. Now, if Alice computes her key $k_A = (\beta_E)^a \bmod p$, which she falsely presumes to share with Bob, k_A in fact is a key for future communications with Erich, who determines the same key by computing (in the arithmetics modulo p)

$$k_E = \alpha^d = r^{ad} = r^{da} = (\beta_E)^a = k_A \ .$$

Similarly, Erich can share a key with Bob, who has not the slightest idea that he in fact communicates with Erich. This raised the issue of *authentication*, which we will deal with in more detail later in Section 7.5 about zero-knowledge protocols.

Exercises

7.2-1 *a.* How many primitive roots do $\mathbb{Z}_{13}^*$ and $\mathbb{Z}_{14}^*$ have?

b. Determine all primitive roots of $\mathbb{Z}_{13}^*$ and $\mathbb{Z}_{14}^*$, and prove that they indeed are primitive roots.

c. Show that every primitive root of 13 and of 14, respectively, generates all of $\mathbb{Z}_{13}^*$ and $\mathbb{Z}_{14}^*$.

7.2-2 *a.* Determine Bob's number $\beta = 3^{23} = 94143178827 \equiv 2 \bmod 5$ from Example 7.6 using the algorithm Square-and-Multiply.

b. For α and β from Example 7.6, determine the joint secret key of Alice and Bob according to the Diffie-Hellman protocol from Figure 7.7.

Step	Alice	Erich	Bob
1			chooses large prime numbers p and q at random and computes $n = pq$ and $\varphi(n) = (p-1)(q-1)$, his public key (n, e) and his private key d, which satisfy (7.8) and (7.9)
2		$\Leftarrow (n, e)$	
3	encrypts m as $c = m^e \bmod n$		
4		$c \Rightarrow$	
5			decrypts c as $m = c^d \bmod n$

Figure 7.8 The RSA protocol.

7.3. RSA and factoring

7.3.1. RSA

The RSA cryptosystem, which is named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman [49], is the first *public-key* cryptosystem. It is very popular still today and is used by various cryptographic applications. Figure 7.8 shows the single steps of the RSA protocol, which we will now describe in more detail, see also Example 7.7.

1. **Key generation:** Bob chooses two large prime numbers at random, p and q with $p \neq q$, and computes their product $n = pq$. He then chooses an exponent $e \in \mathbb{N}$ satisfying

$$1 < e < \varphi(n) = (p-1)(q-1) \quad \text{and} \quad \gcd(e, \varphi(n)) = 1 \tag{7.8}$$

and computes the inverse of e mod $\varphi(n)$, i.e., the unique number d such that

$$1 < d < \varphi(n) \quad \text{and} \quad e \cdot d \equiv 1 \bmod \varphi(n) \ . \tag{7.9}$$

The pair (n, e) is Bob's *public key*, and d is Bob's *private key*.

2. **Encryption:** As in Section 7.1, messages are strings over an alphabet Σ. Any message is subdivided into blocks of a fixed length, which are encoded as positive integers in $|\Sigma|$-adic representation. These integers are then encrypted. Let $m < n$ be the number encoding some block of the message Alice wishes to send to Bob. Alice knows Bob's public key (n, e) and encrypts m as the number $c = E_{(n,e)}(m)$, where the encryption function is defined by

$$E_{(n,e)}(m) = m^e \bmod n \ .$$

3. **Decryption:** Let c with $0 \leq c < n$ be the number encoding one block of the ciphertext, which is received by Bob and also by the eavesdropper Erich. Bob decrypts c by using his private key d and the following decryption function

$$D_d(c) = c^d \bmod n \ .$$

Theorem 7.13 states that the RSA protocol described above indeed is a cryptosystems in the sense of Definition 7.1. The proof of Theorem 7.13 is left to the reader as Exercise 7.3-1.

Theorem 7.13 *Let (n, e) be the public key and d be the private key in the RSA protocol. Then, for each message m with $0 \leq m < n$,*

$$m = (m^e)^d \bmod n \ .$$

Hence, RSA is a public-key cryptosystem.

To make RSA encryption and (authorised) decryption efficient, the algorithm Square-and-Multiply algorithm is again employed for fast exponentiation.

How should one choose the prime numbers p and q in the RSA protocol? First of all, they must be large enough, since otherwise Erich would be able to factor the number n in Bob's public key (n, e) using the extended Euclidean algorithm. Knowing the prime factors p and q of n, he could then easily determine Bob's private key d, which is the unique inverse of e mod $\varphi(n)$, where $\varphi(n) = (p-1)(q-1)$. To keep the prime numbers p and q secret, they thus must be sufficiently large. In practice, p and q should have at least 80 digits each. To this end, one generates numbers of this size randomly and then checks using one of the known randomised primality tests whether the chosen numbers are primes indeed. By the Prime Number Theorem, there are about $N/\ln N$ prime numbers not exceeding N. Thus, the odds are good to hit a prime after reasonably few trials.

In theory, the primality of p and q can be decided even in *deterministic* polynomial time. Agrawal et al. [1, 2] recently showed that the primality problem, which is defined by

$$\texttt{PRIMES} = \{\text{bin}(n) \mid n \text{ is prime}\} \ ,$$

is a member of the complexity class P. Their breakthrough solved a longstanding open problem in complexity theory: Among a few other natural problems such as the graph isomorphism problem, the primality problem was considered to be one of the rare candidates of problems that are neither in P nor NP-complete.[2] For practical purposes, however, the known randomised algorithms are more useful than the deterministic algorithm by Agrawal et al. The running time of $\mathcal{O}(n^{12})$ obtained in their original paper [1, 2] could be improved to $\mathcal{O}(n^6)$ meanwhile, applying a more sophisticated analysis, but this running time is still not as good as that of the randomised algorithms.

[2] The complexity classes P and NP will be defined in Section 8.1 and the notion of NP-completeness will be defined in Section 8.2.

```
Miller-Rabin($n$)
determine the representation $n - 1 = 2^k m$, where $m$ and $n$ are odd
choose a number $z \in \{1, 2, \ldots, n-1\}$ at random under the uniform distribution
compute $x = z^m \bmod n$
if ($x \equiv 1 \bmod n$)
   then return "$n$ is a prime number"
   else for $j \leftarrow 0$ to $k - 1$
         do if ($x \equiv -1 \bmod n$)
               then return "$n$ is a prime number"
               else  $x \leftarrow x^2 \bmod n$
                     return "$n$ is not a prime number"
```

One of the most popular randomised primality tests is the algorithm Miller-Rabin developed by Rabin [47], which is based on the ideas underlying the deterministic algorithm of Miller [40]. The Miller-Rabin test is a so-called Monte Carlo algorithm, since the "no" answers of the algorithm are always reliable, whereas its "yes" answers have a certain error probability. An alternative to the Miller-Rabin test is the primality test of Solovay and Strassen [62]. Both primality tests run in time $\mathcal{O}(n^3)$. However, the Solovay-Strassen test is less popular because it is not as efficient in practice and also less accurate than the Miller-Rabin test.

The class of problems solvable via Monte Carlo algorithms with always reliable "yes" answers is named RP, which stands for *R*andomised *P*olynomial Time. The complementary class, $\mathrm{coRP} = \{L \mid \overline{L} \in \mathrm{RP}\}$, contains all those problems solvable via Monte Carlo algorithms with always reliable "no" answers. Formally, RP is defined via nondeterministic polynomial-time Turing machines (NPTMs, for short; see Section 8.1 and in particular Definitions 8.1, 8.2, and 8.3) whose computations are viewed as random processes: For each nondeterministic guess, the machine flips an unbiased coin and follows each of the resulting two next configurations with probability $1/2$ until a final configuration is reached. Depending on the number of accepting computation paths of the given NPTM, one obtains a certain acceptance probability for each input. Errors may occur. The definition of RP requires that the error probability must be below the threshold of $1/2$ for an input to be accepted, and there must occur no error at all for an input to be rejected.

Definition 7.14 (Randomised polynomial time). *The class* RP *consists of exactly those problems A for which there exists an NPTM M such that for each input x, if $x \in A$ then $M(x)$ accepts with probability at least $1/2$, and if $x \notin A$ then $M(x)$ accepts with probability 0.*

Theorem 7.15 `PRIMES` *is in* coRP*.*

Theorem 7.15 follows from the fact that, for example, the Miller-Rabin test is a Monte Carlo algorithm for the primality problem. We present a proof sketch only. We show that the Miller-Rabin test accepts `PRIMES` with one-sided error probability as in Definition 7.14: If the given number n (represented in binary) is a prime number then the algorithm cannot answer erroneously that n is not prime. For a contradiction, suppose n is prime but the Miller-Rabin test halts with the output: "n is not a prime number". Hence, $z^m \not\equiv 1 \bmod n$. Since x is squared in each iteration of the `for` loop, we sequentially test the values

$$z^m, z^{2m}, \ldots, z^{2^{k-1}m}$$

modulo n. By assumption, for none of these values the algorithm says n were prime. It follows that for each j with $0 \le j \le k-1$,

$$z^{2^j m} \not\equiv -1 \bmod n \; .$$

Since $n - 1 = 2^k m$, Fermat's Little Theorem (see Corollary 7.5) implies $z^{2^k m} \equiv 1 \bmod n$. Thus, $z^{2^{k-1}m}$ is a square roots of 1 modulo n. Since n is prime, there are only two square roots of 1 modulo n, namely $\pm 1 \bmod n$, see Exercise 7.3-1. Since $z^{2^{k-1}m} \not\equiv -1 \bmod n$, we must have $z^{2^{k-1}m} \equiv 1 \bmod n$. But then, $z^{2^{k-2}m}$ again is a square root of 1 modulo n. By the same argument, $z^{2^{k-2}m} \equiv 1 \bmod n$. Repeating this argument again and again, we eventually obtain $z^m \equiv 1 \bmod n$, a contradiction. It follows that the Miller-Rabin test works correctly for each prime number. On the other hand, if n is not a prime number, it can be shown that the error probability of the Miller-Rabin tests does not exceed the threshold of $1/4$. Repeating the number of independent trials, the error probability can be made arbitrarily close to zero, at the cost of increasing the running time of course, which still will be polynomially in $\log n$, where $\log n$ is the size of the input n represented in binary.

Example 7.7[RSA] Suppose Bob chooses the prime numbers $p = 67$ and $q = 11$. Thus, $n = 67 \cdot 11 = 737$, so we have $\varphi(n) = (p-1)(q-1) = 66 \cdot 10 = 660$. If Bob now chooses the smallest exponent possible for $\varphi(n) = 660$, namely $e = 7$, then $(n, e) = (737, 7)$ is his public key. Using the extended Euclidean algorithm, Bob determines his private key $d = 283$, and we have $e \cdot d = 7 \cdot 283 = 1981 \equiv 1 \bmod 660$; see Exercise 7.3-2. As in Section 7.1, we identify the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ with the set $\mathbb{Z}_{26} = \{0, 1, \ldots, 25\}$. Messages are strings over Σ and are encoded in blocks of fixed length as natural numbers in 26-adic representation. In our example, the block length is $\ell = \lfloor \log_{26} n \rfloor = \lfloor \log_{26} 737 \rfloor = 2$.

More concretely, any block $b = b_1 b_2 \cdots b_\ell$ of length ℓ with $b_i \in \mathbb{Z}_{26}$ is represented by the number

$$m_b = \sum_{i=1}^{\ell} b_i \cdot 26^{\ell - i} \; .$$

Since $\ell = \lfloor \log_{26} n \rfloor$, we have

$$0 \le m_b \le 25 \cdot \sum_{i=1}^{\ell} 26^{\ell - i} = 26^{\ell} - 1 < n \; .$$

The RSA encryption function encrypts the block b (i.e., the corresponding number m_b) as $c_b = (m_b)^e \bmod n$. The ciphertext for block b then is $c_b = c_0 c_1 \cdots c_\ell$ with $c_i \in \mathbb{Z}_{26}$. Thus, RSA maps blocks of length ℓ injectively to blocks of length $\ell + 1$. Figure 7.9 shows how to subdivide a message of length 34 into 17 blocks of length 2 and how to encrypt the single blocks, which are represented by numbers. For example, the first block, "RS", is turned into a number as follows: "R" corresponds to 17 and "S" to 18, and we have

$$17 \cdot 26^1 + 18 \cdot 26^0 = 442 + 18 = 460 \; .$$

M	R S	A I	S T	H E	K E	Y T	O P	U B	L I	C K	E Y	C R	Y P	T O	G R	A P	H Y
m_b	460	8	487	186	264	643	379	521	294	62	128	69	639	508	173	15	206
c_b	697	387	229	340	165	223	586	5	189	600	325	262	100	689	354	665	673

Figure 7.9 Example of an RSA encryption (M = Message).

Step	Alice	Erich	Bob
1	chooses $n = pq$, her public key (n, e) and her private key d just as Bob does in the RSA protocol in Figure 7.8		
2		$(n, e) \Rightarrow$	
3	signs the message m by $\text{sig}_A(m) = m^d \bmod n$		
4		$(m, \text{sig}_A(m)) \Rightarrow$	
5			checks $m \equiv (\text{sig}_A(m))^e \bmod n$ to verify Alice's signature

Figure 7.10 Digital RSA signatures.

The resulting number c_b is written again in 26-adic representation and can have the length $\ell+1$: $c_b = \sum_{i=0}^{\ell} c_i \cdot 26^{\ell-i}$, where $c_i \in \mathbb{Z}_{26}$, see also Exercise 7.3-2. So, the first block, $697 = 676 + 21 = 1 \cdot 26^2 + 0 \cdot 26^1 + 21 \cdot 26^0$, is encrypted to yield the ciphertext "BAV".

Decryption is also done blockwise. In order to decrypt the first block with the private key $d = 283$, compute $697^{283} \bmod 737$, again using fast exponentiation with SQUARE-AND-MULTIPLY. To prevent the numbers from becoming too large, it is recommendable to reduce modulo $n = 737$ after each multiplication. The binary expansion of the exponent is $283 = 2^0 + 2^1 + 2^3 + 2^4 + 2^8$, and we obtain

$$697^{283} \equiv 697^{2^0} \cdot 697^{2^1} \cdot 697^{2^3} \cdot 697^{2^4} \cdot 697^{2^8} \equiv 697 \cdot 126 \cdot 9 \cdot 81 \cdot 15 \equiv 460 \bmod 737$$

as desired.

7.3.2. Digital RSA signatures

The public-key cryptosystem RSA from Figure 7.8 can be modified so as to produce digital signatures. This protocol is shown in Figure 7.10. It is easy to see that this protocol works as desired; see Exercise 7.3-2. This digital signature protocol is vulnerable to "*chosen-plaintext attacks*" in which the attacker can choose a plaintext and obtains the corresponding ciphertext. This attack is described, e.g., in [50].

7.3.3. Security of RSA

As mentioned above, the security of the RSA cryptosystem crucially depends on the assumption that large numbers cannot be factored in a reasonable amount of time. Despite much effort in the past, no efficient factoring algorithm has been found until now. Thus, it is widely believed that there is no such algorithm and the factoring problem is hard. A rigorous proof of this hypothesis, however, has not been found either. And even if one could prove this hypothesis, this would not imply a proof of security of RSA. Breaking RSA is at most as hard as the factoring problem; however, the converse is not known to hold. That is, it is not known whether these two problems are equally hard. It may be possible to break RSA without factoring n.

We omit listing potential attacks on the RSA system here. Rather, the interested reader is pointed to the comprehensive literature on this subject; note also Problem 7-4 at the end of this chapter. We merely mention that for each of the currently known attacks on RSA, there are suitable countermeasures, rules of thumb that either completely prevent a certain attack or make its probability of success negligibly small. In particular, it is important to take much care when choosing the prime numbers p and q, the modulus $n = pq$, the public exponent e, and the private key d.

Finally, since the *factoring attacks* on RSA play a particularly central role, we briefly sketch two such attacks. The first one is based on Pollard's $(p-1)$ method [45]. This method is effective for composite numbers n having a prime factor p such that the prime factors of $p - 1$ each are small. Under this assumption, a multiple ν of $p - 1$ can be determined without knowing p. By Fermat's Little Theorem (see Corollary 7.5), it follows that $a^\nu \equiv 1 \bmod p$ for all integers a coprime with p. Hence, p divides $a^\nu - 1$. If n does not divide $a^\nu - 1$, then $\gcd(a^\nu - 1, n)$ is a nontrivial divisor of n. Thus, the number n can be factored.

How can the multiple ν of $p - 1$ be determined? Pollard's $(p - 1)$ method uses as candidates for ν the products of prime powers below a suitably chosen bound S:

$$\nu = \prod_{q \text{ is prime},\ q^k \leq S} q^k .$$

If all prime powers dividing $p - 1$ are less than S, then ν is a multiple of $p - 1$. The algorithm determines $\gcd(a^\nu - 1, n)$ for a suitably chosen base a. If no nontrivial divisor of n is found, the algorithm is restarted with a new bound $S' > S$.

Other factoring methods, such as the *quadratic sieve*, are described, e.g., in [52, 63]. They use the following simple idea. Suppose n is the number to be factored. Using the sieve, determine numbers a and b such that:

$$a^2 \equiv b^2 \bmod n \quad \text{and} \quad a \not\equiv \pm b \bmod n . \tag{7.10}$$

Hence, n divides $a^2 - b^2 = (a-b)(a+b)$ but neither $a-b$ nor $a+b$. Thus, $\gcd(a-b, n)$ is a nontrivial factor of n.

There are also sieve methods other than the quadratic sieve. These methods are distinguished by the particular way of how to determine the numbers a and b such that (7.10) is satisfied. A prominent example is the "general number field sieve", see [36].

Exercises

7.3-1 *a.* Prove Theorem 7.13. *Hint.* Show $(m^e)^d \equiv m \bmod p$ and $(m^e)^d \equiv m \bmod q$ using Corollary 7.5, Fermat's Little Theorem. Since p and q are prime numbers with $p \neq q$ and $n = pq$, the claim $(m^e)^d \equiv m \bmod n$ now follows from the Chinese remainder theorem.

b. The proof sketch of Theorem 7.15 uses the fact that any prime number n can

Step	Alice	Erich	Bob
1	chooses two large numbers x and y at random, keeps x secret and computes $x\sigma y$		
2		$(y, x\sigma y) \Rightarrow$	
3			chooses a large number z at random, keeps z secret and computes $y\sigma z$
4		$\Leftarrow y\sigma z$	
5	computes $k_A = x\sigma(y\sigma z)$		computes $k_B = (x\sigma y)\sigma z$

Figure 7.11 The Rivest-Sherman protocol for secret-key agreement, based on σ.

have only two square roots of 1 modulo n, namely $\pm 1 \bmod n$. Prove this fact. *Hint.* It may be helpful to note that r is a square root of 1 modulo n if and only if n divides $(r-1)(r+1)$.

