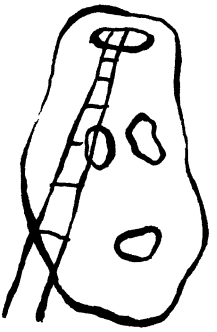
# 2

# Elementary Fundamentals



*"One whoose knowledge*
*is not constantly increasing*
*is not clever at all."*

JEAN PAUL

## 2.1 Introduction

We assume that the reader has had undergraduate courses in mathematics and algorithmics. Despite this assumption we present all elementary fundamentals needed for the rest of this book in this chapter. The main reasons to do this are the following ones:

(i) to make the book completely self-contained in the sense that all arguments needed to design and analyze the algorithms presented in the subsequent chapters are explained in the book in detail,
(ii) to explain the mathematical considerations that are essential in the process of algorithm design,
(iii) to informally explain the fundamental ideas of complexity and algorithm theory and to present their mathematical formalization, and
(iv) to fix the notation in this book.

We do not recommend reading this whole chapter before starting to read the subsequent chapters devoted to the central topic of this book. For every at least a little bit experienced reader we recommend skipping the part about the mathematics and looking up specific results if one needs them later for a particular algorithm design. On the other hand, it is reasonable to read the part about the elementary fundamentals of algorithms because it is strongly connected with the philosophy of this book and the basic notation is fixed there. More detailed recommendations about how to use this chapter are given below.

This chapter is divided into two main parts, namely "Fundamentals of Mathematics" and "Fundamentals of Algorithmics". The part devoted to ele-

J. Hromkovi, *Algorithmics for Hard Problems*

mentary mathematics consists of five sections. Section 2.2.1 contains elementary fundamentals of linear algebra. It focuses on systems of linear equations, matrices, vector spaces, and their geometrical interpretations. The concepts and results presented here are used in Section 3.7 only, where the problem of linear programming and the simplex method are considered. The reader can therefore look at this section when some considerations of Section 3.7 are not clear enough. Section 2.2.2 provides elementary fundamentals of combinatorics, counting, and graph theory. Notions such as permutation and combination are defined and some basic series are presented. Furthermore the notations $O$, $\Omega$, and $\Theta$ for the analysis of the asymptotic growth of functions are fixed and a version of the Master Theorem for solving recurrences is presented. Finally, the basic notions (such as graph, directed graph, multigraph, connectivity, Hamiltonian tour, and Eulerian tour) are defined. The content of Section 2.2.2 is the base for many parts of the book because it is used for the analysis of algorithm complexity as well as for the design of algorithms for graph-theoretical problems. Section 2.2.3 is devoted to Boolean functions and their representations in the forms of Boolean formulae and branching programs. The terminology and basic knowledge presented here are useful for investigating satisfiability problems, which belong to the paradigmatic algorithm problems of main interest. This is the reason why we consider these problems in all subsequent chapters about the algorithm design for hard problems. Section 2.2.4 differs a little bit from the previous sections. While the previous sections are mainly devoted to fixing terminology and presenting some elementary knowledge, Section 2.2.4 contains also some nontrivial, important results like the Fundamental Theorem of Arithmetics, Prime Number Theorem, Fermat's Theorem, and Chinese Remainder Theorem. We included the proofs of these results, too, because the design of randomized algorithms for problems of algorithmic number theory requires a thorough understanding of this topic. If one is not familiar with the contents of this section, it is recommended to look at it before reading the corresponding parts of Chapter 5. Section 2.2.5 is devoted to the elementary fundamentals of probability theory. Here only discrete probability distributions are considered. The main point is that we do not only present fundamental notions like probability space, conditional probability, random variable, and expectation, but we also show their relation to the nature and to the analysis of randomized algorithms. Thus, this part is a prerequisite for Chapter 5 on randomized algorithms.

Section 2.3 is devoted to the fundamentals of algorithm and complexity theory. The main ideas and concepts connected with the primary topic of this book are presented here. It is strongly recommended having at least a short look at this part before starting to read the proper parts of algorithm design techniques for hard problems in Chapters 3, 4, 5, and 6. Section 2.3 is organized as follows. Section 2.3.1 gives the basic terminology of formal language theory (such as alphabet, word, language). This is useful for the representation of data and for the formal description of algorithmic problems and basic concepts of complexity theory. In Section 2.3.2 the formal definitions of all

algorithmic problems considered in this book are given. There, we concentrate on decision problems and optimization problems. Section 2.3.3 provides a short survey on the basic concepts of complexity theory as a theory for classification of algorithmic problems according to their computational difficulty. The fundamental notions such as complexity measurement, nondeterminism, polynomial-time reduction, verifier, and NP-hardness are introduced and discussed in the framework of the historical development of theoretical computer science. This provides the base for starting the effort to solve hard algorithmic problems in the subsequent chapters. Finally, Section 2.3.4 gives a concise overview of the algorithms design techniques (divide-and-conquer, dynamic programming, backtracking, local search, greedy algorithms) that are usually taught in undergraduate courses on algorithmics. All these techniques are later applied to attack particular hard problems. Especially, it is reasonable to read this section before reading Section 3, where these techniques are developed or combined with other ideas in order to obtain solutions to different problems.

## 2.2 Fundamentals of Mathematics

### 2.2.1 Linear Algebra

The aim of this section is to introduce the fundamental notions of linear algebra such as linear equations, matrices, vectors, and vector spaces and to provide elementary knowledge about them. The terminology introduced here is needed for the study of the problem of linear programming and for introducing the simplex method in Section 3.4. Vectors are also often used to represent data (inputs) of many computing problems and matrices are used to represent graphs, directed graphs, and multigraphs in several following sections.

In what follows, we consider the following fundamental sets:

$\mathbb{N} = \{0, 1, 2, \ldots\}$ ... the set of all natural numbers,
$\mathbb{Z} = \{0, -1, 1, -2, 2, \ldots\}$ ... the set of all integers,
$\mathbb{Q} = \{\frac{m}{n} \mid m, n \in \mathbb{Z}, n \neq 0\}$ ... the set of rational numbers,
$\mathbb{R}$ ... the set of real numbers,
$(a, b) = \{x \in \mathbb{R} \mid a < x < b\}$ for all $a, b \in \mathbb{R}$, $a < b$,
$[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$ for all $a, b \in \mathbb{R}$, $a < b$,
For every set $S$, the notation $Pot(S)$ or $2^S$ denotes the power set of $S$, i.e.,

$$Pot(S) = \{Q \mid Q \subseteq S\}.$$

An equation of the type

$$y = ax,$$

expressing the variable $y$ in terms of the variable $x$ for a fixed constant $a$, is called a **linear equation**. The notion of linearity is connected to the geometrical interpretation that corresponds to a straight line.

**Definition 2.2.1.1.** *Let $S$ be a subset of $\mathbb{R}$ such that if $a, b \in S$, then $a+b \in S$ and $a \cdot b \in S$. The equation*

$$y = a_1x_1 + a_2x_2 + \cdots + a_nx_n, \tag{2.1}$$

*where $a_1, \ldots, a_n$ are constants from $S$ and $x_1, x_2, \ldots, x_n$ are variables over $S$ is called a* **linear equation** *over $S$. We say that the equation (2.1) expresses $y$ in terms of variables $x_1, x_2, \ldots, x_n$. The variables $x_1, x_2, \ldots, x_n$ are also called the* **unknowns** *of the linear equation (2.1).*

*For a fixed value of $y$, a* **solution** *to the linear equation (2.1) is a sequence $s_1, s_2, \ldots, s_n$ of numbers of $S$ such that*

$$y = a_1s_1 + a_2s_2 + \cdots + a_ns_n.$$

For instance $1, 1, 1$ (i.e., $x_1 = 1, x_2 = 1, x_3 = 1$) is a solution to the linear equation

$$x_1 + 2x_2 + 3x_3 = 6$$

over $\mathbb{Z}$. Another solution is $-1, -1, 3$.

**Definition 2.2.1.2.** *Let $S$ be a subset of $\mathbb{R}$ such that if $a, b \in S$ then $a \cdot b \in S$ and $a + b \in S$. Let $m$ and $n$ be positive integers. A* **system of $m$ linear equations in $n$ variables (unknowns) $x_1, \ldots, x_n$ over $S$** *(or simply a* **linear system over $S$***) is a set of $m$ linear equations over $S$, where each of these linear equations is expressed in terms of the same variables $x_1, x_2, \ldots, x_n$. In other words, a system of $m$ linear equations over $S$ is*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= y_m, \end{aligned} \tag{2.2}$$

*where $a_{ij}$ are constants from $S$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$, and $x_1, x_2, \ldots, x_n$ are variables (unknowns) over $S$. For each $i \in \{1, \ldots, m\}$, the linear equation*

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = y_i$$

*is called the* **$i$th equation** *of the linear system (2.2). The system (2.2) of linear equations is called* **homogeneous** *if $y_1 = y_2 = \cdots = y_m = 0$.*

*For given values of $y_1, y_2, \ldots, y_m$, a* **solution** *to the linear system (2.2) is a sequence $s_1, s_2, \ldots, s_n$ of numbers of $S$ such that each equation of the linear system (2.2) is satisfied when $x_1 = s_1, x_2 = s_2, \ldots, x_n = s_n$ (i.e., such that $s_1, s_2, \ldots, s_n$ is a solution for each equation of the linear system (2.2)).*

The system of linear equations over $\mathbb{Z}$

$$\begin{aligned} x_1 + 2x_2 &= 10 \\ 2x_1 - 2x_2 &= -4 \\ 3x_1 + 5x_2 &= 26 \end{aligned}$$

is a system of three linear equations in two variables $x_1$ and $x_2$. $x_1 = 2$ and $x_2 = 4$ is a solution to this system.[1]

Note that there are systems of linear equations that do not have any solution. An example is the following linear system:

$$\begin{aligned} x_1 + \ 2x_2 &= 10 \\ x_1 - \ \ x_2 &= -2 \\ 6x_1 + 10x_2 &= 40 \end{aligned}$$

In what follows we define vectors and matrices. They provide a very convenient formalism for representing and manipulating linear systems as well as many other objects in mathematics and computer science.

**Definition 2.2.1.3.** *Let $S \subseteq \mathbb{R}$ be any set satisfying $a + b \in S$ and $a \cdot b \in S$ for all $a, b \in S$. Let $n$ and $m$ be positive integers. An $\boldsymbol{m \times n}$* **matrix** *$\boldsymbol{A}$* **over** *$\boldsymbol{S}$ is a rectangular array of $m \cdot n$ elements of $S$ arranged in $m$ horizontal* **rows** *and $n$ vertical* **columns***:*

$$A = [a_{ij}]_{i=1,\ldots,m,j=1,\ldots,n} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}.$$

*For all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$, $a_{ij}$ is called the* **$(i,j)$-entry**[2] *of $A$. The* **$i$th row** *of $A$ is*

$$(a_{i1}, a_{i2}, \ldots, a_{in})$$

*for all $i \in \{1, \ldots, m\}$. For each $j \in \{1, \ldots, n\}$, the* **$j$th column** *of $A$ is*

$$\begin{pmatrix} a_{1j} \\ a_{2j} \\ a_{3j} \\ \vdots \\ a_{mj} \end{pmatrix}.$$

*For any positive integers $n$ and $m$, a $1 \times n$ matrix is called an* **$n$-dimensional row-vector**, *and a $m \times 1$ matrix is called an* **$m$-dimensional column-vector**.

[1] We assume that the reader is familiar with the method of elimination that efficiently finds a solution of linear systems. We do not present the method here, because it is not interesting to us from the algorithmic point of view.

[2] Observe that $a_{ij}$ lies on the intersection of the $i$th row and the $j$th column.

*For any positive integer $n$, an $n \times n$ matrix is called a* **square matrix of order $n$.** *If $A = [a_{ij}]_{i,j=1,\ldots,n}$ is a square matrix, then we say that the elements $a_{11}, a_{22}, \ldots, a_{nn}$ form the* **main diagonal** *of $A$.*

*$A$ is called the* **1-diagonal matrix** *(or the* **identity matrix***), denoted by $\boldsymbol{I_n}$, if*

*(i) $a_{ii} = 1$ for $i = 1, \ldots, n$, and*
*(ii) $a_{ij} = 0$ for all $i \neq j$, $i, j \in \{1, \ldots, n\}$.*

*$A$ is called the* **0-diagonal matrix** *if*

*(i) $a_{ii} = 0$ for all $i = 1, \ldots, n$, and*
*(ii) $a_{ij} = 1$ for $i \neq j$, $i, j \in \{1, \ldots, n\}$.*

*An $m \times n$ matrix $B = [b_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$ is called a* **Boolean matrix** *if $b_{ij} \in \{0, 1\}$ for $i = 1, \ldots, m$, $j = 1, \ldots, n$.*

*An $m \times n$ matrix $B = [b_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$ is called a* **zero matrix** *if $b_{ij} = 0$ for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$. The zero matrix of size $m \times n$ is denoted by $\boldsymbol{0_{m \times n}}$.*

The following matrix $B = [b_{ij}]_{i=1,\ldots,3,j=1,\ldots,4}$ is an example of a $3 \times 4$ matrix over $\mathbb{Q}$.

$$B = \begin{pmatrix} 1 & \frac{2}{3} & 4 & -6 \\ \frac{1}{2} & 1 & \frac{3}{4} & -8 \\ -3 & \frac{6}{5} & 2 & 0 \end{pmatrix}.$$

$\left(\frac{1}{2}, 1, \frac{3}{4}, -8\right)$ is the second row of $B$. The $(3,4)$-entry of $B$ is $b_{34} = 0$.

**Definition 2.2.1.4.** *Let $m, n$ be two positive integers. Two $m \times n$ matrices $A = [a_{ij}]$ and $B = [b_{ij}]$ are said to be* **equal** *if $a_{ij} = b_{ij}$ for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$.*

*The* **sum** *of $A$ and $B$ $\boldsymbol{(A + B)}$ is the matrix $C = [c_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$ defined by*[3]

$$c_{ij} = a_{ij} + b_{ij}$$

*for all $i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$.*

**Exercise 2.2.1.5.** Let $A, B, C$ be matrices over $\mathbb{R}$ of same size $m \times n$, $m, n \in \mathbb{N} - \{0\}$. Prove that

(i) $A + B = B + A$,
(ii) $A + (B + C) = (A + B) + C$,
(iii) there exists a **negative** of $A$ (i.e., there exists a $m \times n$ matrix $D = (-A)$ such that $A + D = 0_{m \times n}$).

□

[3] Observe that the sum of the matrices is defined only when $A$ and $B$ have the same number of rows and the same number of columns.

**Definition 2.2.1.6.** *Let $m, p, n$ be positive integers. Let*

$$A = [a_{ij}]_{i=1,\ldots,m, j=1,\ldots,p} \text{ and } B = [b_{ij}]_{i=1,\ldots,p, j=1,\ldots,n}$$

*be two matrices. The* **product (multiplication)** *of $A$ and $B$ is the $m \times n$ matrix $C = [c_{ij}]_{i=1,\ldots,m, j=1,\ldots,n}$ defined by*

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ik}b_{kj}$$

*for $i = 1, \ldots, m$, $j = 1, \ldots, n$.*

To illustrate Definition 2.2.1.6 consider the matrices

$$A = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 3 & -1 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 & -3 \\ 5 & 0 \\ 0 & 4 \end{pmatrix}.$$

Then,

$$A \cdot B = \begin{pmatrix} 1 \cdot 1 + 0 \cdot 5 + (-2) \cdot 0 & 1 \cdot (-3) + 0 \cdot 0 + (-2) \cdot 4 \\ 0 \cdot 1 + 3 \cdot 5 + (-1) \cdot 0 & 0 \cdot (-3) + 3 \cdot 0 + (-1) \cdot 4 \end{pmatrix} = \begin{pmatrix} 1 & -11 \\ 15 & -4 \end{pmatrix}.$$

Observe that $B \cdot A$ is not defined because the product of $B$ and $A$ is defined only if the number of columns of $B$ is equal to the number of rows of $A$.

**Exercise 2.2.1.7.** Find two squared matrices $A$ and $B$ over $\mathbb{Z}$ such that $A \cdot B \neq B \cdot A$. □

**Exercise 2.2.1.8.** Let $A, B, C$ be $m \times p$, $p \times q$, $q \times n$ matrices over $\mathbb{R}$, respectively. Prove:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C.$$

□

**Exercise 2.2.1.9.** Prove that for every $n \times n$ matrix $A$,

$$A \cdot I_n = I_n \cdot A = A.$$

□

**Definition 2.2.1.10.** *Let $r$ be a real number, and let $A = [a_{ij}]$ be an $m \times n$ matrix over $\mathbb{R}$. The* **scalar multiple of $A$ by $r$**, *$r \cdot A$, is the $m \times n$ matrix $B = [b_{ij}]$, where*

$$b_{ij} = r \cdot a_{ij}$$

*for $i = 1, \ldots, m$, $j = 1, \ldots, n$.*

**Definition 2.2.1.11.** *Let $A = [a_{ij}]$ be an $m \times n$ matrix, $m, n \in \mathbb{N} - \{0\}$. The $n \times m$ matrix $B = [b_{ij}]_{i=1,\ldots,n, j=1,\ldots,m}$, where*

$$a_{ij} = b_{ji} \text{ for } i = 1, \ldots, m, j = 1, \ldots, n$$

*is called the* **transpose of $A$** *and denoted by $A^{\mathsf{T}}$. If $A = A^{\mathsf{T}}$, then we say that $A$ is* **symmetric**.

**Exercise 2.2.1.12.** Prove, for every real number $r$ and any matrices $A$ and $B$ over $\mathbb{R}$, that the following equalities hold:

(i) $(A^{\mathsf{T}})^{\mathsf{T}} = A$,
(ii) $(A+B)^{\mathsf{T}} = A^{\mathsf{T}} + B^{\mathsf{T}}$,
(iii) $(A \cdot B)^{\mathsf{T}} = B^{\mathsf{T}} \cdot A^{\mathsf{T}}$,
(iv) $(rA)^{\mathsf{T}} = r \cdot A^{\mathsf{T}}$.

□

Now we show how matrices can be used to represent systems of linear equations. Consider the system (2.2) of Definition 2.2.1.2. Define

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}.$$

Then, the linear system (2.2) can be written in the matrix form

$$A \cdot X = Y.$$

The matrix $A$ is called the **coefficient matrix** of the linear system (2.2).

For instance, for the system of linear equations

$$\begin{aligned} -x_1 + 2x_2 - 3x_3 &= 7 \\ 6x_1 + \ x_2 + \ x_3 &= 5 \end{aligned}$$

the coefficient matrix is

$$A = \begin{pmatrix} -1 & 2 & -3 \\ 6 & 1 & 1 \end{pmatrix} \text{ and } X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, Y = \begin{pmatrix} 7 \\ 5 \end{pmatrix}.$$

**Definition 2.2.1.13.** *Let $A$ be a squared $n \times n$ matrix, $n \in \mathbb{N} - \{0\}$. $A$ is called* **nonsingular** *(or* **invertible***) if there exists an $n \times n$ matrix $B$ such that*

$$A \cdot B = B \cdot A = I_n.$$

*The matrix $B$ is called the* **inverse of $A$** *and denoted*[4] *by* $\mathbf{A^{-1}}$. *If there exists no inverse of $A$, then $A$ is called* **singular** *(or* **noninvertible***).*

One can easily verify that, for

$$A = \begin{pmatrix} 2 & 3 \\ 2 & 2 \end{pmatrix} \text{ and } B = \begin{pmatrix} -1 & \frac{3}{2} \\ 1 & -1 \end{pmatrix},$$

$A \cdot B = B \cdot A = I_2$, and so $A^{-1} = B$ and $B^{-1} = A$. Observe also that $I_n^{-1} = I_n$ for any positive integer $n$.

[4] Note that if, for a matrix $A$, there exists a matrix $B$ with the property $A \cdot B = B \cdot A = I_n$, then $B$ is the unique inverse of $A$.

**Exercise 2.2.1.14.** Prove the following assertion. If $A_1, A_2, \ldots, A_r$ are $n \times n$ nonsingular matrices, then $A_1 \cdot A_2 \cdot \cdots \cdot A_r$ is nonsingular, and

$$(A_1 \cdot A_2 \cdot \cdots \cdot A_r)^{-1} = A_r^{-1} \cdot A_{r-1}^{-1} \cdot \cdots \cdot A_1^{-1}. \qquad \square$$

Let $A \cdot X = Y$ be a system of linear equations where the coefficient matrix $A$ is an $n \times n$ nonsingular matrix. Then one can solve this system by constructing $A^{-1}$ because multiplying the equality

$$A \cdot X = Y$$

by $A^{-1}$ from the left side one obtains

$$A^{-1} \cdot A \cdot X = A^{-1} \cdot Y.$$

Since $A^{-1} \cdot A = I_n$ and $I_n \cdot X = X$ we obtain

$$X = A^{-1} \cdot Y.$$

Now we look at the geometrical interpretation of systems of linear equations.

**Definition 2.2.1.15.** *For any positive integer $n$, we define the* **$n$-dimensional ($\mathbb{R}$-) vector space**

$$\mathbb{R}^n = \left\{ \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \;\middle|\; a_i \in \mathbb{R} \text{ for } i = 1, \ldots, n \right\}.$$

*The vector $0_{n \times 1}$ is called the* **origin** *of $\mathbb{R}^n$.*

There are two possible geometrical interpretations of the elements of $\mathbb{R}^n$. One possibility is to assign to an element

$$X = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$$

of $\mathbb{R}^n$ the point with the coordinates $a_1, a_2, \ldots, a_n$ in $\mathbb{R}^n$. Another possibility is to assign a directed line from $(0, 0, \ldots, 0)^\mathsf{T}$ to $(a_1, \ldots, a_n)^\mathsf{T}$. This directed line is called the **vector** $(a_1, \ldots, a_n)^\mathsf{T}$.

Consider $\mathbb{R}^2$. To build a geometrical interpretation of $\mathbb{R}^2$ one starts from the origin $(0, 0)^\mathsf{T}$. One draws two infinite lines which are orthogonal to each other and which intersect at the origin. One of the lines is usually in a horizontal position and is called the ***x*-axis**. The other infinite line, the ***y*-axis**, is taken in a vertical position (see Figure 2.1). Then, the positive reals are

placed on the $x$-axis to the right of the origin in increasing order and the negative reals are placed on the $x$-axis to the left in decreasing order. Similarly, the $y$-axis above the origin contains the positive real numbers and the $y$-axis below the origin contains the negative real numbers. For any point $X$ of the plane one can determine the **coordinates** of $X$ as follows:

(i) Take a line $l$ that contains the point $X$ and is orthogonal (perpendicular) to the $x$-axis (parallel to the $y$-axis). The real number $a_x$ associated with the intersection of the $x$-axis and $l$ is the ***x*-coordinate** of $X$.

(ii) Take a line $h$ that contains the point $X$ and is orthogonal to the $y$-axis (parallel to the $x$-axis). The real number $a_y$ associated with the intersection of the $y$-axis and $h$ is the ***y*-coordinate** of $X$.

We shall denote the point $X$ by $P(a_x, a_y)$, and the corresponding vector by $(a_x, a_y)^{\mathsf{T}}$.