7.3-2 *a.* Let $\varphi(n) = 660$ and $e = 7$ be the values from Example 7.7. Show that the extended Euclidean algorithm indeed provides the private key $d = 283$, the inverse of $7 \bmod 660$.

b. Consider the plaintext in Figure 7.9 from Example 7.7 and its RSA encryption. Determine the encoding of this ciphertext by letters of the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ for each of the 17 blocks.

c. Decrypt each of the 17 ciphertext blocks in Figure 7.9 and show that the original message is obtained indeed.

d. Prove that the RSA digital signature protocol from Figure 7.10 works.

7.4. The protocols of Rivest, Rabi, and Sherman

Rivest, Rabi, and Sherman proposed protocols for secret-key agreement and digital signatures. The secret-key agreement protocol given in Figure 7.11 is due to Rivest and Sherman. Rabi and Sherman modified this protocol to a digital signature protocol, see Exercise 7.4-1

The Rivest-Sherman protocol is based on a *total, strongly noninvertible, associative one-way function.* Informally put, a *one-way function* is a function that is easy to compute but hard to invert. One-way functions are central cryptographic primitives and many cryptographic protocols use them as their key building blocks. To capture the notion of noninvertibility, a variety of models and, depending on the model used, various candidates for one-way functions have been proposed. In most cryptographic applications, noninvertibility is defined in the average-case complexity model. Unfortunately, it is not known whether such one-way functions exist; the security of the corresponding protocols is merely based on the assumption of their existence. Even in the less challenging worst-case model, in which so-called "complexity-theoretic" one-way functions are usually defined, the question of whether any type of one-way function exists remains an open issue after many years of research.

A total (i.e., everywhere defined) function σ mapping from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ is ***associative*** if and only if $(x\sigma y)\sigma z = x\sigma(y\sigma z)$ holds for all $x, y, z \in \mathbb{N}$, where we use the infix notation $x\sigma y$ instead of the prefix notation $\sigma(x, y)$. This property implies that the above protocol works:

$$k_A = x\sigma(y\sigma z) = (x\sigma y)\sigma z = k_B \ ,$$

so Alice and Bob indeed compute the same secret key.

The notion of strong noninvertibility is not to be defined formally here. Informally put, σ is said to be *strongly noninvertible* if σ is not only a one-way function, but even if in addition to the function value one of the corresponding arguments is given, it is not possible to compute the other argument efficiently. This property implies that the attacker Erich, knowing y and $x\sigma y$ and $y\sigma z$, is not able to compute the secret numbers x and z, from which he could easily determine the secret key $k_A = k_B$.

Exercises

7.4-1 Modify the Rivest-Sherman protocol for secret-key agreement from Figure 7.11 to a protocol for digital signatures.

7.4-2 *(a.* Try to give a formal definition of the notion of "strong noninvertibility" that is defined only informally above. Use the worst-case complexity model.

b. Suppose σ is a *partial* function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, i.e., σ may be undefined for some pairs in $\mathbb{N} \times \mathbb{N}$. Give a formal definition of "associativity" for partial functions. What is wrong with the following (flawed) attempt of a definition: "A partial function $\sigma : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is said to be *associative* if and only if $x\sigma(y\sigma z) = (x\sigma y)\sigma z$ holds for all $x, y, z \in \mathbb{N}$ for which each of the four pairs (x, y), (y, z), $(x, y\sigma z)$, and $(x\sigma y, z)$ is in the domain of σ." *Hint.* A comprehensive discussion of these notions can be found in [23, 25, 26].

7.5. Interactive proof systems and zero-knowledge

7.5.1. Interactive proof systems and Arthur-Merlin games

In Section 7.2, the "*man-in-the-middle*" attack on the Diffie-Hellman protocol was mentioned. The problem here is that Bob has not verified the true identity of his communication partner before executing the protocol. While he assumes to communicate with Alice, he in fact exchanges messages with Erich. In other words, Alice's task is to convince Bob of her true identity without any doubt. This cryptographic task is called *authentication*. Unlike digital signatures, whose purpose is to authenticate electronically transmitted *documents* such as emails, electronic contracts, etc., the goal now is to authenticate *individuals* such as human or computer parties participating in a cryptographic protocol.[3]

In order to authenticate herself, Alice might try to prove her identity by a secret information known to her alone, say by giving her PIN ("*P*ersonal *I*dentifaction *N*umber") or any other private information that no one knows but her. However, there is a catch. To prove her identity, she would have to give her secret away to Bob.

[3] Here, an "individual" or a "party" is not necessarily a human being; it may also be a computer program that automatically executes a protocol with another computer program.

But then, it no longer is a secret! Bob, knowing her secret, might pretend to be Alice in another protocol he executes with Chris, a third party. So the question is how to prove knowledge of a secret without giving it away. This is what zero-knowledge is all about. Zero-knowledge protocols are special interactive proof systems, which were introduced by Goldwasser, Micali, and Rackoff and, independently, by Babai and Moran. Babai and Moran's notion (which is essentially equivalent to the interactive proof systems proposed by Goldwasser et al.) is known as Arthur-Merlin games, which we will now describe informally.

Merlin and Arthur wish to jointly solve a problem L, i.e., they wish to jointly decide whether or not a given input x belongs to L. The mighty wizard Merlin is represented by an NP machine M, and the impatient King Arthur is represented by a randomised polynomial-time Turing machine A. To make their decision, they play the following game, where they are taking turns to make alternating moves. Merlin's intention is always to convince Arthur that x belongs to L (no matter whether or not that indeed is the case). Thus, each of Merlin's moves consists in presenting a proof for "$x \in L$", which he obtains by simulating $M(x, y)$, where x is the input and y describes all previous moves in this game. That is, the string y encodes all previous nondeterministic choices of M and all previous random choices of A.

King Arthur, however, does not trust Merlin. Of course, he cannot check the mighty wizard's proofs all alone; this task simply exceeds his computational power. But he knows Merlin well enough to express some doubts. So, he replies with a nifty challenge to Merlin by picking some details of his proofs at random and requiring certificates for them that he can verify. In order to satisfy Arthur, Merlin must convince him with overwhelming probability. Thus, each of Arthur's moves consists in the simulation of $A(x, y)$, where x again is the input and y describes the previous history of the game.

The idea of Arthur-Merlin games can be captured via alternating existential and probabilistic quantifiers, where the former formalise Merlin's NP computation and the latter formalise Arthur's randomised polynomial-time computation.[4] In this way, a hierarchy of complexity classes can be defined, the so-called Arthur-Merlin hierarchy. We here present only the class MA from this hierarchy, which corresponds to an Arthur-Merlin game with two moves, with Merlin moving first.

Definition 7.16 (MA in the Arthur-Merlin hierarchy). *The class* MA *contains exactly those problems L for which there exists an* NP *machine M and a randomised polynomial-time Turing machine A such that for each input x:*

- *If $x \in L$ then there exists a path y of $M(x)$ such that $A(x, y)$ accepts with probability at least $3/4$ (i.e., Arthur cannot refute Merlin's proof y for "$x \in L$", and Merlin thus wins).*
- *If $x \notin L$ then for each path y of $M(x)$, $A(x, y)$ rejects with probability at least $3/4$ (i.e., Arthur is not taken in by Merlin's wrong proofs for "$x \in L$" and thus wins).*

Analogously, the classes AM, MAM, AMA, . . . *can be defined, see Exercise 7.5-1.*

[4] This is similar to the well-known characterisation of the levels of the polynomial hierarchy via alternating $\exists$ and $\forall$ quantifiers, see Section 8.4 and in particular item 3 Theorem 8.11.

In Definition 7.16, the probability threshold of $3/4$ for Arthur to accept or to reject, respectively, is chosen at will and does not appear to be large enough at first glance. In fact, the probability of success can be amplified using standard techniques and can be made arbitrarily close to one. In other words, one might have chosen even a probability threshold as low as $1/2 + \varepsilon$, for an arbitrary fixed constant $\varepsilon > 0$, and would still have defined the same class. Furthermore, it is known that for a constant number of moves, the Arthur-Merlin hierarchy collapses down to AM:

$$\mathrm{NP} \subseteq \mathrm{MA} \subseteq \mathrm{AM} = \mathrm{AMA} = \mathrm{MAM} = \cdots .$$

It is an open question whether or not any of the inclusions $\mathrm{NP} \subseteq \mathrm{MA} \subseteq \mathrm{AM}$ is a strict one.

A similar model, which can be used as an alternative to the Arthur-Merlin games, are the *interactive proof systems* mentioned above. The two notions use different terminology: Merlin now is called the “prover” and Arthur the “verifier”. Also, their communication is not interpreted as a game but rather as a protocol. Another difference between the two models, which appears to be crucial at first, is that Arthur’s random bits are public (so, in particular, Merlin knows them), whereas the random bits of the verifier in an interactive proof system are private. However, Goldwasser and Sipser [21] proved that, in fact, it does not matter whether the random bits are private or public, so Arthur-Merlin games essentially are equivalent to interactive proof systems.

If one allows a polynomial number of moves (instead of a constant number), then one obtains the complexity class IP. Note that interactive proof systems are also called IP protocols. By definition, IP contains all of NP. In particular, the graph isomorphism problem is in IP. We will see later that IP also contains problems from $\mathrm{coNP} = \{\overline{L} \mid L \in \mathrm{NP}\}$ that are supposed to be not in NP. In particular, the proof of Theorem 8.16 shows that the complement of the graph isomorphism problem is in AM and thus in IP. A celebrated result by Shamir [59] says that IP equals PSPACE, the class of problems solvable in polynomial space.

Let us now turn back to the problem of authentication mentioned above, and to the related notion of zero-knowledge protocols. Here is the idea. Suppose Arthur and Merlin play one of their games. So, Merlin sends hard proofs to Arthur. Merlin alone knows how to get such proofs. Being a wise wizard, he keeps this knowledge secret. And he uses his secret to authenticate himself in the communication with Arthur.

Now suppose that malicious wizard Marvin wishes to fool Arthur by pretending to be Merlin. He disguises as Merlin and uses his magic to look just like him. However, he does not know Merlin’s secret of how to produce hard proofs. His magic is no more powerful than that of an ordinary randomised polynomial-time Turing machine. Still, he seeks to simulate the communication between Merlin and Arthur. An interactive proof system has the ***zero-knowledge property*** if the information communicated between Marvin and Arthur cannot be distinguished from the information communicated between Merlin and Arthur. Note that Marvin, who does not know Merlin’s secret, cannot introduce any information about this secret into the simulated IP protocol. Nonetheless, he is able to perfectly copy the original protocol, so no one can tell a difference. Hence, the (original) protocol itself must have the property that it does not leak any information whatsoever about Merlin's secret. If there is nothing to put in, there can be nothing to take out.

Definition 7.17 (Zero-knowledge protocol). *Let L be any set in* IP*, and let (M, A) be an interactive proof system for L, where M is an NPTM and A is a randomised polynomial-time Turing machine. The* IP *protocol (M, A) is a* zero-knowledge protocol for L *if and only if there exists a randomised polynomial-time Turing machine such that $(\widehat{M}, A)$ simulates the original protocol (M, A) and, for each $x \in L$, the tuples $(m_1, m_2, \ldots, m_k)$ and $(\widehat{m}_1, \widehat{m}_2, \ldots, \widehat{m}_k)$ representing the information communicated in (M, A) and in $(\widehat{M}, A)$, respectively, are identically distributed over the random choices in (M, A) and in $(\widehat{M}, A)$, respectively.*

The notion defined above is called "*honest-verifier perfect zero-knowledge*" in the literature, since (a) it is assumed that the verifier is *honest* (which may not necessarily be true in cryptographic applications though), and (b) it is required that the information communicated in the simulated protocol *perfectly* coincides with the information communicated in the original protocol. Assumption (a) may be somewhat too idealistic, and assumption (b) may be somewhat too strict. That is why also other variants of zero-knowledge protocols are studied, see the notes at the end of this chapter.

7.5.2. Zero-knowledge protocol for graph isomorphism

Let us consider a concrete example now. As mentioned above, the graph isomorphism problem (`GI`, for short) is in NP, and the complementary problem $\overline{\texttt{GI}}$ is in AM, see the proof of Theorem 8.16. Thus, both problems are contained in IP. We now describe a zero-knowledge protocol for `GI` that is due to Goldreich, Micali, and Wigderson [16]. Figure 7.12 shows this IP protocol between the prover Merlin and the verifier Arthur.

Although there is no efficient algorithm known for `GI`, Merlin can solve this problem, since `GI` is in NP. However, there is no need for him to do so in the protocol. He can simply generate a large graph G_0 with n vertices and a random permutation $\pi \in \mathfrak{S}_n$. Then, he computes the graph $G_1 = \pi(G_0)$ and makes the pair (G_0, G_1) public. The isomorphism π between G_0 and G_1 is kept secret as Merlin's private information.

Of course, Merlin cannot send π to Arthur, since he does not want to give his secret away. Rather, to prove that the two graphs, G_0 and G_1, indeed are isomorphic, Merlin randomly chooses an isomorphism ρ under the uniform distribution and a bit $a \in \{0, 1\}$ and computes the graph $H = \rho(G_a)$. He then sends H to Arthur whose response is a challenge for Merlin: Arthur sends a random bit $b \in \{0, 1\}$, chosen under the uniform distribution, to Merlin and requests to see an isomorphism σ between G_b and H. Arthur accepts if and only if σ indeed satisfies $\sigma(G_b) = H$.

The protocol works, since Merlin knows his secret isomorphism π and his random permutation ρ: It is no problem for Merlin to compute the isomorphism σ between G_b and H and thus to authenticate himself. The secret π is not given away. Since G_0 and G_1 are isomorphic, Arthur accepts with probability one. The case of two nonisomorphic graphs does not need to be considered here, since Merlin has chosen isomorphic graphs G_0 and G_1 in the protocol; see also the proof of Theorem 8.16.

Step	Merlin		Arthur
1	randomly chooses a permutation ρ on $V(G_0)$ and a bit a, computes $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			chooses a random bit b and requests to see an isomorphism between G_b and H
4		$\Leftarrow b$	
5	computes the isomorphism σ satisfying $\sigma(G_b) = H$ as follows: if $b = a$ then $\sigma = \rho$; if $0 = b \neq a = 1$ then $\sigma = \pi\rho$; if $1 = b \neq a = 0$ then $\sigma = \pi^{-1}\rho$.		
6		$\sigma \Rightarrow$	
7			verifies that $\sigma(G_b) = H$ and accepts accordingly

Figure 7.12 Goldreich, Micali, and Wigderson's zero-knowledge protocol for `GI`.

Step	Marvin		Arthur
1	randomly chooses a permutation ρ on $V(G_0)$ and a bit a, computes $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			chooses a random bit b and requests to see an isomorphism between G_b and H
4		$\Leftarrow b$	
5	if $b \neq a$ then $\widehat{M}$ deletes all information communicated in this round; if $b = a$ then $\widehat{M}$ sends $\sigma = \rho$		
6		$\sigma \Rightarrow$	
7			$b = a$ implies $\sigma(G_b) = H$, so Arthur accepts Marvin's faked identity

Figure 7.13 Simulation of the zero-knowledge protocol for `GI` without knowing π.

Now, suppose, Marvin wishes to pretend to be Merlin when communicating with Arthur. He does know the graphs G_0 and G_1, but he doesn't know the secret isomorphism π. Nonetheless, he tries to convince Arthur that he does know π. If Arthur's random bit b happens to be the same as his bit a, to which Marvin committed *before* he sees b, then Marvin wins. However, if $b \neq a$, then computing $\sigma = \pi\rho$ or $\sigma = \pi^{-1}\rho$ requires knowledge of π. Since `GI` is not efficiently solvable (and even too hard for a randomised polynomial-time Turing machine), Marvin cannot determine the isomorphism π for sufficiently large graphs G_0 and G_1. But without knowing π, all he can do is guess. His chances of hitting a bit b with $b = a$ are at most $1/2$. Of course, Marvin can always guess, so his success probability is exactly $1/2$. If Arthur challenges him in r independent rounds of this protocol again and again, the probability of Marvin's success will be only 2^{-r}. Already for $r = 20$, this probability is negligibly small: Marvin's probability of success is then less than one in one million.

It remains to show that the protocol from Figure 7.12 is a zero-knowledge protocol. Figure 7.13 shows a simulated protocol with Marvin who does not know Merlin's secret π but pretends to know it. The information communicated in one round of the protocol has the form of a triple: (H, b, σ). If Marvin is lucky enough to choose a random bit a with $a = b$, he can simply send $\sigma = \rho$ and wins: Arthur (or any third party watching the communication) will not notice the fraud. On the other hand, if $a \neq b$ then Marvin's attempt to betray will be uncovered. However, that is no problem for the malicious wizard: He simply deletes this round from the protocol and repeats. Thus, he can produce a sequence of triples of the form (H, b, σ) that is indistinguishable from the corresponding sequence of triples in the original protocol between Merlin and Arthur. It follows that Goldreich, Micali, and Wigderson's protocol for `GI` is a zero-knowledge protocol.

Exercises

7.5-1 Arthur-Merlin hierarchy:

a. Analogously to MA from Definition 7.16, define the other classes AM, MAM, AMA, ... of the Arthur-Merlin hierarchy.

b. What is the inclusion structure between the classes MA, coMA, AM, coAM, and the classes of the polynomial hierarchy defined in Definition 8.10 of Subsection 8.4.1

7.5-2 Zero-knowledge protocol for graph isomorphism:

a. Consider the graphs G and H from Example 7.4 in Section 7.1.3. Execute the zero-knowledge protocol from Figure 7.12 with the graphs $G_0 = G$ and $G_1 = H$ and the isomorphism $\pi = \binom{1\ 2\ 3\ 4\ 5}{3\ 4\ 1\ 5\ 2}$. Use an isomorphism ρ of your choice, and try all possibilities for the random bits a and b. Repeat this Arthur-Merlin game with an unknown isomorphism ρ chosen by somebody else.

b. Modify the protocol from Figure 7.12 such that, with only one round of the protocol, Marvin's success probability is less than 2^{-25}.

Problems

7-1 Arithmetics in the ring $\mathbb{Z}_k$

Let $k \in \mathbb{N}$ and $x, y, z \in \mathbb{Z}$. We say *x is congruent to y modulo k* ($x \equiv y \bmod k$, for short) if and only if k divides the difference $y - x$. For example, $-10 \equiv 7 \bmod 17$ and $4 \equiv 0 \bmod 2$. The congruence $\equiv$ modulo k defines an *equivalence relation* on $\mathbb{Z}$, i.e., it is *reflexive* ($x \equiv x \bmod k$), *symmetric* (if $x \equiv y \bmod k$ then $y \equiv x \bmod k$), and *transitive* ($x \equiv y \bmod k$ and $y \equiv z \bmod k$ imply $x \equiv z \bmod k$).

The set $x + k\mathbb{Z} = \{y \in \mathbb{Z} \mid y \equiv x \bmod k\}$ is the *remainder class of* $x \bmod k$. For example, the remainder class of $3 \bmod 7$ is

$$3 + 7\mathbb{Z} = \{3, 3 \pm 7, 3 \pm 2 \cdot 7, \ldots\} = \{3, 10, -4, 17, -11, \ldots\} .$$

We represent the remainder class of $x \bmod k$ by the smallest natural number in $x + k\mathbb{Z}$. For instance, 3 represents the remainder class of $3 \bmod 7$. The set of all remainder classes $\bmod k$ is $\mathbb{Z}_k = \{0, 1, \ldots, k-1\}$. On $\mathbb{Z}_k$, *addition* is defined by $(x+k\mathbb{Z})+(y+k\mathbb{Z}) = (x+y)+k\mathbb{Z}$, and *multiplication* is defined by $(x+k\mathbb{Z})\cdot(y+k\mathbb{Z}) =$

$(x \cdot y) + k\mathbb{Z}$. For example, in the arithmetics modulo 7, we have $(4+7\mathbb{Z})+(5+7\mathbb{Z}) = (4+5)+7\mathbb{Z} = 2+7\mathbb{Z}$ and $(4+7\mathbb{Z})\cdot(5+7\mathbb{Z}) = (4\cdot 5)+7\mathbb{Z} = 6+7\mathbb{Z}$.

Prove that in the arithmetics modulo k:

a. $(\mathbb{Z}_k, +, \cdot)$ is a commutative ring with one;

b. $\mathbb{Z}_k^*$, which is defined in Example 7.3, is a multiplicative group;

c. $(\mathbb{Z}_p, +, \cdot)$ is a field for each prime number p.

d. Prove that the neutral element of a group and the inverse of each group element are unique.

e. Let $\mathfrak{R}$ be a commutative ring with one. Prove that the set of all invertible elements in $\mathfrak{R}$ forms a multiplicative group. Determine this group for the ring $\mathbb{Z}_k$.

7-2 Tree isomorphism

The graph isomorphism problem can be solved efficiently on special graph classes, such as on the class of trees. An (undirected) *tree* is a connected graph without cycles. (A *cycle* is a sequence of pairwise incident edges that returns to the point of origin.) The *leaves of a tree* are the vertices with degree one. The *tree isomorphism problem* is defined by

$$\mathtt{TI} \quad = \quad \{(G,H) \mid G \text{ and } H \text{ are isomorphic trees}\}\ .$$

Design an efficient algorithm for this problem.

Hint. Label the vertices of the given pair of trees successively by suitable number sequences. Compare the resulting sequences of labels in the single loops of the algorithm. Starting from the leaves of the trees and then working step by step towards the center of the trees, the algorithm halts as soon as all vertices are labelled. This algorithm can be found, e.g., in [32].

7-3 Computing the determinant

Design an efficient algorithm in pseudocode for computing the determinant of a matrix. Implement your algorithm in a programming language of your choice. Can the inverse of a matrix be computed efficiently?

7-4 Low-exponent attack

a. Consider the RSA cryptosystem from Figure 7.8. For the sake of efficiency, the public exponent $e = 3$ has been popular. However, this choice is dangerous. Suppose Alice, Bob, and Chris encrypt the same message m with the same public exponent $e = 3$, but perhaps with distinct moduli, n_A, n_B, and n_C. Erich intercepts the resulting three ciphertexts: $c_i = m^3 \bmod n_i$ for $i \in \{A, B, C\}$. Then, Erich can easily decrypt the message m. How?

Hint. Erich knows the Chinese remainder theorem, which also was useful in Exercise 7.3-1.

A recommended value for the public exponent is $e = 2^{16} + 1$, since its binary expansion has only two ones. Thus, the square-and-multiply algorithm runs fast for this e.

b. The attack described above can be extended to k ciphertexts that are related with each other as follows. Let a_i and b_i be known, $1 \leq i \leq k$, and suppose that k messages $c_i = (a_i m + b_i)^e \bmod n_i$ are sent and are intercepted by Erich. Further, suppose that $k > e(e+1)/2$ and $\min(n_i) > 2^{e^2}$. How can attacker Erich now determine the original message m?

Hint. Apply so-called lattice reduction techniques (see, e.g., Micciancio and Goldwasser [39]). The attack mentioned here is due to Håstad [22] and has been strengthened later by Coppersmith [12].

c. How can the attacks described above be prevented?

Chapter Notes

Singh's book [61] gives a nice introduction to the history of cryptology, from its ancient roots to modern cryptosystems. For example, you can find out there about recent discoveries related to the development of RSA and Diffie-Hellman in the nonpublic sector. Ellis, Cocks, and Williamson from the Communications Electronics Security Group (CESG) of the British Government Communications Head Quarters (GCHQ) proposed the RSA system from Figure 7.8 and the Diffie-Hellman protocol from Figure 7.7 even earlier than Rivest, Shamir, and Adleman and at about the same time as but independent of Diffie and Hellman, respectively. RSA and Diffie-Hellman are described in probably every book about cryptography written since their invention. A more comprehensive list of attacks against RSA than that of Section 7.3 can be found in, e.g., [7, 30, 42, 50, 52, 60].

Primality tests such as the Miller-Rabin test and factoring algorithms are also described in many books, e.g., in [18, 52, 53, 63].

The notion of strongly noninvertible associative one-way functions, on which the secret-key agreement protocol from Figure 7.11 is based, is due to Rivest and Sherman. The modification of this protocol to a digital signature scheme is due to Rabi and Sherman. In their paper [46], they also proved that commutative, associative one-way function exist if and only if $\mathrm{P} \neq \mathrm{NP}$. However, the one-way functions they construct are neither total nor strongly noninvertible, even if $\mathrm{P} \neq \mathrm{NP}$ is assumed. Hemaspaandra and Rothe [26] proved that total, strongly noninvertible, commutative, associative one-way functions exist if and only if $\mathrm{P} \neq \mathrm{NP}$. Further investigations on this topic can be found in [6, 23, 25, 28].

The notion of interactive proof systems and zero-knowledge protocols is due to Goldwasser, Micali, and Rackoff [20]. One of the best and most comprehensive sources on this field is Chapter 4 in Goldreich's book [18]; see also the books [34, 43, 52] and the surveys [19, 17, 50]. Arthur-Merlin games were introduced by Babai and Moran [5, 4] and have been investigated in many subsequent papers. Variants of the notion of zero-knowledge, which differ from the notion in Definition 7.17 in their technical details, are extensively discussed in, e.g., [18] and also in, e.g., [17, 19, 50].

8. Complexity Theory

In Chapter 7, efficient algorithms were introduced that are important for cryptographic protocols. Designing efficient algorithms of course is a central task in all areas of computer science. Unfortunately, many important problems have resisted all attempts in the past to devise efficient algorithms solving them. Well-known examples of such problems are the satisfiability problem for boolean formulas and the graph isomorphism problem.
One of the most important tasks in complexity theory is to classify such problems according to their computational complexity. Complexity theory and algorithmics are the two sides of the same medal; they complement each other in the following sense. While in algorithmics one seeks to find the best *upper bound* for some problem, an important goal of complexity theory is to obtain the best possible *lower bound* for the same problem. If the upper and the lower bound coincide, the problem has been classified.

The proof that some problem cannot be solved efficiently often appears to be "negative" and not desirable. After all, we wish to solve our problems and we wish to solve them fast. However, there is also some "positive" aspect of proving lower bounds that, in particular, is relevant in cryptography (see Chapter 7). Here, we are interested in the applications of inefficiency: A proof that certain problems (such as the factoring problem or the discrete logarithm) cannot be solved efficiently can support the security of some cryptosystems that are important in practical applications.

In Section 8.1, we provide the foundations of complexity theory. In particular, the central complexity classes P and NP are defined. The question of whether or not these two classes are equal is still open after more than three decades of intense research. Up to now, neither a proof of the inequality $\mathrm{P} \neq \mathrm{NP}$ (which is widely believed) could be achieved, nor were we able to prove the equality of P and NP. This question led to the development of the beautiful and useful theory of NP-completeness.

One of the best understood NP-complete problems is SAT, the satisfiability problem of propositional logic: Given a boolean formula φ, does there exist a satisfying assignment for φ, i.e., does there exist an assignment of truth values to φ's variables that makes φ true? Due to its NP-completeness, it is very unlikely that there exist efficient deterministic algorithms for SAT. In Section 8.3, we present a deterministic and a randomised algorithm for SAT that both run in exponential time. Even though these algorithms are *asymptotically inefficient* (which is to say that they are useless in practice for *large* inputs), they are useful for sufficiently small inputs of sizes still relevant in practice. That is, they outperform the naive deterministic exponential-time algorithm for SAT in that they considerably increase the input size for which the algorithm's running time still is tolerable.

In Section 8.4, we come back to the graph isomorphism problem, which was introduced in Section 7.1.3 (see Definition 7.8) and which was useful in Section 7.5.2 with regard to the zero-knowledge protocols. This problem is one of the few natural problems in NP, which (under the plausible assumption that $\mathrm{P} \neq \mathrm{NP}$) may be neither efficiently solvable nor be NP-complete. In this regard, this problem is special among the problems in NP. Evidence for the hypothesis that the graph isomorphism problem may be neither in P nor NP-complete comes from the theory of *lowness*, which is introduced in Section 8.4. In particular, we present Schöning's result that GI is contained in the low hierarchy within NP. This result provides strong evidence that GI is not NP-complete. We also show that GI is contained in the complexity class SPP and thus is low for certain probabilistic complexity classes. Informally put, a set is *low* for a complexity class $\mathcal{C}$ if it does not provide any useful information when used as an "oracle" in $\mathcal{C}$ computations. For proving the lowness of GI, certain group-theoretic algorithms are useful.

8.1. Foundations

As mentioned above, complexity theory is concerned with proving lower bounds. The difficulty in such proofs is that it is not enough to analyse the runtime of just *one* concrete algorithm for the problem considered. Rather, one needs to show that *every* algorithm solving the problem has a runtime no better than the lower bound to be proven. This includes also algorithms that have not been found as yet. Hence, it is necessary to give a formal and mathematically precise definition of the notion of algorithm.

Since the 1930s, a large variety of formal algorithm models has been proposed. All these models are equivalent in the sense that each such model can be transformed (via an algorithmic procedure) into any other such model. Loosely speaking, one might consider this transformation as some sort of compilation between distinct programming languages. The equivalence of all these algorithm models justifies Church's thesis, which says that each such model captures precisely the somewhat vague notion of "intuitive computability". The algorithm model that is most common in complexity theory is the Turing machine, which was introduced in 1936 by Alan Turing (1912 until 1954) in his pathbreaking work [65]. The Turing machine is a very simple abstract model of a computer. In what follows, we describe this model by defining its syntax and its semantics, and introduce at the same time two basic computation paradigms: determinism and nondeterminism. It makes sense to first define the more general model of nondeterministic Turing machines. Deterministic Turing machines then are easily seen to be a special case.

First, we give some technical details and describe how Turing machines work. A

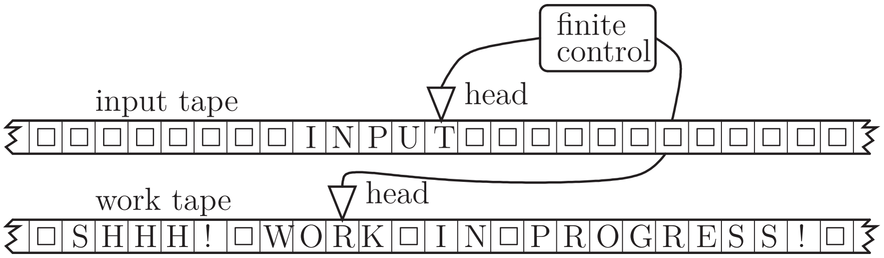
finite control
head
input tape
□□□□□□□□ INPUT □□□□□□□□□□□□
head
work tape
□ SHHH! □ WORK □ IN □ PROGRESS! □

Figure 8.1 A Turing machine with one input and two work tapes.

Turing machine has k infinite work tapes that are subdivided into cells. Every cell may contain one letter of the work alphabet. If a cell does not contain a letter, we indicate this by a special blank symbol, denoted by $\square$. The computation is done on the work tapes. Initially, a designated input tape contains the input string, and all other cells contain the blank. If a computation halts, its result is the string contained in the designated output tape.[1] To each tape belongs a head that accesses exactly one cell of this tape. In each step of the computation, the head can change the symbol currently read and then moves to the left or to the right or does not move at all. At the same time, the current state of the machine, which is stored in its "finite control" can change. Figure 8.1 displays a Turing machine with one input and two work tapes.

Definition 8.1 (Syntax of Turing machines). *A **nondeterministic Turing machine with k tapes** (a k-tape NTM, for short) is a 7-tuple $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$, where Σ is the input alphabet, Γ is the work alphabet, Z is a finite* set of states *disjoint with Γ, $\delta : Z \times \Gamma^k \to \mathfrak{P}(Z \times \Gamma^k \times \{L, R, N\}^k)$ is the* transition function, *$z_0 \in Z$ is the* initial state, *$\square \in \Gamma - \Sigma$ is the* blank symbol, *and $F \subseteq Z$ is the* set of final states. *Here, $\mathfrak{P}(S)$ denotes the* power set *of set S, i.e., the set of all subsets of S.*

For readability, we write $(z, a) \mapsto (z', b, x)$ instead of $(z', b, x) \in \delta(z, a)$ with $z, z' \in Z$, $x \in \{L, R, N\}$ and $a, b \in \Gamma$. This transition has the following meaning. If the current state is z and the head currently reads a symbol a, then:

- *a is replaced by b,*
- *z' is the new state, and*
- *the head moves according to $x \in \{L, R, N\}$, i.e., the head either moves one cell to the left (if $x = L$), or one cell to the right (if $x = R$), or it does not move at all (if $x = N$).*

The special case of a deterministic Turing machine with k tapes *(k-tape DTM, for short) is obtained by requiring that the transition function δ maps from $Z \times \Gamma^k$ to $Z \times \Gamma^k \times \{L, R, N\}^k$.*

For $k = 1$, we obtain the one-tape Turing machine, abbreviated simply by NTM

[1] One can require, for example, that the input tape is a read-only and the output tape is a write-only tape. Similarly, one can specify a variety of further variations of the technical details, but we do not pursue this any further here.

and DTM, respectively. Every k-tape NTM or k-tape DTM can be simulated by a Turing machine with only one tape, where the runtime at most doubles. Turing machines can be considered both as acceptors (which accept languages) and as transducers (which compute functions).

Definition 8.2 (Semantics of Turing machines). *Let $M = (\Sigma, \Gamma, Z, \delta, z_0, \Box, F)$ be an NTM. A* configuration of M *is a string $k \in \Gamma^* Z \Gamma^*$, where $k = \alpha z \beta$ is interpreted as follows: $\alpha\beta$ is the current tape inscription, the head reads the first symbol of β, and z is the current state of M.*

On the set $\mathfrak{K}_M = \Gamma^* Z \Gamma^*$ of all configurations of M, define a binary relation $\vdash_M$, which describes the transition from a configuration $k \in \mathfrak{K}_M$ into a configuration $k' \in \mathfrak{K}_M$ according to δ. For any two strings $\alpha = a_1 a_2 \cdots a_m$ and $\beta = b_1 b_2 \cdots b_n$ in Γ^*, where $m \geq 0$ and $n \geq 1$, and for all $z \in Z$, define

$$\alpha z \beta \vdash_M \begin{cases} a_1 a_2 \cdots a_m z' c b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, N) \text{ and } m \geq 0 \text{ and } n \geq 1 \\ a_1 a_2 \cdots a_m c z' b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, R) \text{ and } m \geq 0 \text{ and } n \geq 2 \\ a_1 a_2 \cdots a_{m-1} z' a_m c b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, L) \text{ and } m \geq 1 \text{ and } n \geq 1 \end{cases} .$$

Two special cases need be considered separately:

1. *If $n = 1$ and $(z, b_1) \mapsto (z', c, R)$ (i.e., M's head moves to the right and reads a $\Box$ symbol), then $a_1 a_2 \cdots a_m z b_1 \vdash_M a_1 a_2 \cdots a_m c z' \Box$.*
2. *If $m = 0$ and $(z, b_1) \mapsto (z', c, L)$ (i.e., M's head moves to the left and reads a $\Box$ symbol), then $z b_1 b_2 \cdots b_n \vdash_M z' \Box c b_2 \cdots b_n$.*

The initial configuration of M on input x *is always $z_0 x$. The* final configurations of M on input x *have the form $\alpha z \beta$ with $z \in F$ and $\alpha, \beta \in \Gamma^*$.*

Let $\vdash_M^*$ be the reflexive, transitive closure of $\vdash_M$: For $k, k' \in \mathfrak{K}_M$, we have $k \vdash_M^* k'$ if and only if there is a finite sequence $k_0, k_1, \ldots, k_t$ of configurations in $\mathfrak{K}_M$ such that

$$k = k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t = k' ,$$

where possibly $k = k_0 = k_t = k'$. If $k_0 = z_0 x$ is the initial configuration of M on input x, then this sequence of configurations is a finite computation of $M(x)$, *and we say* M halts on input x. *The* language accepted by M *is defined by*

$$L(M) = \{x \in \Sigma^* \mid z_0 x \vdash_M^* \alpha z \beta \text{ with } z \in F \text{ and } \alpha, \beta \in \Gamma^*\} .$$

For NTMs, any configuration may be followed by more than one configuration. Thus, they have a *computation tree*, whose root is labelled by the initial configuration and whose leaves are labelled by the final configurations. Note that trees are special graphs (recall Definition 7.8 in Section 7.1.3 and Problem 7-2), so they have vertices and edges. The vertices of a computation tree $M(x)$ are the configurations of M on input x. For any two configurations k and k' from $\mathfrak{K}_M$, there is exactly one directed edge from k to k' if and only if $k \vdash_M k'$. A path in the computation tree of $M(x)$ is a sequence of configurations $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t \vdash_M \cdots$. The computation tree of an NTM can have infinite paths on which never a halting configuration is reached. For DTMs, each non-halting configuration has a unique successor configuration. Thus, the computation tree of a DTM degenerates to a linear chain.