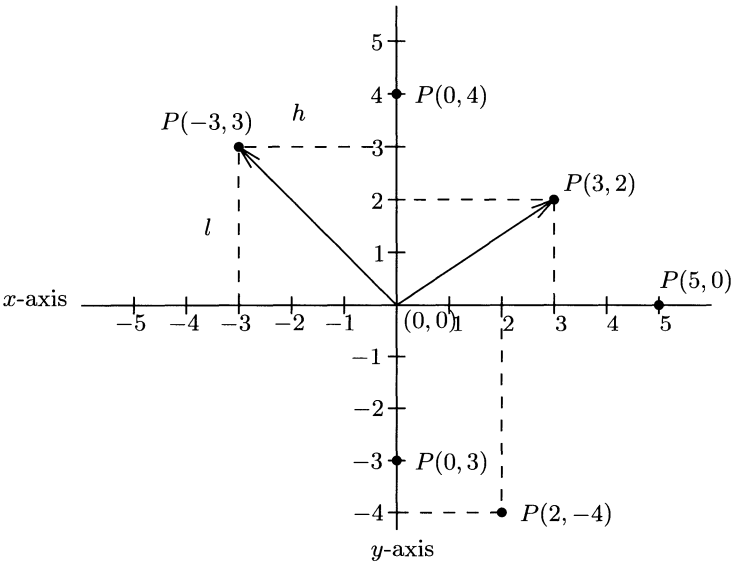



**Fig. 2.1.**

**Definition 2.2.1.16.** *Let* $P(a_1, a_2)$ *and* $P(b_1, b_2)$ *be two points in* $\mathbb{R}^2$. *The* **(Euclidean) distance** *between* $P(a_1, a_2)$ *and* $P(b_1, b_2)$ *is defined by*

$$\textbf{\textit{distance}}\,(\boldsymbol{P(a_1, a_2), P(b_1, b_2)}) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}.$$

We observe from Figure 2.2 that the Euclidean distance between two points is exactly the length of the line that connects $P(a_1, a_2)$ and $P(b_1, b_2)$ because of the Pythagorean Theorem.

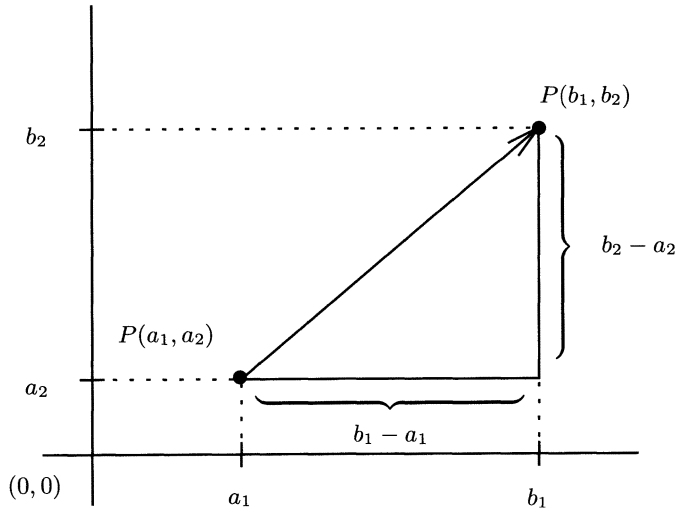



**Fig. 2.2.**

**Exercise 2.2.1.17.** Prove that, for every three points $P(a_1, a_2)$, $P(b_1, b_2)$, and $P(c_1, c_2)$,

(i) $distance(P(a_1, a_2), P(a_1, a_2)) = 0$,
(ii) $distance(P(a_1, a_2), P(b_1, b_2)) = distance(P(b_1, b_2), P(a_1, a_2))$, and
(iii) $distance(P(a_1, a_2), P(b_1, b_2)) \leq distance(P(a_1, a_2), P(c_1, c_2))$
$+distance(P(c_1, c_2), P(b_1, b_2))$.

□

We can find a natural interpretation of systems of linear equations of two variables in $\mathbb{R}^2$. One can assign to every linear equation

$$a_1 x_1 + a_2 x_2 = b$$

with $(a_1, a_2) \neq (0, 0)$ the straight line $x_1 = \frac{b - a_2 x_2}{a_1} = \frac{b}{a_1} - \frac{a_2}{a_1} \cdot x_2$ if $a_1 \neq 0$ and the straight line $x_2 = \frac{b}{a_2}$ if $a_1 = 0$. This straight line consists of all points of $\mathbb{R}^2$ that satisfy the given linear equation. So, the set of all solutions to a system of linear equations is exactly the intersection of all straight lines corresponding to the particular equations.

Figure 2.3 contains four lines corresponding to the linear equations $x_1 + 3x_2 = 8$, $x_1 + 3x_2 = 4$, $x_1 + 0 \cdot x_2 = 2$, and $x_1 - x_2 = 0$. The system of linear equations

$$\begin{aligned} x_1 + 3x_2 &= 4 \\ x_1 - \ x_2 &= 0 \end{aligned}$$

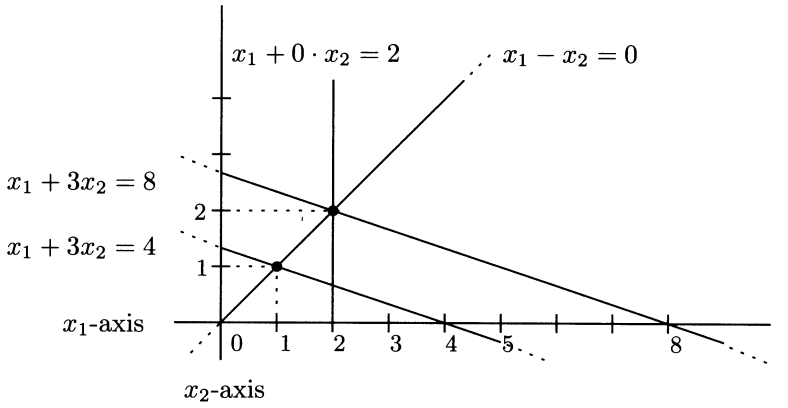



**Fig. 2.3.**

has the unique solution $P(1,1)$ (i.e., $x_1 = 1$, $x_2 = 1$) that is the intersection point of the lines corresponding to these equations. The system of linear equations

$$\begin{aligned} x_1 + 3x_2 &= 4 \\ x_1 + 3x_2 &= 8 \end{aligned}$$

does not have any solution because the corresponding lines do not intersect.[5] On the other hand, the system of linear equations

$$\begin{aligned} x_1 + 3x_2 &= 4 \\ 2x_1 + 6x_2 &= 8 \end{aligned}$$

has infinitely many solutions because both linear equalities $x_1 + 3x_2 = 4$ and $2x_1 + 6x_2 = 8$ define the same line, and so every point of this line is a solution. Finally, we observe that the system of linear equations

$$\begin{aligned} x_1 + \quad 3x_2 &= 8 \\ x_1 + 0 \cdot x_2 &= 2 \\ x_1 - \quad x_2 &= 0 \end{aligned}$$

has exactly one solution $P(2,2)$ (i.e., $x_1 = 2$, $x_2 = 2$), and that the set of linear equations

$$\begin{aligned} x_1 + \quad 3x_2 &= 4 \\ x_1 + 0 \cdot x_2 &= 2 \\ x_1 - \quad x_2 &= 0 \end{aligned}$$

[5] Because they are parallel.

does not have any solution.

We see that any nontrivial linear equation of two variables determines a line that is a one-dimensional part of $\mathbb{R}^2$. In general, any linear equation over $n$ variables determines an $(n-1)$-dimensional subpart[6] of $\mathbb{R}^n$. To understand it geometrically, we present some elementary fundamentals of the theory of vector spaces.

**Definition 2.2.1.18.** *Let $W \subseteq \mathbb{R}^n$, $n \in \mathbb{N}-\{0\}$. We say that $W$ is a* **(linear) vector subspace of $\mathbb{R}^n$** *if, for all $r_1, r_2 \in \mathbb{R}$ and all $(a_1, a_2, \ldots, a_n)^\mathsf{T}$, $(b_1, b_2, \ldots, b_n)^\mathsf{T} \in W$,*

$$r_1 \cdot (a_1, a_2, \ldots, a_n)^\mathsf{T} + r_2 \cdot (b_1, b_2, \ldots, b_n)^\mathsf{T} \in W.$$

Note that every vector subspace of $\mathbb{R}^n$ contains the origin $0_{n\times 1} = (0, \ldots, 0)^\mathsf{T}$, because one can set $r_1 = r_2 = 0$.

Let $V = \{(a_1, a_2, 0)^\mathsf{T} \mid a_1, a_2 \in \mathbb{R}\}$. We observe that $V$ is a vector subspace of $\mathbb{R}^3$ because

$$r_1 \cdot (b_1, b_2, 0)^\mathsf{T} + r_2 \cdot (d_1, d_2, 0)^\mathsf{T} = (r_1 b_1 + r_2 d_1, r_1 b_2 + r_2 d_2, 0)^\mathsf{T} \in V$$

for all real numbers $r_1, r_2, b_1, b_2, d_1, d_2$.

**Definition 2.2.1.19.** *Let $A = [a_{ij}]_{i=1,\ldots,m, j=1,\ldots,n}$ be a matrix and let $X = (x_1, x_2, \ldots, x_n)^\mathsf{T}$. For every homogeneous linear system $AX = 0_{1\times m}$, we define the set of solutions to $A \cdot X = 0_{1\times m}$ as*

$$\boldsymbol{Sol(A)} = \{Y \in \mathbb{R}^n \mid A \cdot Y = 0_{m\times 1}\}.$$

*Analogously, for every $A$ and every $b \in \mathbb{R}^m$, the set of solutions to $A \cdot X = b$ is*

$$\boldsymbol{Sol(A, b)} = \{Y \in \mathbb{R}^n \mid A \cdot Y = b\}.$$

**Lemma 2.2.1.20.** *Let $A \cdot X = 0_{n\times 1}$ be a system of linear equations where $A$ is an $m \times n$ matrix, $m, n \in \mathbb{N} - \{0\}$. The set $Sol(A)$ of all solutions to the linear system $A \cdot X = 0_{n\times 1}$ is a vector subspace of $\mathbb{R}^n$.*

*Proof.* $X = 0_{n\times 1}$ is a solution of every homogeneous system of linear equations of $n$ variables; thus, $Sol(A)$ is not empty. Let $X_1$ and $X_2$ be arbitrary vectors from $Sol(A)$, and let $r_1$ and $r_2$ be arbitrary reals. We have to prove that $r_1 X_1 + r_2 X_2 \in Sol(A)$. This can be done by the following simple calculation.

$$\begin{aligned} A(r_1 X_1 + r_2 X_2) &= A r_1 X_1 + A r_2 X_2 \\ &= r_1 A X_1 + r_2 A X_2 \\ &= r_1 \cdot 0_{n\times 1} + r_2 \cdot 0_{n\times 1} = 0_{n\times 1}. \end{aligned}$$

□

[6] Later, we shall see it defined by the term "affine subspace" (the term "manifold" is used in some literature, too).

The **trivial vector subspace** of $\mathbb{R}^n$ is $\{0_{n\times 1}\}$. Obviously, there is no non-trivial vector subspace $W$ of $\mathbb{R}^n$ with a finite cardinality. This is because for each $X \in W$, $X \neq 0_{n\times 1}$, the infinite set

$$\{r \cdot X \mid r \in \mathbb{R}\} \subseteq W.$$

**Definition 2.2.1.21.** *Let $X, X_1, X_2, \ldots, X_k$ be vectors from $\mathbb{R}^n$, $n, k \in \mathbb{N} - \{0\}$. The vector $X$ is a* **linear combination of the vectors $X_1, X_2, \ldots, X_k$** *if there exist real numbers $c_1, c_2, \ldots, c_k$ such that*

$$X = c_1X_1 + c_2X_2 + \cdots + c_kX_k.$$

For instance, $(4, 2, 10, -10)^{\mathsf{T}}$ is a linear combination of the vectors $(1, 2, 1, -1)^{\mathsf{T}}$, $(1, 0, 2, -3)^{\mathsf{T}}$, and $(1, 1, 0, -2)^{\mathsf{T}}$ because

$$(4, 2, 10, -10)^{\mathsf{T}} = 2 \cdot (1, 2, 1, -1)^{\mathsf{T}} + 4 \cdot (1, 0, 2, -3)^{\mathsf{T}} - 2 \cdot (1, 1, 0, -2)^{\mathsf{T}}.$$

**Definition 2.2.1.22.** *Let $S = \{X_1, X_2, \ldots, X_k\} \subseteq \mathbb{R}^n$ be a set of nonzero vectors, and let $W$ be a subset of $\mathbb{R}^n$, $k \in \mathbb{N}, n \in \mathbb{N} - \{0\}$.*

*We say that $S$* **spans** *$W$ if every vector from $W$ is a linear combination of vectors from $S$. The trivial vector subspace $\{0_{n\times 1}\} \subseteq \mathbb{R}^n$ is spanned by the empty set $S$.*

*The set $S$ is called* **linearly dependent** *if there exist reals $c_1, c_2, \ldots, c_k$ not all zero, such that*

$$c_1X_1 + c_2X_2 + \cdots + c_kX_k = 0_{n\times 1}.$$

*Otherwise, $S$ is called* **linearly independent** *(i.e., the equality $c_1X_1 + c_2X_2 + \cdots + c_kX_k = 0_{n\times 1}$ holds only for $c_1 = c_2 = \cdots = c_k = 0$).*

*The set $S$ is called a* **basis** *of a vector subspace $U \subseteq \mathbb{R}^n$, if*

*(i) $S$ spans $U$, and*
*(ii) $S$ is linearly independent.*

For any subspace $V \subseteq \mathbb{R}^n$ we say that the **dimension** of $V$ is a $k \in \mathbb{N}$ if there exists a set of vectors $S \subseteq \mathbb{R}^n$ such that

(i) $|S| = k$, and
(ii) $S$ is a basis of $V$.