$(s_0, a) \mapsto (s_1, \$, R)$	$(s_2, \$) \mapsto (s_2, \$, R)$	$(s_5, c) \mapsto (s_5, c, L)$
$(s_1, a) \mapsto (s_1, a, R)$	$(s_3, c) \mapsto (s_3, c, R)$	$(s_5, \$) \mapsto (s_5, \$, L)$
$(s_1, b) \mapsto (s_2, \$, R)$	$(s_3, \square) \mapsto (s_4, \square, L)$	$(s_5, b) \mapsto (s_5, b, L)$
$(s_1 \$) \mapsto (s_1, \$, R)$	$(s_4, \$) \mapsto (s_4, \$, L)$	$(s_5, a) \mapsto (s_5, a, L)$
$(s_2, b) \mapsto (s_2, b, R)$	$(s_4, \square) \mapsto (s_6, \square, R)$	$(s_5, \square) \mapsto (s_0, \square, R)$
$(s_2, c) \mapsto (s_3, \$, R)$	$(s_4, c) \mapsto (s_5, c, L)$	$(s_0, \$) \mapsto (s_0, \$, R)$

Figure 8.2 Transition function δ of M for $L = \{a^n b^n c^n \mid n \geq 1\}$.

Z	Meaning	Intention
s_0	initial state	start next cycle
s_1	one a stored	look for next b
s_2	one a and one b stored	look for next c
s_3	one a, one b, and one c deleted	look for right boundary
s_4	right boundary reached	move back and test if all a, b, and c are deleted
s_5	test not successful	move back and start next cycle
s_6	test successful	accept

Figure 8.3 M's states, their meaning and their intention.

Example 8.1 Consider the language $L = \{a^n b^n c^n \mid n \geq 1\}$. A Turing machine accepting L is given by

$$M = (\{a, b, c\}, \{a, b, c, \$, \square\}, \{s_0, s_1, \ldots, s_6\}, \delta, s_0, \square, \{s_6\}) ,$$

where the transitions of δ are stated in Figure 8.2. Figure 8.3 provides the meaning of the single states of M and the intention corresponding to the each state. See also Exercise 8.1-2.

In order to classify problems according to their computational complexity, we need to define complexity classes. Each such class is defined by a given resource function and contains all problems that can be solved by a Turing machine that requires no more of a resource (e.g., computation time or memory space) than is specified by the resource function. We consider only the resource time here, i.e., the number of steps—as a function of the input size—needed to solve (or to accept) the problem. Further, we consider only the traditional *worst-case* complexity model. That is, among all inputs of size n, those that require the maximum resource are decisive; one thus assumes the worst case to occur. We now define deterministic and nondeterministic time complexity classes.

Definition 8.3 (Deterministic and nondeterministic time complexity).

- *Let M be a DTM with $L(M) \subseteq \Sigma^*$ and let $x \in \Sigma^*$ be an input. Define the* time function of $M(x)$, *which maps from Σ^* to $\mathbb{N}$, as follows:*

$$\mathrm{Time}_M(x) = \begin{cases} k & \text{if } M(x) \text{ has exactly } k+1 \text{ configurations} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Define the function $\mathrm{time}_M : \mathbb{N} \to \mathbb{N}$ *by:*

$$\mathrm{time}_M(n) = \begin{cases} \max_{x:|x|=n} \mathrm{Time}_M(x) & \textit{if } \mathrm{Time}_M(x) \textit{ is defined} \\ & \quad \textit{for all } x \textit{ with } |x| = n \\ \textit{undefined} & \textit{otherwise} \ . \end{cases}$$

- *Let M be an NTM with $L(M) \subseteq \Sigma^*$ and let $x \in \Sigma^*$ be an input. Define the* time function of $M(x)$, *which maps from Σ^* to $\mathbb{N}$, as follows:*

$$\mathrm{NTime}_M(x) = \begin{cases} \min\{\mathrm{Time}_M(x,\alpha) \mid M(x) \textit{ accepts on path } \alpha\} & \textit{if } x \in L(M) \\ \textit{undefined} & \textit{otherwise} \ . \end{cases}$$

Define the function $\mathrm{ntime}_M : \mathbb{N} \to \mathbb{N}$ *by*

$$\mathrm{ntime}_M(n) = \begin{cases} \max_{x:|x|=n} \mathrm{NTime}_M(x) & \textit{if } \mathrm{NTime}_M(x) \textit{ is defined} \\ & \quad \textit{for all } x \textit{ with } |x| = n \\ \textit{undefined} & \textit{otherwise} \ . \end{cases}$$

- *Let t be a computable function that maps from $\mathbb{N}$ to $\mathbb{N}$. Define the* deterministic and nondeterministic complexity classes with time function t *by*

$$\mathrm{DTIME}(t) = \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ for some DTM } M \textit{ and} \\ \textit{for all } n \in \mathbb{N}, \mathrm{time}_M(n) \le t(n) \end{array} \right\} ;$$

$$\mathrm{NTIME}(t) = \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ for an NTM } M \textit{ and} \\ \textit{for all } n \in \mathbb{N} \textit{ is } \mathrm{ntime}_M(n) \le t(n) \end{array} \right\} .$$

- *Let* $\mathbb{P}\mathrm{ol}$ *be the set of all polynomials. Define the complexity classes* P *and* NP *as follows:*

$$\mathrm{P} = \bigcup_{t \in \mathbb{P}\mathrm{ol}} \mathrm{DTIME}(t) \qquad \textit{and} \qquad \mathrm{NP} = \bigcup_{t \in \mathbb{P}\mathrm{ol}} \mathrm{NTIME}(t) \ .$$

Why are the classes P and NP so important? Obviously, exponential-time algorithms cannot be considered efficient in general. Garey and Johnson compare the rates of growth of some particular polynomial and exponential time functions $t(n)$ for certain input sizes relevant in practice, see Figure 8.4. They assume that a computer executes one million operations per second. Then all algorithms bounded by a polynomial run in a "reasonable" time for inputs of size up to $n = 60$, whereas for example an algorithm with time bound $t(n) = 3^n$ takes more than 6 years already for the modest input size of $n = 30$. For $n = 40$ it takes almost 4000 centuries, and for $n = 50$ a truly astronomic amount of time.

The last decades have seen an impressive development of computer and hardware technology. Figure 8.5 (taken from [15]) shows that this is not enough to provide an essentially better runtime behaviour for exponential-time algorithms, even assuming that the previous trend in hardware development continues. What would happen if one had a computer that is 100 times or even 1000 times as fast as current computers

$t(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	.00001 sec	.00002 sec	.00003 sec	.00004 sec	.00005 sec	.00006 sec
n^2	.0001 sec	.0004 sec	.0009 sec	.0016 sec	.0025 sec	.0036 sec
n^3	.001 sec	.008 sec	.027 sec	.064 sec	.125 sec	.256 sec
n^5	.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min	13.0 min
2^n	.001 sec	1.0 sec	17.9 min	12.7 days	35.7 years	366 cent.
3^n	.059 sec	58 min	6.5 years	3855 cent.	$2 \cdot 10^8$ cent.	$1.3 \cdot 10^{13}$ cent.

Figure 8.4 Comparison of some polynomial and exponential time functions.

$t_i(n)$	Computer today	100 times faster	1000 times faster
$t_1(n) = n$	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
$t_2(n) = n^2$	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
$t_3(n) = n^3$	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
$t_4(n) = n^5$	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
$t_5(n) = 2^n$	N_5	$N_5 + 6.64$	$N_5 + 9.97$
$t_6(n) = 3^n$	N_6	$N_6 + 4.19$	$N_6 + 6.29$

Figure 8.5 What happens when the computers get faster?

are? For functions $t_i(n)$, $1 \leq i \leq 6$, let N_i be the maximum size of inputs that can be solved by a $t_i(n)$ time-bounded algorithm within one hour. Figure 8.5 also taken from [15]) shows that a computer 1000 times faster than today's computers increases N_5 for $t_5(n) = 2^n$ by only an additive value close to 10. In contrast, using computers with the same increase in speed, an n^5 time-bounded algorithm can handle problem instances about four times as large as before.

Intuitively, the complexity class P contains the efficiently solvable problems. The complexity class NP contains many problems important in practice but currently not efficiently solvable. Examples are the satisfiability problem and the graph isomorphism problem that will be dealt with in more detail later in this chapter. The question of whether or not the classes P and NP are equal is still open. This famous P-versus-NP question gave rise to the theory of NP-completeness, which is briefly introduced in Section 8.2.

Exercises

8.1-1 Can Church's thesis ever be proven formally?

8.1-2 Consider the Turing machine M in Example 8.1.

a. What are the sequences of configurations of M for inputs $x = a^3b^3c^2$ and $y = a^3b^3c^3$, respectively?

b. Prove that M is correct, i.e., show that $L(M) = \{a^nb^nc^n \mid n \geq 1\}$.

c. Estimate the running time of M.

d. Show that the graph isomorphism problem and the graph automorphism problem introduced in Definition 7.8 are both in NP.

8.2. NP-completeness

The theory of NP-completeness provides methods to prove lower bounds for problems in NP. An NP problem is said to be complete in NP if it belongs to the hardest problems in this class, i.e., if it is at least as hard as any NP problem. The complexity of two given problems can be compared by polynomial-time reductions. Among the different types of reduction one can consider, we focus on the *polynomial-time many-one reducibility* in this section. In Section 8.4, more general reducibilities will be introduced, such as the *polynomial-time Turing reducibility* and the *polynomial-time (strong) nondeterministic Turing reducibility.*

Definition 8.4 (Reducibility, NP-Completeness). *A set A is* reducible *to a set B (in symbols, $A \leq_{\mathrm{m}}^{\mathrm{p}} B$) if and only if there exists a polynomial-time computable function r such that for all $x \in \Sigma^*$, $x \in A \iff r(x) \in B$. A set B is said to be* $\leq_{\mathrm{m}}^{\mathrm{p}}$-hard for NP *if and only if $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ for each set $A \in \mathrm{NP}$. A set B is said to be* $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete in NP *(*NP-complete*, for short) if and only if B is $\leq_{\mathrm{m}}^{\mathrm{p}}$-hard for* NP *and $B \in \mathrm{NP}$.*

Reductions are efficient algorithms that can be used to show that problems are not efficiently solvable. That is, if one can efficiently transform a hard problem into another problem via a reduction, the hardness of the former problem is inherited by the latter problem. At first glance, it might seem that infinitely many efficient algorithms are required to prove some problem X NP-hard, namely one reduction from each of the infinitely many NP problems to X. However, an elementary result says that it is enough to find just one such reduction, from *some* NP-complete problem V. Since the $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility is transitive (see Exercise 8.2-2), the NP-hardness of V implies the NP-hardness of X via the reduction $A \leq_{\mathrm{m}}^{\mathrm{p}} V \leq_{\mathrm{m}}^{\mathrm{p}} X$ for each NP problem A.

In 1971, Stephen Cook found a first such NP-complete problem: the satisfiability problem of propositional logic, `SAT` for short. For many NP-completeness result, it is useful to start from the special problem `3-SAT`, the restriction of the satisfiability problem in which each given Boolean formula is in conjunctive normal form and each clause contains exactly three literals. `3-SAT` is also NP-complete.

Definition 8.5 (Satisfiability problem). *The Boolean constants* false *and* true *are represented by* 0 *and* 1*. Let $x_1, x_2, \ldots, x_m$ be Boolean variables, i.e., $x_i \in \{0,1\}$ for each i. Variables and their negations are called* literals*. A Boolean formula φ is* satisfiable *if and only if there is an assignment to the variables in φ that makes the formula true. A Boolean formula φ is in* conjunctive normal form *(*CNF*, for short) if and only if φ is of the form $\varphi(x_1, x_2, \ldots, x_m) = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k_i} \ell_{i,j} \right)$, where the $\ell_{i,j}$ are literals over $\{x_1, x_2, \ldots, x_m\}$. The disjunctions $\bigvee_{j=1}^{k_i} \ell_{i,j}$ of literals are called the* clauses *of φ. A Boolean formula φ is in k-*CNF *if and only if φ is in CNF and each clause of φ contains exactly k literals. Define the following two problems:*

$$\begin{aligned} \mathit{SAT} &= \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula in CNF}\} ; \\ \texttt{3-SAT} &= \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula in 3-CNF}\} . \end{aligned}$$

Example 8.2[Boolean formulas] Consider the following two satisfiable Boolean formulas (see also Exercise 8.2-1):

$$\begin{aligned}\varphi(w,x,y,z) &= (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (w \vee \neg y \vee z) \wedge (\neg w \vee \neg x \vee z);\\ \psi(w,x,y,z) &= (\neg w \vee x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z) \wedge (\neg w \vee y \vee z) \wedge (w \vee \neg x \vee \neg z)\ .\end{aligned}$$

Here, φ is in 3-CNF, so φ is in `3-SAT`. However, ψ is not in 3-CNF, since the first clause contains four literals. Thus, ψ is in `SAT` but not in `3-SAT`.

Theorem 8.6 states the above-mentioned result of Cook.

Theorem 8.6 (Cook). *The problems `SAT` and `3-SAT` are* NP*-complete.*

The proof that `SAT` is NP-complete is omitted. The idea is to encode the computation of an arbitrary NP machine M running on input x into a Boolean formula $\varphi_{M,x}$ such that $\varphi_{M,x}$ is satisfiable if and only if M accepts x.

`SAT` is a good starting point for many other NP-completeness results. In fact, in many cases it is very useful to start with its restriction `3-SAT`. To give an idea of how such proofs work, we now show that `SAT` $\leq_{\mathrm{m}}^{\mathrm{p}}$ `3-SAT`, which implies that `3-SAT` is NP-complete. To this end, we need to find a reduction r that transforms any given Boolean formula φ in CNF into another Boolean formula $\widehat{\varphi}$ in 3-CNF (i.e., with exactly three literals per clause) such that

$$\varphi \text{ is satisfiable} \iff \widehat{\varphi} \text{ is satisfiable}\ . \tag{8.1}$$

Let $\varphi(x_1, x_2, \ldots, x_n)$ be the given formula with clauses $C_1, C_2, \ldots, C_m$. Construct the formula $\widehat{\varphi}$ from φ as follows. The variables of $\widehat{\varphi}$ are

- φ's variables $x_1, x_2, \ldots, x_n$ and
- for each clause C_j of φ, a number of additional variables $y_1^j, y_2^j, \ldots, y_{k_j}^j$ as needed, where k_j depends on the structure of C_j according to the case distinction below.

Now, define $\widehat{\varphi} = \widehat{C}_1 \wedge \widehat{C}_2 \wedge \cdots \wedge \widehat{C}_m$, where each clause $\widehat{C}_j$ of $\widehat{\varphi}$ is constructed from the clause C_j of φ as follows. Suppose that $C_j = (z_1 \vee z_2 \vee \cdots \vee z_k)$, where each z_i is a literal over $\{x_1, x_2, \ldots, x_n\}$. Distinguish the following four cases.

- If $k = 1$, define
$$\widehat{C}_j = (z_1 \vee y_1^j \vee y_2^j) \wedge (z_1 \vee \neg y_1^j \vee y_2^j) \wedge (z_1 \vee y_1^j \vee \neg y_2^j) \wedge (z_1 \vee \neg y_1^j \vee \neg y_2^j)\ .$$
- If $k = 2$, define $\widehat{C}_j = (z_1 \vee z_2 \vee y_1^j) \wedge (z_1 \vee z_2 \vee \neg y_1^j)$.
- If $k = 3$, define $\widehat{C}_j = C_j = (z_1 \vee z_2 \vee z_3)$, i.e., the jth clause remains unchanged.
- If $k \geq 4$, define
$$\begin{aligned}\widehat{C}_j &= (z_1 \vee z_2 \vee y_1^j) \wedge (\neg y_1^j \vee z_3 \vee y_2^j) \wedge (\neg y_2^j \vee z_4 \vee y_3^j) \wedge \cdots \wedge\\ &\quad (\neg y_{k-4}^j \vee z_{k-2} \vee y_{k-3}^j) \wedge (\neg y_{k-3}^j \vee z_{k-1} \vee z_k)\ .\end{aligned}$$

It remains to show that (a) the reduction r is polynomial-time computable, and (b) the equivalence (8.1) is true. Both claims are easy to see; the details are left to the reader as Exercise 8.2-4.

Thousands of problems have been proven NP-complete by now. A comprehensive collection can be found in the work of Garey und Johnson [15].

Exercises

8.2-1 Find a satisfying assignment each for the Boolean formulas φ and ψ from Example 8.2.

8.2-2 Show that the $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility is transitive: $(A \leq_{\mathrm{m}}^{\mathrm{p}} B \wedge B \leq_{\mathrm{m}}^{\mathrm{p}} C) \Longrightarrow A \leq_{\mathrm{m}}^{\mathrm{p}} C$.

8.2-3 Prove that `SAT` is in NP.

8.2-4 Consider the reduction `SAT` $\leq_{\mathrm{m}}^{\mathrm{p}}$ `3-SAT`. Prove the following:

a. the reduction r is polynomial-time computable, and

b. the equivalence (8.1) holds.

8.3. Algorithms for the satisfiability problem

By Theorem 8.6, `SAT` and `3-SAT` are both NP-complete. Thus, if `SAT` were in P, it would immediately follow that P $=$ NP, which is considered unlikely. Thus, it is very unlikely that there is a polynomial-time deterministic algorithm for `SAT` or `3-SAT`. But what is the runtime of the best deterministic algorithms for them? And what about randomised algorithms? Let us focus on the problem `3-SAT` in this section.

8.3.1. A deterministic algorithm

The "naive" deterministic algorithm for `3-SAT` works as follows: Given a Boolean formula φ with n variables, sequentially check the 2^n possible assignments to the variables of φ. Accept if the first satisfying assignment is found, otherwise reject. Obviously, this algorithm runs in time $\mathcal{O}(2^n)$. Can this upper bound be improved?

Yes, it can. We will present an slightly better deterministic algorithm for `3-SAT` that still runs in exponential time, namely in time $\tilde{\mathcal{O}}(1.9129^n)$, where the $\tilde{\mathcal{O}}$ notation neglects polynomial factors as is common for exponential-time algorithms.[2] The point of such an improvement is that a $\tilde{\mathcal{O}}(c^n)$ algorithm, where $1 < c < 2$ is a constant, can deal with larger instances than the naive $\tilde{\mathcal{O}}(2^n)$ algorithm in the same amount of time before the exponential growth rate eventually hits and the running time becomes infeasible. For example, if $c = \sqrt{2} \approx 1.414$ then $\tilde{\mathcal{O}}\left(\sqrt{2}^{\,2n}\right) = \tilde{\mathcal{O}}(2^n)$. Thus, this algorithm can deal with inputs twice as large as the naive algorithm in the same amount of time. Doubling the size of inputs that can be handled by some algorithm can be quite important in practice.

The deterministic algorithm for `3-SAT` is based on a simple "*backtracking*" idea. Backtracking is useful for problems whose solutions consist of n components each having more than one choice possibility. For example, a solution of `3-SAT` is composed of the n truth values of a satisfying assignment, and each such truth value can be

[2] The result presented here is not the best result known, but see Figure 8.7 on page 393 for further improvements.

either true (represented by 1) or false (represented by 0).

The idea is the following. Starting from the initially empty solution (i.e., the empty truth assignment), we seek to construct by recursive calls to our backtracking algorithm, step by step, a larger and larger partial solution until eventually a complete solution is found, if one exists. In the resulting recursion tree,[3] the root is marked by the empty solution, and the leaves are marked with complete solutions of the problem. If the current branch in the recursion tree is "dead" (which means that the subtree underneath it cannot lead to a correct solution), one can prune this subtree and "backtracks" to pursue another extention of the partial solution constructed so far. This pruning may save time in the end.

```
Backtracking-SAT($\varphi, \beta$)
1  if ($\beta$ assigns truth values to all variables of $\varphi$)
2     then return $\varphi(\beta)$
3     else  if ($\beta$ makes one of the clauses of $\varphi$ false)
4              then return 0
5     ‘ ▷ "dead branch"
6                      else if Backtracking-SAT($\varphi, \beta 0$))
7                            then return 1
8                            else  return Backtracking-SAT($\varphi, \beta 1$))
```

The input of algorithm Backtracking-SAT are a Boolean formula φ and a partial assignment β to some of the variables of φ. This algorithm returns a Boolean value: 1, if the partial assignment β can be extended to a satisfying assignment to all variables of φ, and 0 otherwise. Partial assignments are here considered to be strings of length at most n over the alphabet $\{0, 1\}$.

The first call of the algorithm is Backtracking-SAT(φ, λ), where λ denotes the empty assignment. If it turns out that the partial assignment constructed so far makes one of the clauses of φ false, it cannot be extended to a satisfying assignment of φ. Thus, the subtree underneath the corresponding vertex in the recursion tree can be pruned; see also Exercise 8.3-1.

To estimate the runtime of Backtracking-SAT, note that this algorithm can be specified so as to select the variables in an "intelligent" order that minimises the number of steps needed to evaluate the variables in any clause. Consider an arbitrary, fixed clause C_j of the given formula φ. Each satisfying assignment β of φ assigns truth values to the three variables in C_j. There are $2^3 = 8$ possibilities to assign a truth value to these variables, and one of them can be excluded certainly: the assignment that makes C_j false. The corresponding vertex in the recursion tree of Backtracking-SAT(φ, β) thus leads to a "dead" branch, so we prune the subtree underneath it.

Depending on the structure of φ, there may exist further "dead" branches whose subtrees can also be pruned. However, since we are trying to find an upper bound in

[3] The inner vertices of the recursion tree represent the recursive calls of the algorithm, its root is the first call, and the algorithm terminates at the leaves without any further recursive call.

the worst case, we do not consider these additional "dead" subtrees. It follows that

$$\tilde{\mathcal{O}}\left((2^3 - 1)^{\frac{n}{3}}\right) = \tilde{\mathcal{O}}(\sqrt[3]{7}^n) \approx \tilde{\mathcal{O}}(1.9129^n)$$

is an upper bound for Backtracking-SAT in the worst case. This bound slightly improves upon the trivial $\tilde{\mathcal{O}}(2^n)$ upper bound of the "naive" algorithm for 3-SAT.

As mentioned above, the deterministic time complexity of 3-SAT can be improved even further. For example, Monien and Speckenmeyer [41] proposed a divide-and-conquer algorithm with runtime $\tilde{\mathcal{O}}(1.618^n)$. Dantsin et al. [13] designed a deterministic "local search with restart" algorithm whose runtime is $\tilde{\mathcal{O}}(1.481^n)$, which was further improved by Brueggemann and Kern [9] in 2004 to a $\tilde{\mathcal{O}}(1.4726^n)$ bound.

There are also randomised algorithms that have an even better runtime. One will be presented now, a "random-walk" algorithm that is due to Schöning [56].

8.3.2. A randomised algorithm

A *random walk* can be done on a specific structure, such as in the Euclidean space, on an infinite grid or on a given graph. Here we are interested in random walks occurring on a graph that represents a certain stochastic automaton. To describe such automata, we first introduce the notion of a finite automaton.

A *finite automaton* can be represented by its transition graph, whose vertices are the states of the finite automaton, and the transitions between states are directed edges marked by the symbols of the alphabet Σ. One designated vertex is the *initial state* in which the computation of the automaton starts. In each step of the computation, the automaton reads one input symbol (proceeding from left to right) and moves to the next state along the edge marked by the symbol read. Some vertices represent *final states*. If such a vertex is reached after the entire input has been read, the automaton accepts the input, and otherwise it rejects. In this way, a finite automaton accepts a set of input strings, which is called its language.

A *stochastic automaton* $\mathcal{S}$ is a finite automaton whose edges are marked by probabilities in addition. If the edge from u to v in the transition graph of $\mathcal{S}$ is marked by $p_{u,v}$, where $0 \leq p_{u,v} \leq 1$, then $\mathcal{S}$ moves from state u to state v with probability $p_{u,v}$. The process of random transitions of a stochastic automaton is called a *Markov chain* in probability theory. Of course, the acceptance of strings by a stochastic automaton depends on the transition probabilities.

```
Random-SAT($\varphi$)
1  for $i \leftarrow 1$ to $\lceil (4/3)^n \rceil$)
                                        $\triangleright$ $n$ is the number of variables in $\varphi$
2      do randomly choose an assignment $\beta \in \{0,1\}^n$
          under the uniform distribution
```

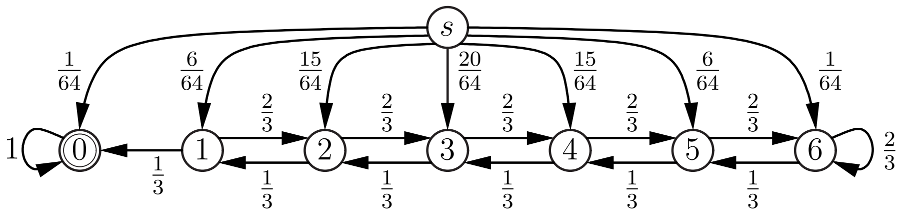

Figure 8.6 Transition graph of a stochastic automaton for describing RANDOM-SAT.

```
3        for $j \leftarrow 1$ to $n$
4            if $(\varphi(\beta) = 1)$
5                then return the satisfying assignment $\beta$ to $\varphi$
6                else   choose a clause $C = (x \vee y \vee z)$ with $C(\beta) = 0$
7                       randomly choose a literal $\ell \in \{x, y, z\}$
                        under the uniform distribution
8                       determine the bit $\beta_\ell \in \{0, 1\}$ in $\beta$ assigning $\ell$
9                       swap $\beta_\ell$ to $1 - \beta_\ell$ in $\beta$
10   return "$\varphi$ is not satisfiable"
```

Here, we are not interested in recognising languages by stochastic automata, but rather we will use them to describe a random walk by the randomised algorithm RANDOM-SAT. Given a Boolean formula φ with n variables, RANDOM-SAT tries to find a satisfying assignment to φ's variables, if one exists.

On input φ, RANDOM-SAT starts by guessing a random initial assignment β, where each bit of β takes on the value 0 and 1 with probability $1/2$. Suppose φ is satisfiable. Let $\tilde{\beta}$ be an arbitrary fixed assignment of φ. Let X be a random variable that expresses the *Hamming distance between* β *and* $\tilde{\beta}$, i.e., X gives the number of bits in which β and $\tilde{\beta}$ differ. Clearly, X can take on values $j \in \{0, 1, \dots, n\}$ and is distributed according to the binomial distribution with parameters n and $1/2$. That is, the probability for $X = j$ is exactly $\binom{n}{j}2^{-n}$.

RANDOM-SAT now checks whether the initial assignment β already satisfies φ, and if so, it accepts. Otherwise, if β does not satisfy φ, there must exist a clause in φ not satisfied by β. RANDOM-SAT now picks any such clause, randomly chooses under the uniform distribution some literal in this clause, and "flips" the corresponding bit in the current assignment β. This procedure is repeated n times. If the current assignment β still does not satisfy φ, RANDOM-SAT restarts with a new initial assignment, and repeats this entire procedure t times, where $t = \lceil (4/3)^n \rceil$.

Figure 8.6 shows a stochastic automaton $\mathcal{S}$, whose edges are not marked by symbols but only by transition probabilities. The computation of RANDOM-SAT on input φ can be viewed as a random walk on $\mathcal{S}$ as follows. Starting from the initial state s, which will never be reached again later, RANDOM-SAT(φ) first moves to one of the states $j \in \{0, 1, \dots, n\}$ according to the binomial distribution with parameters n and $1/2$. This is shown in the upper part of Figure 8.6 for a formula with $n = 6$ variables. Reaching such a state j means that the randomly chosen initial assignment β and the fixed satisfying assignment $\tilde{\beta}$ have Hamming distance j. As long as $j \neq 0$, Random-SAT(φ) changes one bit β_ℓ to $1-\beta_\ell$ in the current assignment β, searching for a satisfying assignment in each iteration of the inner `for` loop. In the random walk on $\mathcal{S}$, this corresponds to moving one step to the left to state $j-1$ or moving one step to the right to state $j+1$, where only states less than or equal to n can be reached.

The fixed assignment $\tilde{\beta}$ satisfies φ, so it sets at least one literal in each clause of φ to true. If we fix *exactly* one of the literals satisfied by $\tilde{\beta}$ in each clause, say ℓ, then Random-SAT(φ) makes a step to the left if and only if ℓ was chosen by Random-SAT(φ). Hence, the probability of moving from state $j > 0$ to state $j-1$ is $1/3$, and the probability of moving from state $j > 0$ to state $j+1$ is $2/3$.

If the state $j = 0$ is reached eventually after at most n iterations of this process, β and $\tilde{\beta}$ have Hamming distance 0, so β satisfies φ and Random-SAT(φ) returns β and halts accepting. Of course, one might also hit a satisfying assignment (distinct from $\tilde{\beta}$) in some state $j \neq 0$. But since this would only increase the acceptance probability, we neglect this possibility here.

If this process is repeated n times unsuccessfully, then the initial assignment β was chosen so badly that Random-SAT now dumps it, and restarts the above process from scratch with a new initial assignment. The entire procedure is repeated at most t times, where $t = \lceil (4/3)^n \rceil$. If it is still unsuccessful after t trials, Random-SAT rejects its input.

Since the probability of moving away from state 0 to the right is larger than the probability of moving toward 0 to the left, one might be tempted to think that the success probability of Random-SAT is very small. However, one should not underestimate the chance that one already after the initial step from s reaches a state close to 0. The closer to 0 the random walk starts, the higher is the probability of reaching 0 by random steps to the left or to the right.

We only give a rough sketch of estimating the success probability (assuming that φ is satisfiable) and the runtime of Random-SAT(φ). For convenience, suppose that 3 divides n. Let p_i be the probability for the event that Random-SAT(φ) reaches the state 0 within n steps after the initial step, under the condition that it reaches some state $i \leq n/3$ with the initial step from s. For example, if the state $n/3$ is reached with the initial step and if no more than $n/3$ steps are done to the right, then one can still reach the final state 0 by a total of at most n steps. If one does more than $n/3$ steps to the right starting from state $n/3$, then the final state cannot be reached within n steps. In general, starting from state i after the initial step, no more than $(n-i)/2$ steps to the right may be done. As noted above, a step to the right is done with probability $2/3$, and a step to the left is done with probability $1/3$. It follows that

$$p_i \quad = \quad \binom{n}{\frac{n-i}{2}} \left(\frac{2}{3}\right)^{\frac{n-i}{2}} \left(\frac{1}{3}\right)^{n-\frac{n-i}{2}} . \tag{8.2}$$

Now, let q_i be the probability for the event that Random-SAT(φ) reaches some state $i \leq n/3$ with the initial step. Clearly, we have

$$q_i \quad = \quad \binom{n}{i} \cdot 2^{-n} . \tag{8.3}$$

Finally, let p be the probability for the event that Random-SAT(φ) reaches the final state 0 within the inner `for` loop. Of course, this event can occur also when starting from a state $j > n/3$. Thus,

$$p \geq \sum_{i=0}^{n/3} p_i \cdot q_i \ .$$

Approximating this sum by the entropy function and estimating the binomial coefficients from (8.2) and (8.3) in the single terms by Stirling's formula, we obtain the lower bound $\Omega((3/4)^n)$ for p.

To reduce the error probability, Random-SAT performs a total of t independent trials, each starting with a new initial assignment β. For each trial, the probability of success (i.e., the probability of finding a satisfying assignment of φ, if one exists) is at least $(3/4)^n$, so the error is bounded by $1 - (3/4)^n$. Since the trials are independent, these error probabilities multiply, which gives an error of $\left(1 - (3/4)^n\right)^t \leq \mathrm{e}^{-1}$. Thus, the total probabilitiy of success of Random-SAT(φ) is at least $1 - 1/\mathrm{e} \approx 0.632$ if φ is satisfiable. On the other hand, Random-SAT(φ) does not make any error at all if φ is unsatisfiable; in this case, the output is : "φ is not satisfiable".

The particular choice of this value of t can be explained as follows. The runtime of a randomised algorithm, which performs independent trials such as Random-SAT(φ), is roughly reciprocal to the success probability of one trial, $p \approx (3/4)^n$. In particular, the error probability (i.e., the probability that that in none of the t trials a satisfying assignment of φ is found even though φ is satisfiable) can be estimated by $(1-p)^t \leq e^{-t \cdot p}$. If a fixed error of ε is to be not exceeded, it is enough to choose t such that $e^{-t \cdot p} \leq \varepsilon$; equivalently, such that $t \geq \ln(1/\varepsilon)/p$. Up to constant factors, this can be accomplished by choosing $t = \lceil (4/3)^n \rceil$. Hence, the runtime of the algorithm is in $\tilde{\mathcal{O}}\left((4/3)^n\right)$.

Exercises

8.3-1 Start the algorithm Backtracking-SAT for the Boolean formula $\varphi = (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg u \vee y \vee z) \wedge (u \vee \neg y \vee z)$ and construct step by step a satisfying assignment of φ. How does the resulting recursion tree look like?

8.4. Graph isomorphism and lowness

In this section, we need some of the group-theoretic and graph-theoretic foundations presented in Section 7.1.3. In particular, recall the notion of permutation group from Definition 7.6 and the graph isomorphism problem (`GI`, for short) and the graph automorphism problem (`GA`, for short) from Definition 7.8; see also Example 7.4 in Chapter 7. We start by providing some more background from complexity theory.

8.4.1. Reducibilities and complexity hierarchies

In Section 8.2, we have seen that the problems `SAT` and `3-SAT` are NP-complete. Clearly, P = NP if and only if *every* NP problem (including the NP-complete problems) is in P, which in turn is true if and only if *some* NP-complete problem is in P. So, no NP-complete problem can be in P if P $\neq$ NP. An interesting question is whether, under the plausible assumption that P $\neq$ NP, every NP problem is either in P or NP-complete. Or, assuming P $\neq$ NP, can there exist NP problems that are *neither* in P *nor* NP-complete? A result by Ladner [35] answers this question.

Theorem 8.7 (Ladner). *If* P $\neq$ NP *then there exist sets in* NP *that are neither in* P *nor* NP*-complete.*

The problems constructed in the proof of Theorem 8.7 are not overly natural problems. However, there are also good candidates of "natural" problems that are neither in P nor NP-complete. One such candidate is the graph isomorphism problem. To provide some evidence for this claim, we now define two hierarchies of complexity classes, the *low hierarchy* and the *high hierarchy*, both introduced by Schöning [54]. First, we need to define the *polynomial hierarchy*, which builds upon NP. And to this end, we need a more flexible reducibility than the (polynomial-time) many-one reducibility $\leq_{\mathrm{m}}^{\mathrm{p}}$ from Definition 8.4, namely the *Turing reducibility* $\leq_{\mathrm{T}}^{\mathrm{p}}$. We will also define the (polynomial-time) *nondeterministic Turing reducibility*, $\leq_{\mathrm{T}}^{\mathrm{NP}}$, and the (polynomial-time) *strong nondeterministic Turing reducibility*, $\leq_{\mathrm{sT}}^{\mathrm{NP}}$. These two reducibilities are important for the polynomial hierarchy and for the high hierarchy, respectively. Turing reducibilities are based on the notion of oracle Turing machines, which we now define.

Definition 8.8 (Oracle Turing machine). *An* ***oracle set*** *is a set of strings. An* oracle Turing machine *M, with oracle B, is a Turing machine that has a special worktape, which is called the* oracle tape *or* query tape. *In addition to other states, M contains a special* query state, *$s_?$, and the two* answer states *$s_{\texttt{yes}}$ and $s_{\texttt{no}}$. During a computation on some input, if M is not in the query state $s_?$, it works just like a regular Turing machine. However, when M enters the query state $s_?$, it interrupts its computation and queries its oracle about the string q currently written on the oracle tape. Imagine the oracle B as some kind of "black box": B answers the query of whether or not it contains q within one step of M's computation, regardless of how difficult it is to decide the set B. If $q \in B$, then M changes its current state into the new state $s_{\texttt{yes}}$ and continues its computation. Otherwise, if $q \notin B$, M continues its computation in the new state $s_{\texttt{no}}$. We say that the computation of M on input x* is performed relative to the oracle *B, and we write $M^B(x)$.*

The language accepted by M^B is denoted $L(M^B)$. We say a language L is represented by an oracle Turing machine *M if and only if $L = L(M^{\emptyset})$. We say a class $\mathcal{C}$ of languages is* relativizable *if and only if it can be represented by oracle Turing machines relative to the empty oracle. For any relativizable class $\mathcal{C}$ and for any oracle B, define the class $\mathcal{C}$* relative to *B by*

$$\mathcal{C}^B = \{L(M^B) \mid M \textit{ is an oracle Turing machine representing some set in } \mathcal{C}\} .$$

For any class $\mathcal{B}$ of oracle sets, define $\mathcal{C}^{\mathcal{B}} = \bigcup_{B \in \mathcal{B}} \mathcal{C}^B$.

Let NPOTM (respectively, DPOTM) be a shorthand for *nondeterministic* (respectively, *deterministic*) *polynomial-time oracle Turing machine*. For example, the following classes can be defined:

$$\begin{aligned}
\mathrm{NP}^{\mathrm{NP}} &= \bigcup_{B\in\mathrm{NP}} \mathrm{NP}^B = \{L(M^B) \mid M \text{ is an NPOTM and } B \text{ is in NP}\}\ ;\\
\mathrm{P}^{\mathrm{NP}} &= \bigcup_{B\in\mathrm{NP}} \mathrm{P}^B = \{L(M^B) \mid M \text{ is a DPOTM and } B \text{ is in NP}\}\ .
\end{aligned}$$

For the empty oracle set $\emptyset$, we obtain the unrelativized classes $\mathrm{NP} = \mathrm{NP}^{\emptyset}$ and $\mathrm{P} = \mathrm{P}^{\emptyset}$, and we then write NPTM instead of NPOTM and DPTM instead of DPOTM.