We say also that $V$ is a $k$-dimensional subspace of $\mathbb{R}^n$, or that $\boldsymbol{dim(V) = k}$.

For instance, $(1, 0, 0, 0)^{\mathsf{T}}$, $(0, 1, 0, 0)^{\mathsf{T}}$, $(0, 0, 1, 0)^{\mathsf{T}}$, $(0, 0, 0, 1)^{\mathsf{T}}$ is a basis of $\mathbb{R}^4$ because

(i) $(a, b, c, d)^{\mathsf{T}} = a \cdot (1, 0, 0, 0)^{\mathsf{T}} + b \cdot (0, 1, 0, 0)^{\mathsf{T}} + c \cdot (0, 0, 1, 0)^{\mathsf{T}} + d \cdot (0, 0, 0, 1)^{\mathsf{T}}$ for each vector $(a, b, c, d)^{\mathsf{T}} \in \mathbb{R}^4$, and
(ii) $c_1 \cdot (1, 0, 0, 0)^{\mathsf{T}} + c_2 \cdot (0, 1, 0, 0)^{\mathsf{T}} + c_3 \cdot (0, 0, 1, 0)^{\mathsf{T}} + c_4 \cdot (0, 0, 0, 1)^{\mathsf{T}} = (0, 0, 0, 0)^{\mathsf{T}}$ if and only if $c_1 = c_2 = c_3 = c_4 = 0$.

Let us consider the following linear equation

$$a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n = b \tag{2.3}$$

of $n$ variables (unknowns) $x_1, x_2, \ldots, x_n$. We can also write (2.3) as $A \cdot X = b$, where $A = (a_1, a_2, \ldots, a_n)$ and $X = (x_1, x_2, \ldots, x_n)^{\mathsf{T}}$.

**Lemma 2.2.1.23.** *For every linear homogeneous equation $A \cdot X = 0$, where $A = (a_1, \ldots, a_n) \neq (0, \ldots, 0)$, $X = (x_1, \ldots, x_n)^{\mathsf{T}}$, $Sol(A)$ is an $(n-1)$-dimensional subspace of $\mathbb{R}^n$.*

*Proof.* Without loss of generality we assume $a_1 \neq 0$. Then

$$x_1 = -\frac{a_2}{a_1}x_2 - \frac{a_3}{a_1}x_3 - \cdots - \frac{a_n}{a_1}x_n$$

is another form of the linear equation (2.3) for $b = 0$. So, $Sol(A) =$

$$\left\{ \left( -\frac{a_2}{a_1}y_2 - \frac{a_3}{a_1}y_3 - \cdots - \frac{a_n}{a_1}y_n, y_2, y_3, \ldots, y_n \right)^{\mathsf{T}} \;\middle|\; y_2, y_3, \ldots, y_n \in \mathbb{R} \right\}$$

is the set of all solutions of $AX = 0$. We verify this assumption by the following calculation:

$$(a_1, a_2, \ldots, a_n) \cdot \left( -\frac{a_2}{a_1}y_2 - \frac{a_3}{a_1}y_3 - \cdots - \frac{a_n}{a_1}y_n, y_2, y_3, \ldots, y_n \right)^{\mathsf{T}} =$$
$$(-a_2y_2 - a_3y_3 - \cdots - a_ny_n) + a_2y_2 + a_3y_3 + \cdots + a_ny_n = 0.$$

Now we claim that

$$S = \left\{ \left( -\frac{a_2}{a_1}, 1, 0, \ldots, 0 \right)^{\mathsf{T}}, \left( -\frac{a_3}{a_1}, 0, 1, 0, \ldots, 0 \right)^{\mathsf{T}}, \ldots, \left( -\frac{a_n}{a_1}, 0, \ldots, 0, 1 \right)^{\mathsf{T}} \right\}$$

is a basis of $Sol(A)$. $S$ spans $Sol(A)$ because

$$\left( -\frac{a_2}{a_1}y_2 - \frac{a_3}{a_1}y_3 - \cdots - \frac{a_n}{a_1}y_n, y_2, y_3, \ldots, y_n \right)^{\mathsf{T}} =$$
$$y_2 \left( -\frac{a_2}{a_1}, 1, 0, \ldots, 0 \right)^{\mathsf{T}} + y_3 \left( -\frac{a_3}{a_1}, 0, 1, \ldots, 0 \right)^{\mathsf{T}} + \cdots + y_n \left( -\frac{a_n}{a_1}, 0, \ldots, 1 \right)^{\mathsf{T}}$$

for all $y_2, y_3, \ldots, y_n \in \mathbb{R}$.

It remains to be shown that the vectors of $S$ are linearly independent. Assume that

$$c_1 \left( -\frac{a_2}{a_1}, 1, 0, \ldots, 0 \right)^{\mathsf{T}} + c_2 \left( -\frac{a_3}{a_1}, 0, 1, \ldots, 0 \right)^{\mathsf{T}} + \cdots$$
$$+ c_{n-1} \left( -\frac{a_n}{a_1}, 0, \ldots, 0, 1 \right)^{\mathsf{T}} = 0.$$

Then, particularly

$$\begin{aligned} c_1 \cdot 1 &= 0 \\ c_2 \cdot 1 &= 0 \\ &\vdots \\ c_{n-1} \cdot 1 &= 0. \end{aligned}$$

Thus, $c_1 = c_2 = \cdots = c_{n-1} = 0$. □

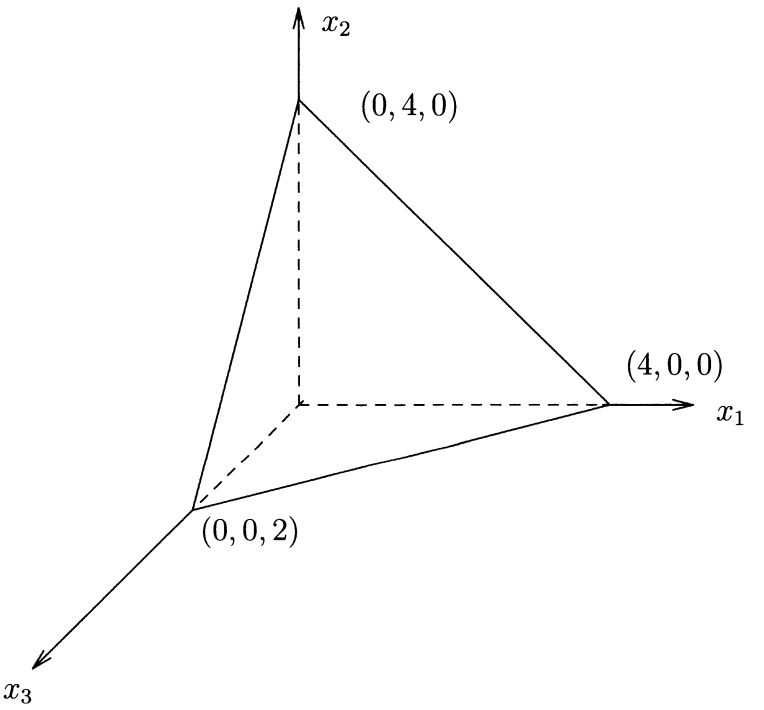



**Fig. 2.4.**

If one considers the vector space $\mathbb{R}^2$, then the set of solutions of a linear equation $c_1x_1 + c_2x_2 = 0$ with $(c_1, c_2) \neq (0,0)$ is the set of all points of the corresponding straight line $x_1 = \frac{c_2}{c_1} \cdot x_2$. Examples are presented in Figure 2.3. In $\mathbb{R}^3$ the set of all solutions of a linear homogeneous equation is a two-dimensional subspace of $\mathbb{R}^3$ (Fig. 2.4).

As we have already observed, the set $Sol(A)$ of solutions $Y$ to $A \cdot Y = 0_{m\times 1}$ is a subspace of $\mathbb{R}^n$. The question is what is the dimension of $Sol(A)$. Obviously, if $A$ is not a zero matrix, the dimension of $Sol(A)$ is at most $n-1$.

**Theorem 2.2.1.24.** *Let $U$ be a subset of $\mathbb{R}^n$, $n \in \mathbb{N} - \{0\}$. $U$ is a subspace of $\mathbb{R}^n$ if and only if there exists a matrix $A$ such that $U = Sol(A)$.*

*Proof.* We have already mentioned that $Sol(A)$ is a subspace of $\mathbb{R}^n$ if $A$ is an $m \times n$ matrix, $m, n \in \mathbb{N} - \{0\}$. It remains to be shown that, for every subspace $U \subseteq \mathbb{R}^m$, there exists such an $A$ that $U = Sol(A)$.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a basis of $U$, $S_i = (s_{i1}, \ldots, s_{in})$ for $i = 1, 2, \ldots, m$. Now we construct an $m \times n$ matrix $A = [a_{ij}]$ such that

$$A \cdot S_k^{\mathsf{T}} = 0_{m \times 1} \tag{2.4}$$

for every $k = 1, 2, \ldots, m$. Setting $S = [s_{ij}]_{i=1,\ldots,m, j=1,\ldots,n}$ we can express (2.4) as

$$A \cdot S^{\mathsf{T}} = 0_{m \times m}. \tag{2.5}$$

But (2.5) is equivalent to the requirement

$$(a_{l1}, a_{l2}, \ldots, a_{ln}) \cdot S^{\mathsf{T}} = 0_{1 \times m} \tag{2.6}$$

for every $l \in \{1, \ldots, m\}$. So the $l$th equation of (2.6) can be seen as a system of linear equations of $n$ unknowns $a_{l1}, a_{l2}, \ldots, a_{ln}$. Solving this linear system one determines the values $a_{l1}, a_{l2}, \ldots, a_{ln}$. Doing it for every $l$, the matrix $A$ is determined. It remains to be proven that $U = Sol(A)$. Next we prove $U \subseteq Sol(A)$. The opposite direction $Sol(A) \subseteq U$ is left to the reader.