In particular, oracle Turing machines can be used for prefix search. Let us consider an example.

Example 8.3[Prefix search by an oracle Turing machine]

Suppose we wish to find the smallest solution of the graph isomorphism problem, which is in NP; see Definition 7.8 in Subsection 7.1.3. Let G and H be two given graphs with $n \geq 1$ vertices each. An isomorphism between G and H is called a *solution of "$(G,H) \in \mathtt{GI}$"*. The set of isomorphisms between G and H, $\mathrm{Iso}(G,H)$, contains all solutions of "$(G,H) \in \mathtt{GI}$".

Our goal is to find the lexicographically smallest solution if $(G,H) \in \mathtt{GI}$; otherwise, we output the empty string λ to indicate that $(G,H) \notin \mathtt{GI}$. That is, we wish to compute the function f defined by $f(G,H) = \min\{\pi \mid \pi \in \mathrm{Iso}(G,H)\}$ if $(G,H) \in \mathtt{GI}$, and $f(G,H) = \lambda$ if $(G,H) \notin \mathtt{GI}$, where the minimum is to be taken according to the lexicographic order on $\mathfrak{S}_n$. More precisely, we view a permutation $\pi \in \mathfrak{S}_n$ as the string $\pi(1)\pi(2)\cdots\pi(n)$ of length n over the alphabet $[n] = \{1,2,\ldots,n\}$, and we write $\pi < \sigma$ for $\pi,\sigma \in \mathfrak{S}_n$ if and only if there is a $j \in [n]$ such that $\pi(i) = \sigma(i)$ for all $i < j$ and $\pi(j) < \sigma(j)$.

From a permutation $\sigma \in \mathfrak{S}_n$, one obtains a *partial permutation* by cancelling some pairs $(i,\sigma(i))$. A partial permutation can be represented by a string over the alphabet $[n] \cup \{*\}$, where $*$ indicates an undefined position. Let $k \leq n$. A *prefix of length k of $\sigma \in \mathfrak{S}_n$* is a partial permutation of σ containing each pair $(i,\sigma(i))$ with $i \leq k$, but none of the pairs $(i,\sigma(i))$ with $i > k$. In particular, for $k = 0$, the empty string λ is the (unique) length 0 prefix of σ. For $k = n$, the total permutation σ is the (unique) length n prefix of itself. Suppose that π is a prefix of length $k < n$ of $\sigma \in \mathfrak{S}_n$ and that $w = i_1 i_2 \cdots i_{|w|}$ is a string over $[n]$ of length $|w| \leq n-k$ with none of the i_j occurring in π. Then, πw denotes the partial permutation that extends π by the pairs $(k+1,i_1),(k+2,i_2),\ldots,(k+|w|,i_{|w|})$. If in addition $\sigma(k+j) = i_j$ for $1 \leq j \leq |w|$, then πw is also a prefix of σ. For our prefix search, given two graphs G and H, we define the set of prefixes of the isomorphisms in $\mathrm{Iso}(G,H)$ by

$$\text{Pre-Iso} = \left\{ (G,H,\pi) \;\middle|\; \begin{array}{l} G \text{ and } H \text{ are graphs with } n \text{ vertices each and} \\ (\exists w \in [n]^*)\ [w = i_1 i_2 \cdots i_{n-|\pi|} \text{ and } \pi w \in \mathrm{Iso}(G,H)] \end{array} \right\}.$$

Note that, for $n \geq 1$, the empty string λ does not encode a permutation in $\mathfrak{S}_n$. Furthermore, $\mathrm{Iso}(G,H) = \emptyset$ if and only if $(G,H,\lambda) \notin \text{Pre-Iso}$, which in turn is true if and only if $(G,H) \notin \mathtt{GI}$.

Starting from the empty string, we will construct, bit by bit, the smallest isomorphism between the two given graphs (if there exists any). We below present an DPOTM N that, using the NP set Pre-Iso as its oracle, computes the function f by prefix search; see also Exercise 8.4-2. Denoting the class of functions computable in polynomial time by FP, we thus have shown that f is in $\mathrm{FP}^{\text{Pre-Iso}}$. Since Pre-Iso is in NP (see Exercise 8.4-2), it follows that f is in $\mathrm{FP}^{\mathrm{NP}}$.

N-Pre-Iso(G, H)

```
if $((G, H, \lambda) \notin$ Pre-Iso$)$
   then return $\lambda$
   else  $\pi \leftarrow \lambda$
         $j \leftarrow 0$
         while $j < n$                              $\triangleright$ $G$ and $H$ have $n$ vertices each.
             do $i \leftarrow 1$
                while $(G, H, \pi i) \notin$ Pre-Iso
                     do $i \leftarrow i + 1$
                     $\pi \leftarrow \pi i$
                     $j \leftarrow j + 1$
         return $\pi$
```

Example 8.3 shows that also Turing machines computing functions can be equipped with an oracle, and that also function classes such as FP can be relativizable. We now return to oracle machines accepting languages and use them to define several reducibilities. All reducibilities considered here are polynomial-time computable.

Definition 8.9 (Turing reducibilities). *Let $\Sigma = \{0,1\}$ be a binary alphabet, let A and B be sets of strings over Σ, and let $\mathcal{C}$ be any complexity class. The set of complements of sets in $\mathcal{C}$ is defined by* $\mathrm{co}\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}$.

Define the following reducibilities:

- ***Turing reducibility:*** $A \leq_{\mathrm{T}}^{\mathrm{p}} B \iff A = L(M^B)$ *for some DPOTM M.*
- ***Nondeterministic Turing reducibility:*** $A \leq_{\mathrm{T}}^{\mathrm{NP}} B \iff A = L(M^B)$ *for some NPOTM M.*
- ***Strong nondeterministic Turing reducibility:*** $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B \iff A \in \mathrm{NP}^B \cap \mathrm{coNP}^B$.
- *Let $\leq_r$ be one of the reducibilities defined above. We call a set B* $\leq_r$-hard for $\mathcal{C}$ *if and only if $A \leq_r B$ for each set $A \in \mathcal{C}$. A set B is said to be* ***$\leq_r$-complete in $\mathcal{C}$*** *if and only if B is $\leq_r$-hard for $\mathcal{C}$ and $B \in \mathcal{C}$.*
- $\mathrm{P}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\, [A \leq_{\mathrm{T}}^{\mathrm{p}} B]\}$ *is the* ***closure of $\mathcal{C}$ under the $\leq_{\mathrm{T}}^{\mathrm{p}}$-reducibility.***
- $\mathrm{NP}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\, [A \leq_{\mathrm{T}}^{\mathrm{NP}} B]\}$ *is the* ***closure of $\mathcal{C}$ under the $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducibility.***

Using the $\leq_{\mathrm{T}}^{\mathrm{p}}$-reducibility and the $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducibility introduced in Definition 8.9, we now define the polynomial hierarchy, and the low and the high hierarchy within NP.

Definition 8.10 (Polynomial hierarchy). *Define the* ***polynomial hierarchy*** PH *inductively as follows:* $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathrm{P}$, $\Delta_{i+1}^p = \mathrm{P}^{\Sigma_i^p}$, $\Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i^p}$ *and* $\Pi_{i+1}^p = \mathrm{co}\Sigma_{i+1}^p$ *for $i \geq 0$, and* $\mathrm{PH} = \bigcup_{k \geq 0} \Sigma_k^p$.

In particular, $\Delta_1^p = \mathrm{P}^{\Sigma_0^p} = \mathrm{P}^{\mathrm{P}} = \mathrm{P}$ and $\Sigma_1^p = \mathrm{NP}^{\Sigma_0^p} = \mathrm{NP}^{\mathrm{P}} = \mathrm{NP}$ and $\Pi_1^p = \mathrm{co}\Sigma_1^p = \mathrm{coNP}$. The following theorem, which is stated without proof, provides some properties of the polynomial hierarchy, see Problem 8-2.

Theorem 8.11 (Meyer and Stockmeyer). *For alle $i \geq 1$ holds:*

1. $\Sigma_{i-1}^p \cup \Pi_{i-1}^p \subseteq \Delta_i^p \subseteq \Sigma_i^p \cap \Pi_i^p$.
2. Σ_i^p, Π_i^p, Δ_i^p, *and* PH *are closed under* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-reductions.* Δ_i^p *is even closed under* $\leq_{\mathrm{T}}^{\mathrm{p}}$*-reductions.*
3. Σ_i^p *contains exactly those sets A for which there exist a set B in* P *and a polynomial p such that for each $x \in \Sigma^*$:*
$$x \in A \iff (\exists^p w_1)\,(\forall^p w_2)\,\cdots\,(\mathfrak{Q}^p w_i)\,[(x, w_1, w_2, \ldots, w_i) \in B],$$
where the quantifiers $\exists^p$ and $\forall^p$ are polynomially length-bounded, and $\mathfrak{Q}^p = \exists^p$ if i is odd, and $\mathfrak{Q}^p = \forall^p$ if i is even.
4. *If $\Sigma_{i-1}^p = \Sigma_i^p$ for some i, then* PH *collapses to* $\Sigma_{i-1}^p = \Pi_{i-1}^p = \Delta_i^p = \Sigma_i^p = \Pi_i^p = \cdots = \mathrm{PH}$.
5. *If $\Sigma_i^p = \Pi_i^p$ for some i, then* PH *collapses to* $\Sigma_i^p = \Pi_i^p = \Delta_{i+1}^p = \Sigma_{i+1}^p = \Pi_{i+1}^p = \cdots = \mathrm{PH}$.
6. *There are $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete problems for each of the classes Σ_i^p, Π_i^p, and Δ_i^p. In contrast, if* PH *has a $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete problem, then* PH *collapses to a finite level, i.e.,* $\mathrm{PH} = \Sigma_k^p = \Pi_k^p$ *for some k.*

Definition 8.12 (Low hierarchy and high hierarchy within NP). *For $k \geq 0$, define the kth level of the*

- low hierarchy $\mathrm{LH} = \bigcup_{k \geq 0} \mathrm{Low}_k$ in NP *by* $\mathrm{Low}_k = \{L \in \mathrm{NP} \mid \Sigma_k^{p,L} \subseteq \Sigma_k^p\}$;
- high hierarchy $\mathrm{HH} = \bigcup_{k \geq 0} \mathrm{High}_k$ in NP *by* $\mathrm{High}_k = \{H \in \mathrm{NP} \mid \Sigma_{k+1}^p \subseteq \Sigma_k^{p,H}\}$.

Informally put, a set L is in Low_k if and only if it is useless as an oracle for a Σ_k^p computation. All information contained in $L \in \mathrm{Low}_k$ can be computed by the Σ_k^p machine itself without the help of an oracle. On the other hand, a set H in High_k is so rich and provides so useful information that the computational power of a Σ_k^p machine is increased by the maximum amount an NP set can provide: it “jumps” to the next level of the polynomial hierarchy. That is, by using an oracle H from High_k, a Σ_k^p machine can simulate any Σ_{k+1}^p computation. Thus, H is as useful for a Σ_k^p machine as any NP-complete set.

For $k = 0$, the question of whether or not $\Sigma_k^p \neq \Sigma_{k+1}^p$ is nothing other than the P-versus-NP question. Theorem 8.13 lists some important properties of these hierarchies, omitting the proofs, see [54] and Exercise 8-2. For the class coAM mentioned in the first item of Theorem 8.13, the reader is referred to the definition of the Arthur-Merlin hierarchy introduced in Subsection 7.5.1; cf. Definition 7.16. Ladner’s theorem (Theorem 8.7) is a special case (for $n = 0$) of item 7 in Theorem 8.13.

Theorem 8.13 (Schöning).

1. $\mathrm{Low}_0 = \mathrm{P}$ *and* $\mathrm{Low}_1 = \mathrm{NP} \cap \mathrm{coNP}$ *and* $\mathrm{NP} \cap \mathrm{coAM} \subseteq \mathrm{Low}_2$.

2. $\mathrm{High}_0 = \{H \mid H$ *is* $\leq_{\mathrm{T}}^{\mathrm{p}}$*-complete in* $\mathrm{NP}\}$.
3. $\mathrm{High}_1 = \{H \mid H$ *is* $\leq_{\mathrm{sT}}^{\mathrm{NP}}$*-complete in* $\mathrm{NP}\}$.
4. $\mathrm{Low}_0 \subseteq \mathrm{Low}_1 \subseteq \cdots \subseteq \mathrm{Low}_k \subseteq \cdots \subseteq \mathrm{LH} \subseteq \mathrm{NP}$.
5. $\mathrm{High}_0 \subseteq \mathrm{High}_1 \subseteq \cdots \subseteq \mathrm{High}_k \subseteq \cdots \subseteq \mathrm{HH} \subseteq \mathrm{NP}$.
6. *For each* $n \geq 0$, $\mathrm{Low}_n \cap \mathrm{High}_n$ *is nonempty if and only if* $\Sigma_n^p = \Sigma_{n+1}^p = \cdots = \mathrm{PH}$.
7. *For each* $n \geq 0$, NP *contains sets that are neither in* Low_n *nor in* High_n *if and only if* $\Sigma_n^p \neq \Sigma_{n+1}^p$.
8. *There exist sets in* NP *that are neither in* LH *nor in* HH *if and only if* PH *is a strictly infinite hierarchy, i.e., if and only if* PH *does not collapse to a finite level.*

8.4.2. Graph isomorphism is in the low hierarchy

We now turn to the result that the graph isomorphism problem (GI) is in Low_2, the second level of the low hierarchy. This result provides strong evidence against the NP-completeness of GI, as can be seen as follows. If GI were NP-complete, then GI would be in $\mathrm{High}_0 \subseteq \mathrm{High}_2$, since by Theorem 8.13 High_0 contains exactly the $\leq_{\mathrm{T}}^{\mathrm{p}}$-complete NP sets and in particular the $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete sets in NP. Again by Theorem 8.13, we have that $\mathrm{Low}_2 \cap \mathrm{High}_2$ is nonempty if and only if PH collapses to Σ_2^p, which is considered very unlikely.

To prove the lowness of the graph isomorphism problem, we first need a technical prerequisite, the so-called hashing lemma, stated here as Lemma 8.15. Hashing is method widely used in computer science for dynamic data management. The idea is the following. Assign to every data set some (short) key in a unique manner. The set of all potential keys, called the universe U, is usually very large. In contrast, the set $V \subseteq U$ of those keys actually used is often much smaller. A *hash function* $h : U \to T$ maps the elements of U to the *hash table* $T = \{0, 1, \ldots, k-1\}$. Hash functions are many-to-one mappings. That is, distinct keys from U can be mapped to the same address in T. However, if possible, one wishes to map two distinct keys from V to distinct addresses in T. That is, one seeks to avoid collisions on the set of keys actually used. If possible, a hash function thus should be injective on V.

Among the various known hashing techniques, we are here interested in *universal hashing*, which was introduced by Carter and Wegman [10] in 1979. The idea is to randomly choose a hash function from a suitable family of hash functions. This method is universal in the sense that it does no longer depend on a particular set V of keys actually used. Rather, it seeks to avoid collisions with high probability on *all* sufficiently small sets V. The probability here is with respect to the random choice of the hash function.

In what follows, we assume that keys are encoded as strings over the alphabet $\Sigma = \{0, 1\}$. The set of all length n strings in Σ^* is denoted by Σ^n.

Definition 8.14 (Hashing). *Let* $\Sigma = \{0, 1\}$, *and let* m *and* t *be positive integers with* $t > m$. *A* hash function $h : \Sigma^t \to \Sigma^m$ *is a linear mapping determined by a Boolean* $t \times m$ *matrix* $B_h = (b_{i,j})_{i,j}$, *where* $b_{i,j} \in \{0, 1\}$. *For* $x \in \Sigma^t$ *and* $1 \leq j \leq m$, *the jth bit of $y = h(x)$ in Σ^m is given by $y_j = (b_{1,j} \wedge x_1) \oplus (b_{2,j} \wedge x_2) \oplus \cdots \oplus (b_{t,j} \wedge x_t)$, where $\oplus$ denotes the logical* exclusive-or *operation, i.e.,*

$$a_1 \oplus a_2 \oplus \cdots \oplus a_n = 1 \iff |\{i \mid a_i = 1\}| \equiv 1 \bmod 2 .$$

Let $\mathcal{H}_{t,m}$ be a family of hash functions for the parameters t and m:

$$\mathcal{H}_{t,m} = \{h : \Sigma^t \to \Sigma^m \mid B_h \textit{ is a Boolean } t{\times}m \textit{ matrix}\} .$$

On $\mathcal{H}_{t,m}$, we assume the uniform distribution: A hash function h is chosen from $\mathcal{H}_{t,m}$ by picking the bits $b_{i,j}$ in B_h independently and uniformly distributed.

Let $V \subseteq \Sigma^t$. For any subfamily $\widehat{\mathcal{H}}$ of $\mathcal{H}_{t,m}$, we say there is a collision on V *if*

$$(\exists \vec{v} \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists \vec{x} \in V)\,[\vec{v} \neq \vec{x} \wedge h(\vec{v}) = h(\vec{x})] .$$

Otherwise, $\widehat{\mathcal{H}}$ is said to be collision-free on V*.*

A collision on V means that none of the hash functions in a subfamily $\widehat{\mathcal{H}}$ is injective on V. The following lemma says that, if V is sufficiently small, a randomly chosen subfamily of $\mathcal{H}_{t,m}$ must be collision-free. In contrast, if V is too large, collisions cannot be avoided. The proof of Lemma 8.15 is omitted.

Lemma 8.15 (Hashing lemma). *Let $t, m \in \mathbb{N}$ be fixed parameters, where $t > m$. Let $V \subseteq \Sigma^t$ and let $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ be a family of hash functions randomly chosen from $\mathcal{H}_{t,m}$ under the uniform distribution. Let*

$$C(V) = \{\widehat{\mathcal{H}} \mid (\exists v \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists x \in V)\,[v \neq x \wedge h(v) = h(x)]\}$$

be the event that for $\widehat{\mathcal{H}}$ a collision occurs on V. Then, the following two statements hold:

1. *If $|V| \leq 2^{m-1}$, then $C(V)$ occurs with probability at most $1/4$.*
2. *If $|V| > (m+1)2^m$, then $C(V)$ occurs with probability 1.*

In Section 7.5, the Arthur-Merlin hierarchy has been defined, and it was mentioned that this hierarchy collapses to its second level. Here, we are interested in the class coAM, cf. Definition 7.16 in Subsection 7.5.1.

Theorem 8.16 (Schöning). `GI` *is in* Low_2*.*

Proof By Theorem 8.13, every NP $\cap$ coAM set is in Low_2. Thus, to prove that `GI` in Low_2, it is enough to show that `GI` is in coAM. Let G and H be two graphs with n vertices each. We wish to apply Lemma 8.15. A first idea is to use

$$A(G,H) = \{(F,\varphi) \mid F \cong G \wedge \varphi \in \mathrm{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \wedge \varphi \in \mathrm{Aut}(F)\}$$

as the set V from that lemma. By Lemma 7.11, we have $|A(G,H)| = n!$ if $G \cong H$, and $|A(G,H)| = 2n!$ if $G \not\cong H$.

The coAM machine we wish to construct for `GI` is polynomial-time bounded.

Thus, the parameters t and m from the hashing lemma must be polynomial in n. So, to apply Lemma 8.15, we would have to choose a polynomial $m = m(n)$ such that

$$n! \leq 2^{m-1} < (m+1)2^m < 2n! \, . \tag{8.4}$$

This would guarantee that the set $V = A(G, H)$ would be large enough to tell two isomorphic graphs G and H apart from two nonisomorphic graphs G and H. Unfortunately, it is not possible to find a polynomial m that satisfies (8.4). Thus, we define a different set V, which yields a gap large enough to distinguish isomorphic graphs from nonisomorphic graphs.

Define $V = A(G, H)^n = \underbrace{A(G, H) \times A(G, H) \times \cdots \times A(G, H)}_{n \text{ times}}$. Now, (8.4) changes to

$$(n!)^n \leq 2^{m-1} < (m+1)2^m < (2n!)^n \, , \tag{8.5}$$

and this inequality can be satisfied by choosing $m = m(n) = 1 + \lceil n \log n! \rceil$, which is polynomially in n as desired.

Construct a coAM machine M for GI as follows. Given the graphs G and H each having n vertices, M first computes the parameter m. The set $V = A(G, H)^n$ contains n-tuples of pairs each having the form (F, φ), where F is a graph with n vertices, and where φ is a permutation in the automorphism group $\mathrm{Aut}(F)$. The elements of V can be suitably encoded as strings over the alphabet $\Sigma = \{0, 1\}$, for a suitable polynomial $t = t(n)$. All computations performed so far are deterministic.

Then, M performs Arthur's probabilistic move by randomly choosing a family $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ of hash functions from $\mathcal{H}_{t,m}$ under the uniform distribution. Each hash function $h_i \in \widehat{\mathcal{H}}$ is represented by a Boolean $t \times m$ matrix. Thus, the $m+1$ hash functions h_i in $\widehat{\mathcal{H}}$ can be represented as a string $z_{\widehat{\mathcal{H}}} \in \Sigma^*$ of length $p(n)$ for a suitable polynomial p. Modify the collision predicate $C(V)$ defined in the hashing lemma as follows:

$$B = \{(G, H, z_{\widehat{\mathcal{H}}}) \mid (\exists v \in V)\, (\forall i : 1 \leq i \leq m+1)\, (\exists x \in V)\, [v \neq x \wedge h_i(v) = h_i(x)]\} \, .$$

Note that the $\forall$ quantifier in B ranges over only polynomially many i and can thus be evaluated in deterministic polynomial time. It follows that the two $\exists$ quantifiers in B can be merged into a *single* polynomially length-bounded $\exists$ quantifier. By Theorem 8.11, B is a set in $\Sigma_1^p = \mathrm{NP}$. Let N be an NPTM for B. For the string $z_{\widehat{\mathcal{H}}}$ that encodes $m+1$ randomly picked hash functions from $\mathcal{H}_{t,m}$, M now simulates the computation of $N(G, H, z_{\widehat{\mathcal{H}}})$. This corresponds to Merlin's move. Finally, M accepts its input (G, H) if and only if $N(G, H, z_{\widehat{\mathcal{H}}})$ accepts.

We now estimate the probability (taken over the random choice of the hash functions in $z_{\widehat{\mathcal{H}}}$ that M accepts its input (G, H). If G and H isomorphic, then $|A(G, H)| = n!$ by Lemma 7.11. Inequality (8.5) implies $|V| = (n!)^n \leq 2^{m-1}$. By Lemma 8.15, the probability that $(G, H, z_{\widehat{\mathcal{H}}})$ is in B (and that $M(G, H)$ thus accepts) is at most $1/4$. However, if G and H are nonisomorphic, Lemma 7.11 implies that $|A(G, H)| = 2n!$. Inequality (8.5) now gives $|V| = (2n!)^n > (m+1)2^m$. By Lemma 8.15, the probability that $(G, H, z_{\widehat{\mathcal{H}}})$ is in B and $M(G, H)$ thus accepts is 1. It follows that GI is in coAM as desired. ∎

8.4.3. Graph isomorphism is in SPP

The probabilistic complexity class RP was introduced in Definition 7.14 in Subsection 7.3.1. In this section, two other probabilistic complexity classes are important that we will now define: PP and SPP, which stand for *P*robabilistic *P*olynomial Time and *S*toic *P*robabilistic *P*olynomial Time, respectively.

Definition 8.17 (PP and SPP). *The class* PP *contains exactly those problems A for which there exists an NPTM M such that for each input x: If $x \in A$ then $M(x)$ accepts with probability at least $1/2$, and if $x \notin A$ then $M(x)$ accepts with probability less than $1/2$.*

For any NPTM M running on input x, let $\mathrm{acc}_M(x)$ denote the number of accepting computation paths of $M(x)$ and let $\mathrm{rej}_M(x)$ denote the number of rejecting computation paths of $M(x)$. Define $\mathrm{gap}_M(x) = \mathrm{acc}_M(x) - \mathrm{rej}_M(x)$.

The class SPP *contains exactly those problems A for which there exists an NPTM M such that for each input x: $(x \in A \implies \mathrm{gap}_M(x) = 1)$ and $(x \notin A \implies \mathrm{gap}_M(x) = 0)$.*

In other words, an SPP machine is "stoic" in the sense that its "gap" (i.e., the difference between its accepting and rejecting computation paths) can take on only two out of an exponential number of possible values, namely 1 and 0. Unlike PP, SPP is a so-called "*promise class*". since an SPP machine M "promises" that $\mathrm{gap}_M(x) \in \{0, 1\}$ for each x.

The notion of lowness can be defined for any relativizable complexity class $\mathcal{C}$: A set A is said to be $\mathcal{C}$-*low* if and only if $\mathcal{C}^A = \mathcal{C}$. In particular, for each k, the kth level Low_k of the low hierarchy within NP (see Definition 8.12) contains exactly the NP sets that are Σ_k^p-low. It is known that all sets in SPP are PP-low. This and other useful properties of SPP are listed in the following theorem without proof; see also [14, 32, 33].

Theorem 8.18

1. SPP *is* PP*-low, i.e.,* $\mathrm{PP}^{\mathrm{SPP}} = \mathrm{PP}$.
2. SPP *is self-low, i.e.,* $\mathrm{SPP}^{\mathrm{SPP}} = \mathrm{SPP}$.
3. *Let A be a set in* NP *via some NPTM N and let L be a set in* SPP^A *via some NPOTM M such that, for each input x, $M^A(x)$ asks only queries q satisfying* $\mathrm{acc}_N(q) \leq 1$*. Then, L is in* SPP.
4. *Let A be a set in* NP *via some NPTM N and let f be a function in* FP^A *via some DPOTM M such that, for each input x, $M^A(x)$ asks only queries q satisfying* $\mathrm{acc}_N(q) \leq 1$*. Then, f is in* $\mathrm{FP}^{\mathrm{SPP}}$.

The following theorem says that the lexicographically smallest permutation in a right coset (see Definition 7.6 in Section 7.1.3) can be determined efficiently. The lexicographic order on $\mathfrak{S}_n$ is defined in Example 8.3.

Theorem 8.19 *Let $\mathfrak{G} \leq \mathfrak{S}_n$ be a permutation group with $\mathfrak{G} = \langle G \rangle$ and let π be a permutation in $\mathfrak{S}_n$. There is a polynomial-time algorithm that, given (G, π), computes the lexicographically smallest permutation in the right coset $\mathfrak{G}\pi$ of $\mathfrak{G}$ in $\mathfrak{S}_n$.*

Proof We now state our algorithm LERC for computing the lexicographically smallest permutation in the right coset $\mathfrak{G}\pi$ of $\mathfrak{G}$ in $\mathfrak{S}_n$, where the permutation group $\mathfrak{G}$ is given by a generator G, see Definition 7.6 in Subsection 7.1.3.

```
LERC($G, \pi$)
1 compute the tower $\mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)}$ of stabilisers in $\mathfrak{G}$
2     $\varphi_0 \leftarrow \pi$
3     for $i \leftarrow 0$ to $n-1$
4         do $x \leftarrow i+1$
5             compute the element $y$ in the orbit $\mathfrak{G}^{(i)}(x)$ for which $\varphi_i(y)$
              is minimum
6             compute a permutation $\tau_i$ in $\mathfrak{G}^{(i)}$ such that $\tau_i(x) = y$
7             $\varphi_{i+1} \leftarrow \tau_i \varphi_i$
8     return $\varphi_n$
```

By Theorem 7.7, the tower $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ of stabilisers of $\mathfrak{G}$ can be computed in polynomial time. More precisely, for each i with $1 \leq i \leq n$, the complete right transversals T_i $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ are determined, and thus a strong generator $S = \bigcup_{i=1}^{n-1} T_i$ of $\mathfrak{G}$.

Note that $\varphi_0 = \pi$ and $\mathfrak{G}^{(n-1)} = \mathfrak{G}^{(n)} = \mathbf{id}$. Thus, to prove that the algorithm works correctly, it is enough to show that for each i with $0 \leq i \leq n-1$, the lexicographically smallest permutation of $\mathfrak{G}^{(i)}\varphi_i$ is contained in $\mathfrak{G}^{(i+1)}\varphi_{i+1}$. By induction, it follows that $\mathfrak{G}^{(n)}\varphi_n = \{\varphi_n\}$ also contains the lexicographically smallest permutation of $\mathfrak{G}\pi = \mathfrak{G}^{(0)}\varphi_0$. Thus, algorithm LERC indeed outputs the lexicographically smallest permutation of φ_n of $\mathfrak{G}\pi$.

To prove the above claim, let us denote the orbit of an element $x \in [n]$ in a permutation group $\mathfrak{H} \leq \mathfrak{S}_n$ by $\mathfrak{H}(x)$. Let τ_i be the permutation in $\mathfrak{G}^{(i)}$ that maps $i+1$ onto the element y in the orbit $\mathfrak{G}^{(i)}(i+1)$ for which $\varphi_i(y) = x$ is the minimal element in the set $\{\varphi_i(z) \mid z \in \mathfrak{G}^{(i)}(i+1)\}$.

By Theorem 7.7, the orbit $\mathfrak{G}^{(i)}(i+1)$ can be computed in polynomial time. Since $\mathfrak{G}^{(i)}(i+1)$ contains at most $n-i$ elements, y can be determined efficiently. Our algorithm ensures that $\varphi_{i+1} = \tau_i\varphi_i$. Since every permutation in $\mathfrak{G}^{(i)}$ maps each element of $[i]$ onto itself, and since $\tau_i \in \mathfrak{G}^{(i)}$, it follows that for each j with $1 \leq j \leq i$, for each $\tau \in \mathfrak{G}^{(i)}$, and for each $\sigma \in \mathfrak{G}^{(i+1)}$,

$$(\sigma\varphi_{i+1})(j) = \varphi_{i+1}(j) = (\tau_i\varphi_i)(j) = \varphi_i(j) = (\tau\varphi_i)(j) \ .$$

In particular, it follows for the lexicographically smallest permutation μ in $\mathfrak{G}^{(i)}\varphi_i$ that every permutation from $\mathfrak{G}^{(i+1)}\varphi_{i+1}$ must coincide with μ on the first i elements, i.e. on $[i]$.

Moreover, for each $\sigma \in \mathfrak{G}^{(i+1)}$ and for the element $x = \varphi_i(y)$ defined above, we have

$$(\sigma\varphi_{i+1})(i+1) = \varphi_{i+1}(i+1) = (\tau_i\varphi_i)(i+1) = x \ .$$

Clearly, $\mathfrak{G}^{(i+1)}\varphi_{i+1} = \{\varphi \in \mathfrak{G}^{(i)}\varphi_i \mid \varphi(i+1) = x\}$. The claim now follows from the fact that $\mu(i+1) = x$ for the lexicographically smallest permutation μ of $\mathfrak{G}^{(i)}\varphi_i$.

Thus, LERC is a correct algorithm. It easy to see that it is also efficient. ■

Theorem 8.19 can easily be extended to Corollary 8.20, see Exercise 8-3.

Corollary 8.20 *Let $\mathfrak{G} \leq \mathfrak{S}_n$ be a permutation group with $\mathfrak{G} = \langle G \rangle$, and let π and ψ be two given permutations in $\mathfrak{S}_n$. There exists a polynomial-time algorithm that, given (G, π, ψ), computes the lexicographically smallest permutation of $\psi\mathfrak{G}\pi$.*

We now prove Theorem 8.21, the main result of this section.

Theorem 8.21 (Arvind and Kurur). `GI` *is in* SPP.

Proof Define the (functional) problem `AUTO` as follows: Given a graph G, compute a strong generator of the automorphism group $\mathrm{Aut}(G)$; see Definition 7.6 and the subsequent paragraph and Definition 7.8 for these notions. By Mathon's [37] result, the problems `AUTO` and `GI` are Turing-equivalent (see also [33]), i.e., `AUTO` is in $\mathrm{FP}^{\mathtt{GI}}$ and `GI` is in $\mathrm{P}^{\mathtt{AUTO}}$. Thus, it is enough to show that `AUTO` is in $\mathrm{FP}^{\mathrm{SPP}}$ because the self-lowness of SPP stated in Theorem 8.18 implies that `GI` is in $\mathrm{P}^{\mathtt{AUTO}} \subseteq \mathrm{SPP}^{\mathrm{SPP}} \subseteq \mathrm{SPP}$, which will complete the proof.

So, our goal is to find an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for `AUTO`. Given a graph G, this algorithm has to compute a strong generator $S = \bigcup_{i=1}^{n-1} T_i$ for $\mathrm{Aut}(G)$, where

$$\mathbf{id} = \mathrm{Aut}(G)^{(n)} \leq \mathrm{Aut}(G)^{(n-1)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$$

is the tower of stabilisers of $\mathrm{Aut}(G)$ and T_i, $1 \leq i \leq n$, is a complete right transversal of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$.

Starting with the trivial case, $\mathrm{Aut}(G)^{(n)} = \mathbf{id}$, we build step by step a strong generator for $\mathrm{Aut}(G)^{(i)}$, where i is decreasing. Eventually, we will thus obtain a strong generator for $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. So suppose a strong generator $S_i = \bigcup_{j=i}^{n-1} T_j$ for $\mathrm{Aut}(G)^{(i)}$ has already been found. We now describe how to determine a complete right transversal T_{i-1} of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$ by our $\mathrm{FP}^{\mathrm{SPP}}$ algorithm. Define the oracle set

$$A = \left\{ (G, S, i, j, \pi) \;\middle|\; \begin{array}{l} S \subseteq \mathrm{Aut}(G) \text{ and } \langle S \rangle \text{ is a pointwise stabilizer of } [i] \\ \text{in } \mathrm{Aut}(G),\ \pi \text{ is a partial permutation, which pointwise} \\ \text{stabilises } [i-1], \text{ and } \pi(i) = j, \text{ and there is a } \tau \text{ in} \\ \mathrm{Aut}(G)^{(i-1)} \text{ with } \tau(i) = j \text{ and } \mathrm{LERC}(S, \tau) \text{ extends } \pi \end{array} \right\}.$$

By Theorem 8.19, the lexicographically smallest permutation $\mathrm{LERC}(S, \tau)$ of the right coset $\langle S \rangle \tau$ can be determined in polynomial time by our algorithm. The partial permutation π belongs to the input (G, S, i, j, π), since we wish to use A as an oracle in order to find the lexicographically smallest permutation by prefix search; cf. Example 8.3.

Consider the following NPTM N for A:

```
N(G, S, i, j, π)
1  verify that $S \subseteq \mathrm{Aut}(G)^{(i)}$
2  nondeterministically guess a permutation $\tau \in \mathfrak{S}_n$; // $G$ has $n$ vertices
3  if $\tau \in \mathrm{Aut}(G)^{(i-1)}$ and $\tau(i) = j$ and $\tau$ extends $\pi$ and $\tau = \mathrm{LERC}(S, \tau)$
4     then accept and halt
5     else reject and halt
```

Thus, A is in NP. Note that if $\tau(i) = j$ then $\sigma(i) = j$, for each permutation σ in the right coset $\langle S \rangle \tau$.

We now show that if $\langle S \rangle = \mathrm{Aut}(G)^{(i)}$ then the number of accepting computation paths of N on input (G, S, i, j, π) is either 0 or 1. In general, $\mathrm{acc}_N(G, S, i, j, \pi) \in \{0, |\mathrm{Aut}(G)^{(i)}|/|\langle S \rangle|\}$.