As $\mathcal{S}$ is a basis of $U$, we have, for every $X \in U$,

$$X = c_1 S_1^{\mathsf{T}} + c_2 S_2^{\mathsf{T}} + \cdots + c_m S_m^{\mathsf{T}}$$

for some $c_1, c_2, \ldots, c_m \in R$. Thus

$$\begin{aligned} A \cdot X &= A \cdot \left(c_1 S_1^{\mathsf{T}} + c_2 S_2^{\mathsf{T}} + \cdots + c_m S_m^{\mathsf{T}}\right) \\ &= c_1 \cdot A \cdot S_1^{\mathsf{T}} + c_2 \cdot A \cdot S_2^{\mathsf{T}} + \cdots + c_m \cdot A \cdot S_m^{\mathsf{T}} \\ &= c_1 \cdot 0_{m \times 1} + c_2 \cdot 0_{m \times 1} + \cdots + c_m \cdot 0_{m \times 1} \\ &\overset{(2.4)}{=} 0_{m \times 1}, \end{aligned}$$

and so $X \in Sol(A)$. □

**Definition 2.2.1.25.** *Let $A$ be an $n \times m$ matrix $[a_{ij}]_{i=1,\ldots,n, j=1,\ldots,m}$. Let $A_i = (a_{i1}, a_{i2}, \ldots, a_{im})^{\mathsf{T}}$ for $i = 1, \ldots, n$, and let $U$ be the subspace of $\mathbb{R}^n$ that is spanned by $\{A_1, A_2, \ldots, A_n\}$. We define the* **rank** *of the matrix $A$ as*

$$\boldsymbol{rank(A)} = dim(U).$$

Obviously, $rank(I_n) = n$ for every positive integer $n$. The rank of the following matrix

$$M = \begin{pmatrix} 1 & 2 & 0 & 3 \\ 2 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 4 \end{pmatrix}$$

is three. This is because

$$(0,1,-1,4) = 1 \cdot (1,2,0,3) - 1 \cdot (2,1,1,0) + 1 \cdot (1,0,0,1),$$

and the fact that the set of vectors $\{(1,2,0,3),(2,1,1,0),(1,0,0,1)\}$ is linearly independent.

**Exercise 2.2.1.26.** Let $A = [a_{ij}]_{i=1,\ldots,n,j=1,\ldots,m}$, $n,m \in \mathbb{N} - \{0\}$. Let $S_i = (a_{i1}, a_{i2}, \ldots, a_{im})^T$ for $i = 1,\ldots,n$, and let $C_j = (a_{1j}, a_{2j}, \ldots, a_{nj})^T$ for $j = 1,\ldots,m$. Let $U$ be the subspace of $\mathbb{R}^m$ spanned by $\{S_1,\ldots,S_n\}$ and $V$ be a subspace of $\mathbb{R}^n$ that is spanned by $\{C_1,\ldots,C_m\}$. Prove that

$$dim(U) = dim(V). \qquad \square$$

**Exercise 2.2.1.27.** Let $B = [b_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$ be an $m \times n$ matrix, $n,m \in \mathbb{N} - \{0\}$. Prove that

$$dim(B) = n - rank(B). \qquad \square$$

**Definition 2.2.1.28.** *Let $U$ be a subspace of $\mathbb{R}^n$, $n \in \mathbb{N} - \{0\}$, and let $C \in \mathbb{R}^n$. The vector set*

$$V = \{X \in \mathbb{R}^n \mid X = C + Y, Y \in U\}$$

*is called an* **affine subspace** *of $\mathbb{R}^n$ translated by $C$ from $U$.*

**Observation 2.2.1.29.** Let $U$ be a subspace of $\mathbb{R}^n$, $n \in \mathbb{N} - \{0\}$. For each $C \in \mathbb{R}^n$,

$$dim\left(\{X \in \mathbb{R}^n \mid X = C + Y, Y \in U\}\right) = dim(U).$$

The following theorem presents the main relation between systems of linear equations and affine subspaces. We omit its technical proof because it does not contain any idea that would be interesting for an algorithm design in this book.

**Theorem 2.2.1.30.** *Let $n$ be a positive integer, and let $U$ be a subset of $\mathbb{R}^n$. $U$ is an affine subspace of $R^n$ iff there exist $m \in \mathbb{N}$, $m \leq n$, an $m \times n$ matrix $A$, and a vector $b \in \mathbb{R}^m$ such that $U = Sol(A,b)$.*

In $\mathbb{R}^3$ the set of all solutions of a linear equation is a two-dimensional subspace of $\mathbb{R}^3$, called also a **plane**. Figure 2.4 contains the plane corresponding to the linear equation $x_1 + x_2 + 2x_3 = 4$. This plane is unambiguously given by the three points $(4,0,0)$, $(0,4,0)$, $(0,0,2)$ in which it crosses the axes of $\mathbb{R}^3$. The line $x_1 + x_2 = 4$ is the intersection of the plane $x_3 = 0$ with the plane

$x_1 + x_2 + 2x_3 = 4$. The line $x_2 + 2x_3 = 4$ is the intersection of the plane $x_1 = 0$ and the plane $x_1 + x_2 + 2x_3 = 4$, and the line $x_1 + 2x_3 = 4$ is the intersection of the plane $x_2 = 0$ and the plane $x_1 + x_2 + 2x_3 = 4$.

A transparent way to work with vector subspaces is to view them as convex sets.

**Definition 2.2.1.31.** *Let $X$ and $Y$ be two points in $\mathbb{R}^n$. A* **convex combination of $X$ and $Y$** *is any point*

$$Z = c \cdot X + (1 - c) \cdot Y$$

*for any real number $c$, $0 \leq c \leq 1$. If $c \notin \{0, 1\}$, then we say that $c$ is a* **strict** *convex combination of $X$ and $Y$.*

Observe that the set

$$\boldsymbol{Convex(X, Y)} = \{Z \in \mathbb{R}^n \mid Z = c \cdot X + (1 - c) \cdot Y \text{ for a } c \in \mathbb{R}, 0 \leq c \leq 1\}$$

is exactly the set of points of the straight line that connects $X$ and $Y$.

**Definition 2.2.1.32.** *Let $n$ be a positive integer. A set $S \subseteq \mathbb{R}^n$ is convex if, for all $X, Y \in S$, $S$ contains all convex combinations of $X$ and $Y$ (i.e., $Convex(X, Y) \subseteq S$ for all $X, Y \in S$).*

The trivial example of convex sets in $\mathbb{R}^n$ are $\mathbb{R}^n$, the empty set, and any singleton set. The set of solutions of a linear equation $a_1x_1 + a_2x_2 = b$ in $\mathbb{R}^2$ is convex. An important property of convex sets is expressed in the following assertion.

**Theorem 2.2.1.33.** *The intersection of any number of convex sets is a convex set.*

*Proof.* Let $\bigcap_{i \in I} S_i$ be the intersection of convex sets $S_i$ for $i \in I$. If $X, Y \in \bigcap_{i \in I} S_i$, then $X$ and $Y$ are in every $S_i$. Any convex combination of $X$ and $Y$ is then in $S_i$ for every $i \in I$, and therefore in $\bigcap_{i \in I} S_i$. □

Now we show that the set of solutions of any linear equation

$$a_1x_1 + a_2x_2 + \cdots + a_nx_x = b$$

of $n$ unknowns $x_1, \ldots, x_n$ is a convex set in $\mathbb{R}^n$. Let $Y = (y_1, \ldots, y_n)^\mathsf{T}$ and $W = (w_1, \ldots, w_n)^\mathsf{T}$ be arbitrary solutions of this linear equation, i.e.,

$$\sum_{i=1}^{n} a_i y_i = b = \sum_{i=1}^{n} a_i w_i.$$

Let

$$Z = c \cdot Y + (1 - c) \cdot W = (cy_1 + (1 - c)w_1, \ldots, cy_n + (1 - c)w_n)$$

be an arbitrary convex combination of $Y$ and $W$, $0 \leq c \leq 1$. We prove that $Z$ is a solution to the linear equation

$$\sum_{i=1}^{n} a_i x_i = b,$$

too.

$$\begin{aligned}(a_1, \ldots, a_n) \cdot Z &= \sum_{i=1}^{n} a_i(cy_i + (1-c)w_i) \\ &= c \cdot \sum_{i=1}^{n} a_i y_i + (1-c) \cdot \sum_{i=1}^{n} a_i w_i \\ &= c \cdot b + (1-c) \cdot b = b.\end{aligned}$$

Because of the above fact and Theorem 2.2.1.33 we obtain that $Sol(A, b)$ is a convex set for every system of linear equations $A \cdot X = b$.

### 2.2.2 Combinatorics, Counting, and Graph Theory

The aim of this section is to give the definitions of some fundamental objects of combinatorics and graph theory, and to present a few elementary results about them. The terms introduced in this section are useful for the analysis of algorithms as well as for the representation of discrete objects in the subsequent chapters.

More precisely, we first introduce the fundamental categories of combinatorics such as permutation and combination and learn to work with them. Then we define the $O$, $\Omega$, and $\Theta$ notation for the study of the asymptotic behavior of functions, and present some simple summations of some fundamental series, and a simplified version of the Master Theorem for solving some specific recurrences. After that we give the fundamental notions of graph theory such as graphs, multigraphs, directed graphs, planarity, connectivity, matching, cut, etc.

First, we define the basic terms of permutation and combination. The starting point is to have a set of objects that are distinguishable from each other.

**Definition 2.2.2.1.** *Let $n$ be a positive integer. Let $S = \{a_1, a_2, \ldots, a_n\}$ be a set of $n$ objects (elements). A* **permutation of $n$ objects $a_1, \ldots, a_n$** *is an ordered arrangement of the objects of $S$.*

For instance, if $S = \{a_1, a_2, a_3\}$, then there are the following six different ways to arrange the three objects $a_1$, $a_2$, $a_3$:

$$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1), (a_3, a_1, a_2), (a_3, a_2, a_1).$$

To denote permutations the simple notation $(i_1, i_2, \ldots, i_n)$ is often used instead of $(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$.

**Lemma 2.2.2.2.** *For every positive integer $n$, the number* $\boldsymbol{n!}$ *of different permutations of $n$ objects is*

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 = \prod_{i=1}^{n} i.$$

*Proof.* The first object in any permutation may be chosen in $n$ different ways (i.e., from $n$ different objects). Once the first object has been chosen, the second object may be chosen in $n-1$ different ways (i.e., from $n-1$ remaining objects), etc. □

By arrangement, we use the notation $\boldsymbol{0!} = 1$.

**Definition 2.2.2.3.** *Let $k$ and $n$ be non-negative integers, $k \leq n$. A* **combination of $k$ objects from $n$ objects** *is a selection of $k$ objects without regard to order.*

A combination of four objects from $\{a_1, a_2, a_3, a_4, a_5\}$ is one of the following sets:

$$\{a_1, a_2, a_3, a_4\}, \{a_1, a_2, a_3, a_5\}, \{a_1, a_2, a_4, a_5\}, \{a_1, a_3, a_4, a_5\}, \{a_2, a_3, a_4, a_5\}.$$

**Lemma 2.2.2.4.** *Let $n$ and $k$ be non-negative integers, $k \leq n$. The number* $\binom{n}{k}$ *of combinations of $k$ objects from $n$ objects is*

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{k!} = \frac{n!}{k! \cdot (n-k)!}.$$

*Proof.* Similar to the proof of Lemma 2.2.2.2, we have $n$ possibilities for the choice of the first element, $n-1$ possibilities of the choice of the second element, etc. Hence, there are

$$n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)$$

ways of choosing $k$ objects from $n$ objects when order is taken into account. But any order of these $k$ elements provides the same set of $k$ elements. So,

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{k!}.$$

□

Observe that $\binom{n}{0} = \binom{n}{n} = 1$.

**Corollary 2.2.2.5.** *For all non-negative integers $k$ and $n$, $k \leq n$,*

$$\binom{n}{k} = \binom{n}{n-k}.$$

**Lemma 2.2.2.6.** *For all positive integers $k$ and $n$, $k \leq n$,*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

*Proof.*

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)! \cdot (n-k)!} + \frac{(n-1)!}{k! \cdot (n-k-1)!} \\ &= \frac{k \cdot (n-1)! + (n-k) \cdot (n-1)!}{k! \cdot (n-k)!} \\ &= \frac{n \cdot (n-1)!}{k! \cdot (n-k)!} = \frac{n!}{k! \cdot (n-k)!} = \binom{n}{k}.\end{aligned}$$

□

The values $\binom{n}{k}$ are also known as the **binomial coefficients** from the following theorem.

**Theorem 2.2.2.7 (Newton's Theorem).** *For every positive integer $n$,*

$$\begin{aligned}(1+x)^n &= \binom{n}{0} + \binom{n}{1} \cdot x + \binom{n}{2} \cdot x^2 + \cdots + \binom{n}{n-1} \cdot x^{n-1} + \binom{n}{n} \cdot x^n \\ &= \sum_{i=0}^{n} \binom{n}{i} \cdot x^i.\end{aligned}$$

**Exercise 2.2.2.8.** Prove Newton's Theorem. □

**Lemma 2.2.2.9.** *For every positive integer $n$,*

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

*Proof.* To prove Lemma 2.2.2.9 it is sufficient to set $x = 1$ in Newton's Theorem. Another argument is that, for each $k \in \{0, 1, \ldots, n\}$, $\binom{n}{k}$ is the number of all $k$-element subsets of an $n$-element set. So, $\sum_{k=0}^{n} \binom{n}{k}$ counts the number of all subsets of a set of $n$ elements, and every set of $n$ elements has exactly $2^n$ different subsets. □

**Exercise 2.2.2.10.** Prove, for every integer $n \geq 3$,

$$\binom{n}{2} = \prod_{l=3}^{n} \frac{l}{l-2}.$$

□

Next, we fix some fundamental notations concerning elementary functions, and look briefly at the asymptotic behavior of functions.

For any positive real number $x$, we denote the greatest integer less than or equal to $x$ by $\lfloor x \rfloor$ and call it the **floor of** $x$. For $x \in \mathbb{R}^+$, the **ceiling of** $x$, denoted by $\lceil x \rceil$, is the least integer greater than or equal to $x$. We observe that

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

for each $x \in \mathbb{R}^+$.

Let $x$ be a variable, and let $d$ be a positive integer. A **polynomial in** $x$ **of degree** $d$ is a function $p(x)$ of the form

$$p(x) = \sum_{i=0}^{d} a_i x^i,$$

where the constants $a_0, a_1, \ldots, a_d$ are the **coefficients** of the polynomial and $a_d \neq 0$.

We use $e$ to denote $\lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$. Note that $e$ is the base of the natural logarithm function.

**Exercise 2.2.2.11.** Prove that, for all reals $x$,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

□

The assertion of Exercise 2.2.2.11 implies

$$1 + x \leq e^x \leq 1 + x + x^2$$

for every $x \in [-1, 1]$. Note that $e = 2.7182\ldots$, and one can approximate $e$ with an arbitrary precision by using the equation of Exercise 2.2.2.11.

**Exercise 2.2.2.12.** Prove that for all real $x$,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$

□

In this book we use $\log n$ to denote the binary logarithm $\log_2 n$ and $\ln n = \log_e n$ to denote the natural logarithm. The equalities of the following exercise provide the elementary rules for working with logarithmic functions.

**Exercise 2.2.2.13.** Prove, for all positive reals $a, b, c$ and $n$,

(i) $\log_c(ab) = \log_c(a) + \log_c(b)$,

(ii) $\log_c a^n = n \cdot \log_c a$,
(iii) $a^{\log_b n} = n^{\log_b a}$, and
(iv) $\log_b a = \frac{1}{\log_a b}$. □

In algorithmics we work with functions from $\mathbb{N}$ to $\mathbb{N}$ in order to measure complexity according to the input size. Here, we are often concerned with how the complexity (running time, for instance) increases with the input size in the limit as the size of the input increases without bound. In this case we say that we are studying the **asymptotic** efficiency of algorithms. This rough characterization of the complexity growth by the order of its growth is usually sufficient to determine the threshold on the input size above which the algorithm is not applicable because of a too huge complexity. In what follows we define the standard asymptotic notation used in algorithmics.

**Definition 2.2.2.14.** *Let $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define*

$$\begin{aligned}
\boldsymbol{O(f(n))} &= \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \exists c, n_0 \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq n_0 : \\
&\qquad t(n) \leq c \cdot f(n)\}. \\
\boldsymbol{\Omega(f(n))} &= \{g : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \exists d, n_0 \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq n_0 : \\
&\qquad g(n) \geq \frac{1}{d} \cdot f(n)\}. \\
\boldsymbol{\Theta(f(n))} &= O(f(n)) \cap \Omega(f(n)) \\
&= \{h : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \exists c_1, c_2, n_0 \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq n_0 : \\
&\qquad \frac{1}{c_1} \cdot f(n) \leq h(n) \leq c_2 \cdot f(n)\}.
\end{aligned}$$

*If $t(n) \in O(f(n))$ we say that* ***t* does not grow asymptotically faster than *f***. *If $g(n) \in \Omega(f(n))$ we say that* ***g* grows asymptotically at least as fast as *f***. *If $h(n) \in \Theta(f(n))$ we say that* ***h* and *f* are asymptotically equivalent**.

**Exercise 2.2.2.15.** Let $p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$ be a polynomial in $n$ for some positive integer $d$ and a positive real number $a_d$. Prove that

$$p(n) \in \Theta(n^d).$$

□

In the literature one usually uses the notation $t(n) = O(f(n))$, $g(n) = \Omega(f(n))$, and $h(n) = \Theta(f(n))$, respectively, instead of $t(n) \in O(f(n))$, $g(n) \in \Omega(f(n))$, and $h(n) \in \Theta(f(n))$, respectively.

**Exercise 2.2.2.16.** Which of the following statements are true? Prove your answers.

(i) $2^n \in \Theta\left(2^{n+a}\right)$ for any positive integer (constant) $a$.

(ii) $2^{b \cdot n} \in \Theta(2^n)$ for any positive integer (constant) $b$.
(iii) $\log_b n \in \Theta(\log_c n)$ for all $b, c \in \mathbb{R}^{>1}$.
(iv) $(n+1)! \in O(n!)$.
(v) $\log_2(n!) \in \Theta(n \cdot \log n)$.

□

In the next part of this section we remind the reader of some elementary series and their sums. For any function $f : \mathbb{N} \to \mathbb{R}$, one can define

$$\boldsymbol{Sum_f(n)} = \sum_{i=1}^{n} f(i) = f(1) + f(2) + \cdots + f(n).$$

$Sum_f(n)$ is called a **series of** $\boldsymbol{f}$. In what follows we consider only some fundamental kinds of series.

**Definition 2.2.2.17.** *Let $a$, $b$, and $d$ be some constants. For every function $f : \mathbb{N} \to \mathbb{R}$, defined by $f(n) = a + (n-1) \cdot d$, $Sum_f(n)$ is called an* **arithmetic series**. *For every function $h : \mathbb{N} \to \mathbb{R}$ defined by $h(n) = a \cdot b^{n-1}$, $Sum_h(n)$ is called a* **geometric series**.

**Lemma 2.2.2.18.** *Let $a$ and $d$ be some constants. Then*

$$Sum_{a+(n-1)\cdot d}(n) = \sum_{i=1}^{n}(a + (i-1) \cdot d) = an + \frac{d \cdot (n-1) \cdot n}{2}.$$

*Proof.*

$$\begin{aligned}
\sum_{i=1}^{n}(a + (i-1) \cdot d) &= \sum_{i=1}^{n} a + d \cdot \sum_{i=1}^{n}(i-1) \\
&= a \cdot n + d \cdot \frac{1}{2}\left(2 \cdot \sum_{i=1}^{n}(i-1)\right) \\
&= a \cdot n + \frac{d}{2}\left(\sum_{i=1}^{n}(i-1) + \sum_{j=n}^{1}(j-1)\right) \\
&= a \cdot n + \frac{d}{2} \cdot \sum_{i=1}^{n}[((n-1) - (i-1)) + (i-1)] \\
&= a \cdot n + \frac{d}{2} \cdot \sum_{i=1}^{n}(n-1) = a \cdot n + \frac{d}{2} \cdot (n-1) \cdot n.
\end{aligned}$$

□

**Lemma 2.2.2.19.** *Let $a$ and $b$ be some constants, $b \in \mathbb{R}^+$, $b \neq 1$. Then*

$$Sum_{a \cdot b^{n-1}}(n) = \sum_{i=1}^{n} a \cdot b^{i-1} = a \cdot \frac{1 - b^n}{1 - b}.$$

*Proof.*

$$Sum_{a\cdot b^{n-1}}(n) = \sum_{i=1}^{n} a \cdot b^{i-1} = a\left(1 + b + b^2 + \cdots + b^{n-1}\right). \tag{2.7}$$

Thus,

$$b \cdot Sum_{a\cdot b^{n-1}}(n) = a \cdot \left(b + b^2 + \cdots + b^n\right). \tag{2.8}$$

Subtracting the equality (2.8) from the equality (2.7) we obtain

$$(1-b) \cdot Sum_{a\cdot b^{n-1}}(n) = a\left(1 - b^n\right),$$

which directly implies the assertion of Lemma 2.2.2.19. □

**Exercise 2.2.2.20.** Prove that for any $b \in (0,1)$,

$$\lim_{n\to\infty} Sum_{a\cdot b^{n-1}}(n) = \sum_{i=1}^{\infty} a \cdot b^{i-1} = a \cdot \frac{1}{1-b}.$$

□

**Definition 2.2.2.21.** *For every positive integer $n$, the* **nth harmonic number** *is defined by the series*

$$\boldsymbol{Har(n)} = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

First, we observe that $Har(n)$ tends to infinity with growing $n$. The simplest way to see it is to partition the term of $Har(n)$ into infinitely many groups of cardinality $2^k$ as follows:

$$\underbrace{\frac{1}{1}}_{\text{group 1}} + \underbrace{\frac{1}{2}+\frac{1}{3}}_{\text{group 2}} + \underbrace{\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}}_{\text{group 3}}$$
$$+\underbrace{\frac{1}{8}+\frac{1}{9}+\frac{1}{10}+\frac{1}{11}+\frac{1}{12}+\frac{1}{13}+\frac{1}{14}+\frac{1}{15}}_{\text{group 4}}+\cdots.$$

Both terms in group 2 are between $\frac{1}{4}$ and $\frac{1}{2}$, and so the sum of that group is between $2 \cdot \frac{1}{4} = \frac{1}{2}$ and $2 \cdot \frac{1}{2} = 1$. All four terms in group 3 are between $\frac{1}{8}$ and $\frac{1}{4}$, and so their sum is also between $4 \cdot \frac{1}{8} = \frac{1}{2}$ and $4 \cdot \frac{1}{4} = 1$. In general, for every positive integer $k$, all $2^{k-1}$ terms of group $k$ are between $2^{-k}$ and $2^{-k+1}$ and hence the sum of the terms of group $k$ is between $\frac{1}{2} = 2^{k-1} \cdot 2^{-k}$ and $1 = 2^{k-1} \cdot 2^{-k+1}$.

This grouping procedure shows us that if $n$ is in group $k$, then $Har(n) > k/2$ and $Har(n) \leq k$. Thus,

$$\frac{\lfloor \log_2 n \rfloor}{2} + \frac{1}{2} < Har(n) \leq \lfloor \log_2 n \rfloor + 1.$$

**Exercise 2.2.2.22.** (*) Prove that

$$Har(n) = \ln n + O(1).$$

□

**Definition 2.2.2.23.** *For any sequence* $a_0, a_1, \ldots, a_n$, $\sum_{k=1}^{n} (a_k - a_{k-1})$ *and* $\sum_{i=0}^{n-1} (a_i - a_{i+1})$ *are called* **telescoping series.**

Obviously,

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0$$

since each of the terms $a_1, a_2, \ldots, a_{n-1}$ is added exactly once and subtracted out exactly once. Analogously,

$$\sum_{i=1}^{n-1} (a_i - a_{i+1}) = a_1 - a_n.$$

The reason to consider telescoping series is that one can easily simplify a series if one recognizes that the series is telescoping. For instance, consider the series

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}.$$

Since $\frac{1}{k\cdot(k+1)} = \frac{1}{k} - \frac{1}{k+1}$ we get

$$\sum_{k=1}^{n-1} \frac{1}{k \cdot (k+1)} = \sum_{k=1}^{n-1} \left( \frac{1}{k} - \frac{1}{k+1} \right) = 1 - \frac{1}{n}.$$

Analyzing the complexity of algorithms one often reduces the analysis to solving a specific recurrence. The typical recurrences are of the form

$$T(n) = a \cdot T\left(\frac{n}{c}\right) + f(n),$$

where $a$ and $c$ are positive integers and $f$ is a function from $\mathbb{N}$ to $\mathbb{R}^+$. In what follows we give a general solution of this recurrence if $f(n) \in \Theta(n)$.