Suppose (G, S, i, j, π) is in A and $\langle S \rangle = \mathrm{Aut}(G)^{(i)}$. If $\tau(i) = j$ for some $\tau \in \mathrm{Aut}(G)^{(i-1)}$ and $j > i$, then the right coset $\langle S \rangle \tau$ contains exactly those permutations in $\mathrm{Aut}(G)^{(i-1)}$ that map i to j. Thus, the only accepting computation path of $N(G, S, i, j, \pi)$ corresponds to the unique lexicographically smallest permutation $\tau = \mathrm{LERC}(S, \tau)$. If, on the other hand, $\langle S \rangle$ is a strict subgroup of $\mathrm{Aut}(G)^{(i)}$, then $\mathrm{Aut}(G)^{(i)}\tau$ can be written as the disjoint union of $k = |\mathrm{Aut}(G)^{(i)}|/|\langle S \rangle|$ right cosets of $\langle S \rangle$. In general, $N(G, S, i, j, \pi)$ thus possesses k accepting computation paths if (G, S, i, j, π) is in A, and otherwise it has no accepting computation path.

```
M-A(G)
 1  set $T_i := \{\mathrm{id}\}$ for each $i$, $0 \le i \le n-2$; // $G$ has $n$ vertices
                ▷ $T_i$ will be a complete right transversal of $\mathrm{Aut}(G)^{(i+1)}$ in $\mathrm{Aut}(G)^{(i)}$.
 3  set $S_i := \emptyset$ for each $i$, $0 \le i \le n-2$
 4  set $S_{n-1} := \{\mathrm{id}\}$
                                  ▷ $S_i$ will be a strong generator for $\mathrm{Aut}(G)^{(i)}$.
 5  for $i \leftarrow n-1$ downto 1
                                        ▷ $S_i$ is already found at the start
                       ▷ of the $i$th iteration, and $S_{i-1}$ will now be computed.
 6     do let $\pi : [i-1] \to [n]$ be the partial permutation
          with $\pi(a) = a$ for each $a \in [i-1]$
 7                    ▷ For $i = 1$, $\pi$ is the nowhere defined partial permutation.
 8        for $j \leftarrow i+1$ to $n$
 9           do $\hat{\pi} := \pi j$, i.e., $\hat{\pi}$ extends $\pi$ by the pair $(i, j)$ with $\hat{\pi}(i) = j$
10              if $((G, S_i, i, j, \hat{\pi}) \in A)$ {
11                 then                     ▷ Construct the smallest permutation
                             in $\mathrm{Aut}(G)^{(i-1)}$ mapping $i$ to $j$ by prefix search.
12                    for $k \leftarrow i+1$ to $n$
13                       do find the element $\ell$ not in the image
                            of $\hat{\pi}$ with $(G, S_i, i, j, \hat{\pi}\ell) \in A$
14                    $\hat{\pi} := \hat{\pi}\ell$
15                                    ▷ Now, $\hat{\pi}$ is a total permutation in $\mathfrak{S}_n$
```

```
16                      $T_{i-1} \leftarrow T_{i-1} \cup \hat{\pi}$ now, $T_{i-1}$ is
                              a complete right transversal of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$
17      $S_{i-1} := S_i \cup T_{i-1}$.
18  return $S_0$              $\triangleright$ $S_0$ is a strong generator for $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$
```

The above algorithm is an FP^A algorithm M^A for AUTO. The DPOTM M makes only queries $q = (G, S_i, i, j, \pi)$ to its oracle A for which $\langle S_i \rangle = \mathrm{Aut}(G)^{(i)}$. Thus, $\mathrm{acc}_N(q) \leq 1$ for each query q actually asked. By item 4 of Theorem 8.18, it follows that AUTO is in $\mathrm{FP}^{\mathrm{SPP}}$.

The claim that the output S_0 of $M^A(G)$ indeed is a strong generator for $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ can be shown by induction on n. The induction base is $n-1$, and $S_{n-1} = \{\mathrm{id}\}$ of course generates $\mathrm{Aut}(G)^{(n-1)} = \mathbf{id}$.

For the induction step, assume that prior to the ith iteration a strong generator S_i for $\mathrm{Aut}(G)^{(i)}$ has already been found. We now show that after the ith iteration the set $S_{i-1} = S_i \cup T_{i-1}$ is a strong generator for $\mathrm{Aut}(G)^{(i-1)}$. For each j with $i+1 \leq j \leq n$, the oracle query "$(G, S_i, i, j, \hat{\pi}) \in A$?" checks whether there is a permutation in $\mathrm{Aut}(G)^{(i-1)}$ mapping i to j. By prefix search, which is performed by making suitable oracle queries to A again, the lexicographically smallest permutation $\hat{\pi}$ in $\mathrm{Aut}(G)^{(i-1)}$ with $\hat{\pi}(i) = j$ is constructed. Note that, as claimed above, only queries q satisfying $\mathrm{acc}_N(q) \leq 1$ are made to A, since S_i is a strong generator for $\mathrm{Aut}(G)^{(i)}$, so $\langle S_i \rangle = \mathrm{Aut}(G)^{(i)}$. By construction, after the ith iteration, T_{i-1} is a complete right transversal of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$. It follows that $S_{i-1} = S_i \cup T_{i-1}$ is a strong generator for $\mathrm{Aut}(G)^{(i-1)}$. Eventually, after n iterations, a strong generator S_0 for $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ is found. ■

From Theorem 8.21 and the first two items of Theorem 8.18, we obtain Corollary 8.22.

Corollary 8.22 GI *is low for both* SPP *and* PP*, i.e.,* $\mathrm{SPP}^{\mathtt{GI}} = \mathrm{SPP}$ *and* $\mathrm{PP}^{\mathtt{GI}} = \mathrm{PP}$.

Exercises

8.4-1 By Definition 8.9, $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ if and only if $A \in \mathrm{P}^B$. Show that $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ if and only if $\mathrm{P}^A \subseteq \mathrm{P}^B$.

8.4-2 Show that the set Pre-Iso defined in Example 8.3 is in NP. Moreover, prove that the machine N defined in Example 8.3 runs in polynomial time, i.e., show that N is a DPOTM.

Problems

8-1 Strong NPOTM

A ***strong NPOTM*** is an NPOTM with three types of final states, i.e., the set F of final states of M is partitioned into F_a (accepting states), F_r (rejecting states), and $F_?$ ("don't know" states) such that the following holds: If $x \in A$ then $M^B(x)$ has at least one computation path halting in an accepting state from F_a but no computation path halting in a rejecting state from F_r. If $x \notin A$ then $M^B(x)$ has at least one computation path halting in a rejecting state from F_r but no computation path halting in an accepting state from F_a. In both cases $M^B(x)$ may have computation paths halting in "don't know" states from $F_?$. In other words, strong NPOTMs are machines that never lie. Prove the following two assertions:

(a) $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B$ if and only if there exists a strong NPOTM M with $A = L(M^B)$.

(b) $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B$ if and only if $\mathrm{NP}^A \subseteq \mathrm{NP}^B$.

Hint. Look at Exercise 8.4-1.

8-2 Proofs

Prove the assertions from Theorems 8.11 and 8.13. (Some are far from being trivial!)

8-3 Modification of the proofs

Modify the proof of Theorem 8.19 so as to obtain Corollary 8.20.

Chapter Notes

Parts of the Chapters 7 and 8 are based on the book [52] that provides the proofs omitted here, such as those of Theorems 8.11, 8.13, and 8.18 and of Lemma 8.15, and many more details.

More background on complexity theory can be found in the books [24, 43, 67, 68]. A valuable source on the theory of NP-completeness is still the classic [15] by Garey and Johnson. The $\leq_{\mathrm{T}}^{\mathrm{p}}$-reducibility was introduced by Cook [11], and the $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility by Karp [31]. A deep and profound study of polynomial-time reducibilities is due to Ladner, Lynch, and Selman [35].

Exercise 8-1 and Problem 8-1 are due to Selman [58].

Dantsin et al. [13] obtained an upper bound of $\tilde{\mathcal{O}}(1.481^n)$ for the deterministic time complexity of `3-SAT`, which was further improved by Brueggemann and Kern [9] to $\tilde{\mathcal{O}}(1.4726^n)$. The randomised algorithm presented in Subsection 8.3.2 is due to Schöning [56]; it is based on a "limited local search with restart". For `k-SAT` with $k \geq 4$, the algorithm by Paturi et al. [44] is slightly better than Schöning's algorithm. Iwama and Tamaki [29] combined the ideas of Schöning [56] and Paturi et al. [44] to obtain a bound of $\tilde{\mathcal{O}}(1.324^n)$ for `k-SAT` with $k \in \{3, 4\}$. For `k-SAT` with $k \geq 5$, their algorithm is not better than that by Paturi et al. [44].

Figure 8.7 gives an overview over some algorithms for the satisfiability problem.

For a thorough, comprehensive treatment of the graph isomorphism problem the reader is referred to the book by Köbler, Schöning, and Torán [34], particularly under complexity-theoretic aspects. Hoffman [27] investigates group-theoretic algorithms for the graph isomorphism problem and related problems.

Algorithm	Type	3-SAT	4-SAT	5-SAT	6-SAT
Backtracking	det.	$\tilde{\mathcal{O}}(1.913^n)$	$\tilde{\mathcal{O}}(1.968^n)$	$\tilde{\mathcal{O}}(1.987^n)$	$\tilde{\mathcal{O}}(1.995^n)$
Monien and Speckenmeyer [41]	det.	$\tilde{\mathcal{O}}(1.618^n)$	$\tilde{\mathcal{O}}(1.839^n)$	$\tilde{\mathcal{O}}(1.928^n)$	$\tilde{\mathcal{O}}(1.966^n)$
Dantsin et al. [13]	det.	$\tilde{\mathcal{O}}(1.481^n)$	$\tilde{\mathcal{O}}(1.6^n)$	$\tilde{\mathcal{O}}(1.667^n)$	$\tilde{\mathcal{O}}(1.75^n)$
Brueggemann and Kern [9]	det.	$\tilde{\mathcal{O}}(1.4726^n)$	—	—	—
Paturi et al. [44]	prob.	$\tilde{\mathcal{O}}(1.362^n)$	$\tilde{\mathcal{O}}(1.476^n)$	$\tilde{\mathcal{O}}(1.569^n)$	$\tilde{\mathcal{O}}(1.637^n)$
Schöning [56]	prob.	$\tilde{\mathcal{O}}(1.334^n)$	$\tilde{\mathcal{O}}(1.5^n)$	$\tilde{\mathcal{O}}(1.6^n)$	$\tilde{\mathcal{O}}(1.667^n)$
Iwama and Tamaki [29]	prob.	$\tilde{\mathcal{O}}(1.324^n)$	$\tilde{\mathcal{O}}(1.474^n)$	—	—

Figure 8.7 Runtimes of some algorithms for the satisfiability problem.

The polynomial hierarchy was introduced by Meyer and Stockmeyer [38, 64]. In particular, Theorem 8.11 is due to them. Schöning [54] introduced the low hierarchy and the high hierarchy within NP. The results stated in Theorem 8.13 are due to him [54]. He also proved that GI is in Low_2, see [55]. Köbler et al. [32, 33] obtained the first lowness results of GI for probabilistic classes such as PP. These results were improved by Arvind and Kurur [3] who proved that GI is even in SPP. The class SPP generalises Valiant's class UP, see [66]. So-called "promise classes" such as UP and SPP have been thoroughly studied in a number of papers; see, e.g., [3, 8, 14, 32, 33, 48, 51]. Lemma 8.15 is due to Carter and Wegman [10].

The author is grateful to Uwe Schöning for his helpful comments on an earlier version of this chapter and for sending the slides of one of this talks that helped simplifying the probability analysis for the algorithm Random-SAT sketched in Subsection 8.3; a more comprehensive analysis can be found in Schöning's book [57]. Thanks are also due to Dietrich Stoyan, Robert Stoyan, Sigurd Assing, Gábor Erdélyi and Holger Spakowski for proofreading previous versions of Chapters 7 and 8. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants RO 1202/9-1 and RO 1202/9-3 and by the Alexander von Humboldt Foundation in the TransCoop program.

Bibliography

[1] M. Agrawal, N. Kayal, N. Saxena. Primes is in p. *Annals of Mathematics*, 160(2):781–793, 2004. 351

[2] M. Agrawal, N. Kayal, N. Saxena. PRIMES is in P. http://www.cse.iitk.ac.in/users/manindra/, 2002. 351

[3] V. Arvind, P. Kurur. Graph isomorphism is in SPP. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science*, 743–750 pages. IEEE Computer Society Press, 2002. 393

[4] L. Babai. Trading group theory for randomness. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, 421–429 pages. ACM Press, 1985. 364

[5] L. Babai, S. Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and Systems Sciences*, 36(2):254–276, 1988. 364

[6] A. Beygelzimer, L. A. Hemaspaandra, C. Homan, J. Rothe. One-way functions in worst-case cryptography: Algebraic and security properties are on the house. *SIGACT News*, 30(4):25–40, 1999. 364

[7] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999. 364

[8] B. Borchert, L. A. Hemaspaandra, J. Rothe. Restrictive acceptance suffices for equivalence problems. *London Mathematical Society Journal of Computation and Mathematics*, 86:86–95, 2000. 393

[9] T. Brueggemann, W. Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, 2004. 376, 392, 393

[10] J. L. Carter, M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. 384, 393

[11] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3th Annual ACM Symposium on Theory of Computing*, 151–158 pages. ACM Press, 1971. 392

[12] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997. 364

[13] E. Dantsin, A. Goerdt, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002. 376, 392, 393

[14] S. Fenner, L. Fortnow, S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994. 387, 393

[15] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 370, 371, 374, 392

[16] O. Goldreich, S. Micali, A. Wigderson. Proofs that yield nothing but their validity or all languages in NP. *Journal of the ACM*, 38(3):691–729, 1991. 360

[17] O. Goldreich. Randomness, interactive proofs, and zero-knowledge – a survey. In R. Herken (Ed.), *The Universal Turing Machine: A Half-Century Survey*, 377–405 pages. Oxford University Press, 1988. 364

[18] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001. 364

[19] S. Goldwasser. Interactive proof systems. In J. Hartmanis (Ed.) Computational Complexity Theory, AMS Short Course Lecture Notes: Introductory Survey Lectures. Proceedings of Symposia in Applied Mathematics, Vol. 38, pages 108–128. American Mathematical Society, 1989. 364

[20] S. Goldwasser, S. Micali, C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. 364

[21] S. Goldwasser, M. Sipser. Private coins versus public coins in interactive proof systems. in s. micali (ed.). In *Randomness and Computation*, Advances in Computing Research, Vol. 5, pages 73–90. JAI Press, 1989. A preliminary version appeared in *Proc. 18th Ann. ACM Symp. on Theory of Computing*, 1986. 359

[22] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, 1988. Special issue on cryptography. 364

[23] L. A. Hemaspaandra, J. Rothe, A. Saxena. Enforcing and defying associativity, commutativity, totality, and strong noninvertibility for one-way functions in complexity theory. In In M. Coppo et al. (Eds.) ICTCS 2005, Lecture Notes in Computer Science, Vol. 117, pages 265–279. Springer verlag, 2005. 357, 364

[24] L. A. Hemaspaandra, M. Ogihara. *The Complexity Theory Companion*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2002. 392

[25] L. A. Hemaspaandra, K. Pasanen, J. Rothe. If P $\neq$ NP then some strongly noninvertible functions are invertible. *Theoretical Computer Science*, 362(1–3):54–62, 2006. 357, 364

[26] L. A. Hemaspaandra, J. Rothe. Creating strong, total, commutative, associative one-way functions from any one-way function in complexity theory. *Journal of Computer and Systems Sciences*, 58(3):648–659, 1999. 357, 364

[27] C. Hoffman (Ed.). *Group-Theoretic Algorithms and Graph Isomorphism*. Lecture Notes in Computer Science. Springer-Verlag, Vol. 136, 1982. 392

[28] C. Homan. Tight lower bounds on the ambiguity in strong, total, associative, one-way functions. *Journal of Computer and System Sciences*, 68(3):657–674, 2004. 364

[29] K. Iwama, S. Tamaki. Improved upper bounds for 3-SAT. In In J. Munro (Ed.) Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 328–329. Society for Industrial and Applied Mathematics, 2004. 392, 393

[30] B. Kaliski, M. Robshaw. The secure use of RSA. *CryptoBytes*, 1(3):7–13, 1995. 364

[31] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher (Eds.), *Complexity of Computer Computations*, 85–103 pages. Plenum Press, 1972. 392

[32] J. Köbler, U. Schöning, S. Toda, J. Torán. Turing machines with few accepting computations and low sets for PP. *Journal of Computer and System Sciences*, 44(2):272–286, 1992. 363, 387, 393

[33] J. Köbler, U. Schöning, J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2:301–330, 1992. 387, 389, 393

[34] J. Köbler, U. Schöning, J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993. 364, 392

[35] R. Ladner, N. A. Lynch, A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–124, 1975. 380, 392

[36] A. Lenstra, H. Lenstra. *The Development of the Number Field Sieve*. Lecture Notes in Mathematics. Vol. 1554, Springer-Verlag, 1993. 355

[37] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979. 389

[38] A. Meyer, L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 129–129. 1972. 393

[39] D. Micciancio, S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective*. Vol. 671.,The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2002. 364

[40] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and Systems Sciences*, 13(3):300–317, 1976. 352

[41] B. Monien, E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985. 376, 393

[42] J. Moore. Protocol failures in cryptosystems. In G. Simmons (Ed.), *Contemporary Cryptology: The Science of Information Integrity*, 541–558 pages. IEEE Computer Society Press, 1992. 364

[43] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 364, 392

[44] R. Paturi, P. Pudlák, M. Saks, F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science,* pages 628–637. IEEE Computer Society Press, 1998. 392, 393

[45] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974. 355

[46] M. Rabi, A. Sherman. An observation on associative one-way functions in complexity theory. *Information Processing Letters*, 64(5):239–244, 1997. 364

[47] M. O. Rabin. Probabilistic algorithms for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. 352

[48] R. Rao, J. Rothe, O. Watanabe. Upward separation for FewP and related classes. *Information Processing Letters*, 52(4):175–180, 1994 (Corrigendum appears in the same journal,74(1–2):89, 2000). 393

[49] R. L. Rivest, A. Shamir, L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. 350

[50] J. Rothe. Some facets of complexity theory and cryptography: A five-lecture tutorial. *ACM Computing Surveys*, 34(4):504–549, 2002. 354, 364

[51] J. Rothe. A promise class at least as hard as the polynomial hierarchy. *Journal of Computing and Information*, 1(1):92–107, 1995. 393

[52] J. Rothe. *Complexity Theory and Cryptology. An Introduction to Cryptocomplexity*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2005. 355, 364, 392

[53] A. Salomaa. *Public-Key Cryptography*. EATCS Monographs on Theoretical Computer Science. Vol. 23., Springer-Verlag., 1996 (2nd edition). 364

[54] U. Schöning. A low and a high hierarchy within NP. *Journal of Computer and System Sciences*, 27:14–28, 1983. 380, 383, 393

[55] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1987. 393

[56] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, 410–414 pages. IEEE Computer Society Press, 1999. 376, 392, 393

[57] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001. 393

[58] A. Selman. Polynomial time enumeration reducibility. *SIAM Journal on Computing*, 7(4):440–457, 1978. 392

[59] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992. 359

[60] A. Shamir. RSA for paranoids. *CryptoBytes*, 1(3):1–4, 1995. 364

[61] S. Singh. *The Code Book. The Secret History of Codes and Code Breaking*. Fourth Estate, 1999. 364

[62] R. Solovay, V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977. Erratum appears in the same journal, 7(1):118, 1978. 352

[63] D. Stinson. *Cryptography: Theory and Practice*. CRC Press, 2002 (2nd edition). 355, 364

[64] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977. 393

[65] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, 2:230–265, 1936 (Correction, *ibid*, vol. 43, pages 544–546, 1937). 366

[66] L. G. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976. 393

[67] K. Wagner, G. Wechsung. *Computational Complexity*. D. Reidel Publishing Company, 1986 (and Kluwer Academic Publishers, 2001). 392

[68] G. Wechsung. *Vorlesungen zur Komplexitätstheorie*. Vol. 32. B. G. Teubner Verlagsgesellschaft, 2000. 392

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *ex* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.