**Theorem 2.2.2.24 (Master Theorem).** *Let* $a$, $b$, *and* $c$ *be positive integers. Let*

$$T(1) = 0,$$
$$T(n) = a \cdot T\left(\frac{n}{c}\right) + b \cdot n.$$

*Then,*

$$T(n) \in \begin{cases} O(n) & \text{if } a < c \\ O(n \log n) & \text{if } a = c \\ O\left(n^{\log_c a}\right) & \text{if } c < a. \end{cases}$$

*Proof.* For simplicity, we assume $n = c^k$ for some positive integer $k$.

$$\begin{aligned} T(n) &= a \cdot T\left(\frac{n}{c}\right) + b \cdot n \\ &= a \cdot \left[a \cdot T\left(\frac{n}{c^2}\right) + b \cdot \frac{n}{c}\right] + b \cdot n \\ &= a^2 \cdot T\left(\frac{n}{c^2}\right) + b \cdot \left(\frac{a}{c} \cdot n + n\right) \\ &= a^k \cdot T(1) + b \cdot n \cdot \sum_{i=0}^{k-1} \left(\frac{a}{c}\right)^i \\ &= bn \cdot \left(\sum_{i=0}^{(\log_c n)-1} \left(\frac{a}{c}\right)^i\right). \end{aligned}$$

Now we distinguish the three cases according to the relation between $a$ and $c$.

(1) Let $a < c$. Then, following Exercise 2.2.2.20 we obtain

$$\sum_{i=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^i \leq \sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i = \frac{1}{1 - \frac{a}{c}} \in O(1).$$

Thus, $T(n) \in O(n)$.

(2) Let $a = c$. Obviously,

$$\sum_{i=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^i = \log_c n \in O(\log n).$$

So, $T(n) \in O(n \log n)$.

(3) Let $a > c$. Following Lemma 2.2.2.19 we obtain

$$\begin{aligned} bn \cdot \sum_{i=0}^{\log_c n - 1} \left(\frac{a}{c}\right)^i &= bn \cdot \left(\frac{1 - \left(\frac{a}{c}\right)^{\log_c n}}{1 - \frac{a}{c}}\right) \\ &= \frac{b}{\frac{a}{c} - 1} \cdot n \cdot \left(\left(\frac{a}{c}\right)^{\log_c n} - 1\right) \in O\left(n \cdot \frac{a^{\log_c n}}{c^{\log_c n}}\right) \\ &= O\left(a^{\log_c n}\right) = O\left(n^{\log_c a}\right). \end{aligned}$$

□

In what follows we present the fundamental terminology of graph theory.

**Definition 2.2.2.25.** *An* **(undirected) graph** $G$ *is a pair* $(V, E)$, *where*

*(i)* $V$ *is a finite set called the* **set of vertices** *of* $G$, *and*
*(ii)* $E$ *is a subset of* $\{\{u, v\} \mid v, u \in V \text{ and } v \neq u\}$ *called the* **set of edges** *of* $G$.

*Any element of* $V$ *is called a* **vertex** *of* $G$, *and any element of* $E$ *is called an* **edge** *of* $G$. $G$ *is called a* **graph of** $|V|$ **vertices**.

An example of a graph is $G' = (V, E)$, where

$$V = \{v_1, v_2, v_3, v_4, v_5\} \text{ and } E = \{\{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_4\}, \{v_2, v_5\}\}.$$

One usually represents a graph as a picture in the plane. The vertices are points (or small circles) in the plane and an edge $\{u, v\}$ is represented as a curve connecting the points $u$ and $v$. The graph $G'$ is depicted in Figure 2.5 in two different ways.

A graph $G$ is called **planar** if there exists such a picture representation of $G$ in the plane that the curves representing the edges do not cross in any point of the plane. The graph $G'$ defined above is planar because Figure 2.5b provides its planar representation.

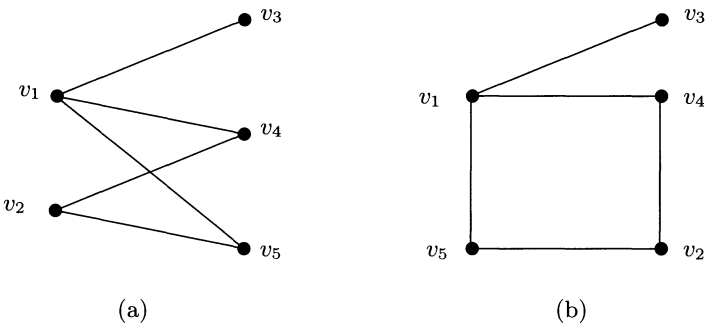



**Fig. 2.5.**

We observe that we have either an edge between two vertices $u$ and $v$ or no edge between $u$ and $v$. So, another suitable representation of a graph of $n$ vertices $v_1, v_2, \ldots, v_n$ is by a symmetric $n \times n$ Boolean matrix $M_G[a_{ij}]_{i,j=1,\ldots,n}$, called the **adjacency matrix** of $G$, where $a_{ij} = 1$ iff $\{v_i, v_j\} \in E$. The following matrix represents the graph $G'$ depicted in Figure 2.5.

$$\begin{pmatrix} 0\,0\,1\,1\,1 \\ 0\,0\,0\,1\,1 \\ 1\,0\,0\,0\,0 \\ 1\,1\,0\,0\,0 \\ 1\,1\,0\,0\,0 \end{pmatrix}$$

For any edge $\{u, v\}$ of a graph $G = (V, E)$ we say that $\{u, v\}$ is **incident** to the vertices $u$ and $v$. Two vertices $x$ and $y$ are **adjacent** if the edge $\{x, y\}$ belongs to $E$. The **degree of a vertex** $v$ of $G$, $\boldsymbol{deg_G(v)}$, is the number of edges incident to $v$. The **degree of a graph** $G = (V, E)$ is

$$\boldsymbol{deg(G)} = \max\{deg_G(v) \mid v \in V\}.$$

**Exercise 2.2.2.26.** Prove that for every graph $G = (V, E)$,

$$\sum_{v \in V} deg_G(v) = 2 \cdot |E|.$$

□

**Exercise 2.2.2.27.** How many graphs of $n$ vertices $v_1, v_2, \ldots, v_n$ exist? □

**Definition 2.2.2.28.** *Let $G = (V, E)$ be a graph. A* **path** *in $G$ is a sequence of vertices $P = v_1, v_2, \ldots, v_m$, $v_i \in V$ for $i = 1, \ldots, m$, such that $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, m-1$. For $i = 1, \ldots, m$, $v_i$ is a* **vertex of $P$**, *and for $j = 1, \ldots, m-1$, $\{v_j, v_{j+1}\}$ is an* **edge of $P$**. *The* **length of $P$** *is the number of its edges (i.e., $m-1$).*

*A path $P = v_1, v_2, \ldots, v_m$ is called* **simple** *if either all its vertices are distinct (i.e., $|\{v_1, v_2, \ldots, v_m\}| = m$) or all its vertices but $v_1$ and $v_m$ are distinct (i.e., $|\{v_1, \ldots, v_m\}| = m-1$ and $v_1 = v_m$).*

*A path $P = v_1, v_2, \ldots, v_m$ is called a* **cycle** *if $v_1 = v_m$. A* **simple cycle** *is a simple path that is a cycle. A simple cycle that contains all vertices of the graph $G$ is called a* **Hamiltonian tour** *of $G$. A cycle that contains all edges of $G$ is called an* **Eulerian tour** *of $G$. If a graph contains a Hamiltonian tour, then it is called* **Hamiltonian**.

Obviously, any path $P = v_1, v_2, \ldots, v_m$ can be viewed as a graph $(\{v_1, \ldots, v_m\}, \{\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{m-1}, v_m\}\})$.

$P = v_1, v_4, v_2, v_5, v_1, v_4, v_2$ is a path of the graph $G'$ from Figure 2.5. $P$ is not simple. The path $v_1, v_4, v_2, v_5, v_1$ is a simple cycle. $G'$ does not contain any Eulerian tour nor any Hamiltonian tour.

**Exercise 2.2.2.29.** Show that if a graph contains a path between two vertices $u$ and $v$, then it contains a simple path between $u$ and $v$. □

For any $n \in \mathbb{N}$, $\boldsymbol{K_n} = (\{v_1, \ldots, v_n\}, \{\{v_i, v_j\} \mid i, j \in \{1, \ldots, n\}, i \neq j\})$ is the **complete** graph of $n$ vertices. Observe that $M_{K_n}$ is the 0-diagonal $n \times n$ Boolean matrix (i.e., the matrix whose diagonal elements are all 0s and non-diagonal elements are all 1s.) A graph $G = (V, E)$ is called **connected** if, for all $x, y \in V$, $x \neq y$, there exists a path between $x$ and $y$ in $G$. $G$ is called **bipartite** if one can partition $V$ into two sets $V_1$ and $V_2$ ($V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$) such that $E \subseteq \{\{u, v\} \mid u \in V_1, v \in V_2\}$. The graph $G'$ in Figure 2.5 is a connected, bipartite graph. The second property can be viewed especially well in Figure 2.5a with $V_1 = \{v_1, v_2\}$ on the left side and $V_2 = \{v_3, v_4, v_5\}$ on the right side.

**Exercise 2.2.2.30.** Prove that for every connected graph $G = (V, E)$,

$$|E| \geq |V| - 1.$$

□

**Exercise 2.2.2.31.** Prove that a connected graph $G$ contains an Eulerian tour if and only if the degree of all vertices of $G$ is even. □

**Definition 2.2.2.32.** *A* **cut** *of a graph $G = (V, E)$ is a triple $(V_1, V_2, E')$, where*

*(i)* $V_1 \cup V_2 = V$, $V_1 \neq \emptyset$, $V_2 \neq \emptyset$, *and* $V_1 \cap V_2 = \emptyset$, *and*
*(ii)* $E' = E \cap \{\{u, v\} \mid u \in V_1, v \in V_2\}$.

Obviously, to determine a cut in a given graph $G = (V, E)$ it is sufficient to give $(V_1, V_2)$ or $E'$ only. For instance, $E' = \{\{v_1, v_5\}, \{v_2, v_4\}\}$ determines the cut $(\{v_1, v_3, v_4\}, \{v_2, v_5\}, E')$ of the graph $G'$ in Figure 2.5.

**Definition 2.2.2.33.** *Let $G = (V, E)$ be a graph. A* **matching** *of $G$ is any subset $M$ of $E$ such that if $\{x, y\}$ and $\{u, v\}$ are two different elements of $M$, then $\{x, y\} \cap \{u, v\} = \emptyset$. A matching $M$ is called a* **maximal matching** *if, for each edge $\{r, s\} \in E - M$, $M \cup \{r, s\}$ is not a matching.*

For instance, $\{\{v_1, v_3\}\}$, $\{\{v_1, v_3\}, \{v_2, v_5\}\}$, and $\{\{v_1, v_5\}, \{v_2, v_4\}\}$ are matchings of $G'$ in Figure 2.5. The last two are maximal matchings in $G'$.

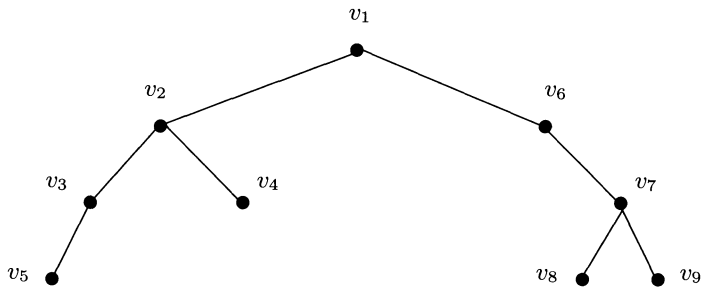



**Fig. 2.6.**

**Definition 2.2.2.34.** *A graph $G = (V, E)$ is called* **acyclic** *if it does not contain any cycle. An acyclic, connected graph is called a* **tree**. *A* **rooted tree** *$T$ is a tree in which one of the vertices is distinguished from the others. This distinguished vertex is called the* **root** *of the tree.*

*Any vertex $u$ different from the root is called a* **leaf (external vertex)** *of the rooted tree $T$ if $deg_T(u) = 1$. A vertex $v$ of $T$ with $deg_T(v) > 1$ is called an* **internal vertex.**

The tree $T = (\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}, \{\{v_1, v_2\}, \{v_1, v_6\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_5\}, \{v_6, v_7\}, \{v_7, v_8\}, \{v_7, v_9\}\})$ is depicted in Figure 2.6. If $v_1$ is considered to be the root of $T$, then $v_4, v_5, v_8$, and $v_9$ are the leaves of $T$. If $v_9$ is taken as the root of $T$, then $v_4, v_5$, and $v_8$ are the leaves of this rooted tree.

If a tree $T$ is a rooted tree with a root $w$, then we usually denote the tree $T$ by $\boldsymbol{T_w}$.

**Definition 2.2.2.35.** *A* **multigraph** $G$ *is a pair* $(V, H)$ *where*

*(i)* $V$ *is a finite set called the* **set of vertices** *of* $G$*, and*
*(ii)* $H$ *is a multiset of elements from* $\{\{u, v\} \mid u, v \in V, u \neq v\}$ *called the* **set of multiedges** *of* $G$.

An example of a multigraph is $G_1 = (V, H)$, where $V = \{v_1, v_2, v_3, v_4\}$, $H = \{\{v_1, v_2\}, \{v_1, v_2\}, \{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{v_1, v_3\}, \{v_3, v_4\}\}$. A possible picture representation is given in Figure 2.7.

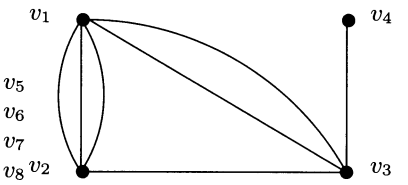



**Fig. 2.7.**

A multigraph $G$ of $n$ vertices $v_1, v_2, \ldots, v_n$ can be represented by a symmetric $n \times n$ matrix $M_G = [b_{ij}]_{i,j=1,\ldots,n}$ where $b_{ij}$ is the number of edges between $v_i$ and $v_j$. $M_G$ is called the **adjacency matrix** of $G$.

The following matrix is the adjacency matrix of the multigraph $G_1$ in Figure 2.7.

$$\begin{pmatrix} 0\,3\,2\,0 \\ 3\,0\,1\,0 \\ 2\,1\,0\,1 \\ 0\,0\,1\,0 \end{pmatrix}$$

All notions such as degree, path, cycle, connectivity, matching can be defined for multigraphs in the same way as for graphs, and so we omit their formal definitions. For any two graphs or multigraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_1, E_2)$ we say that $G_1$ is a **subgraph** of $G_2$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

In what follows we define the directed graphs and some fundamental notions connected with them. Informally, a directed graph differs from a graph in having a direction (from one vertex to another vertex) on every edge.

**Definition 2.2.2.36.** *A* **directed graph** $G$ *is a pair* $(V, E)$*, where*

*(i)* $V$ *is a finite set called the* **set of vertices** *of* $G$*, and*
*(ii)* $E \subseteq (V \times V) - \{(v, v) \mid v \in V\} = \{(u, v) \mid u \neq v, u, v \in V\}$ *is the* **set of (directed) edges** *of* $G$.

*If* $(u, v) \in E$ *then we say that* $(u, v)$ **leaves** *the vertex* $u$ *and that* $(u, v)$ **enters** *the vertex* $v$*. We also say that* $(u, v)$ *is* **incident** *to the vertices* $u$ *and* $v$.

The directed graph $G_2 = (\{v_1, v_2, v_3, v_4, v_5, v_6\}, \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_4, v_2), (v_4, v_5), (v_5, v_3)\})$ is depicted in Figure 2.8. The edge $(v_2, v_4)$ leaves the vertex $v_2$ and enters the vertex $v_4$.

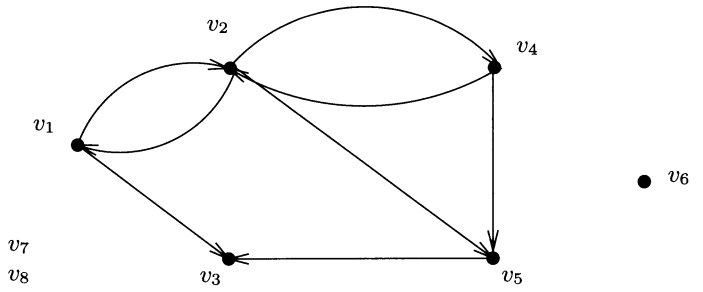



**Fig. 2.8.**

Again, we can use the notion of an adjacency matrix to represent a directed graph. For any directed graph $G = (V, E)$ of $n$ vertices $v_1, \ldots, v_n$, the **adjacency matrix of $G$**, $\boldsymbol{M_G} = [c_{ij}]_{i,j=1,\ldots,n}$, is defined by

$$c_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ 0 \text{ if } (v_i, v_j) \notin E. \end{cases}$$

The following matrix is the adjacency matrix of the directed graph $G_2$ depicted in Figure 2.8.

$$M_{G_2} = \begin{pmatrix} 0\,1\,1\,0\,0\,0 \\ 1\,0\,0\,1\,1\,0 \\ 0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0 \end{pmatrix}$$

**Exercise 2.2.2.37.** How many directed graphs of $n$ vertices $v_1, v_2, \ldots, v_n$ exist? □

Let $G = (V, E)$ be a directed graph, and let $v$ be a vertex of $G$. The **indegree of $v$**, $\boldsymbol{indeg_G(v)}$, is the number of edges of $G$ entering $v$ (i.e., $indeg_G(v) = |E \cap (V \times \{v\})|$). The **outdegree of $v$**, $\boldsymbol{outdeg_G(v)}$, is the number of edges of $G$ leaving $v$ (i.e., $outdeg_G(v) = |E \cap (\{v\} \times V)|$). For instance, $indeg_{G_2}(v_5) = 2$, $outdeg_{G_2}(v_5) = 1$, and $outdeg_{G_2}(v_2) = 3$. The **degree of $v$** of $G$ is

$$\boldsymbol{deg_G(v)} = indeg_G(v) + outdeg_G(v).$$

For instance, $deg_{G_2}(v_2) = 5$ and $deg_G(v_6) = 0$. A vertex $u$ with $deg_G(u) = 0$ is called an **isolated vertex** of $G$. The **degree of a directed graph** $G = (V, E)$ is

$$\boldsymbol{deg(G)} = \max\{deg_G(v) \mid v \in V\}.$$

**Definition 2.2.2.38.** *Let $G = (V, E)$ be a directed graph. A* **(directed) path** *in $G$ is a sequence of vertices $P = v_1, v_2, \ldots, v_m$, $v_i \in V$ for $i = 1, \ldots, m$, such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, m-1$. We say that $P$ is a path from $v_1$ to $v_m$. The* **length of $P$** *is $m-1$, i.e., the number of its edges.*

*A directed path $P = v_1, \ldots, v_m$ is called* **simple** *if either $|\{v_1, \ldots, v_m\}| = m$ or ($v_1 = v_m$, and $|\{v_1, \ldots, v_{m-1}\}| = m-1$). A path $P = v_1, \ldots, v_m$ is called a* **cycle** *if $v_1 = v_m$. A cycle $v_1, \ldots, v_m$ is* **simple** *if $|\{v_1, \ldots, v_{m-1}\}| = m-1$. A directed graph $G$ is called* **acyclic** *if it does not contain any cycle.*

$P = v_1, v_2, v_1, v_3$ is a directed path in the directed graph $G_2$ in Figure 2.8. $v_1, v_2, v_1$ is a simple cycle. A directed graph is **strongly connected** if for all vertices $u, v \in V$, $u \neq v$, there are directed paths from $u$ to $v$ and from $v$ to $u$ in $G$. The graph $G_2$ is not strongly connected.

Numerous real situations can be represented as graphs. Sometimes we need a more powerful representation formalism called weighted graphs.

**Definition 2.2.2.39.** *A* **weighted [directed] graph** *$G$ is a triple $(V, E,$ weight$)$, where*

*(i) $(V, E)$ is a [directed] graph, and*
*(ii) weight is a function from $E$ to $\mathbb{Q}^+$.*

*The adjacency matrix of a weighted graph $G = (V, E, weight)$ is $M_G = [a_{ij}]_{i,j=1,\ldots,|V|}$, where $a_{ij} = weight(\{v_i, v_j\})$ if $\{v_i, v_j\} \in E$ and $a_{ij} = 0$ if $\{v_i, v_j\} \notin E$.*

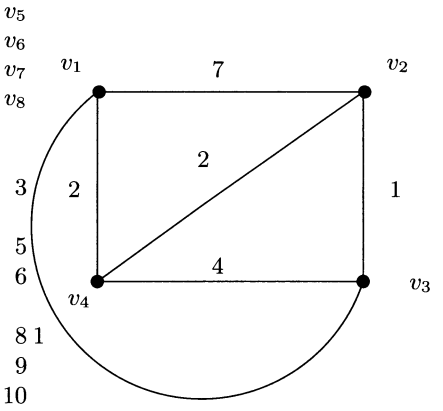



**Fig. 2.9.**

The weighted graph

$$G = (\{v_1, v_2, v_3, v_4\}, \{\{v_i, v_j\} \mid i, j \in \{1, \ldots, 4\}, i \neq j\}, weight),$$

where $weight(\{v_1, v_2\}) = 7$, $weight(\{v_1, v_3\}) = 1$, $weight(\{v_1, v_4\}) = 2$, $weight(\{v_2, v_3\}) = 1$, $weight(\{v_2, v_4\}) = 2$, and $weight(\{v_3, v_4\}) = 4$, is the graph depicted in Figure 2.9. Its adjacency matrix is the following symmetric matrix:

$$\begin{pmatrix} 0 & 7 & 1 & 2 \\ 7 & 0 & 1 & 2 \\ 1 & 1 & 0 & 4 \\ 2 & 2 & 4 & 0 \end{pmatrix}.$$

Observe that $G$ contains a Hamiltonian tour $v_1$, $v_4$, $v_3$, $v_2$, $v_1$.

### Keywords introduced in Section 2.2.2

permutation, combination, binomial coefficients, polynomial, asymptotic growth of functions, arithmetic series, geometric series, harmonic numbers, telescoping series, graph, adjacency matrix of a graph, Hamiltonian tour, Eulerian tour, bipartite graph, cut, matching, tree, multigraph, directed graph, weighted graph

## 2.2.3 Boolean Functions and Formulae

The aim of this section is to give elementary fundamentals of Boolean logic and to present some basic representations of Boolean functions such as Boolean formulae and branching programs. For formulae we consider some special normal forms such as disjunctive normal form (DNF) and conjunctive normal form (CNF).

Boolean logic is a fundamental mathematical system built around the two values "**TRUE**" and "**FALSE**". The values TRUE and FALSE are called **Boolean values**. In what follows we use the value 1 for TRUE and the value 0 for FALSE.

One manipulates Boolean values with logical (Boolean) operations. The fundamental ones are negation, conjunction, disjunction, exclusive or, equivalence, and implication. **Negation** is an unary operation denoted by $\neg$ and defined by $\neg(0) = 1$ and $\neg(1) = 0$. Sometimes we use the notation $\overline{0}$ and $\overline{1}$, respectively, instead of $\neg(0)$ and $\neg(1)$, respectively. **Conjunction** is a binary operation denoted by the symbol $\wedge$, and it corresponds to the logical *AND*, i.e., the result is 1 if and only if both arguments are 1s. **Disjunction** is a binary operation designated with the symbol $\vee$, and it corresponds to the logical *OR*, i.e., the result is 1 if and only if at least one of the arguments is 1. Corresponding to their logical meaning, they are defined as follows:

$$\begin{array}{ll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 \end{array}$$

**Exclusive or** is a binary operation designated by the symbol $\oplus$, and **implication** is designated by the symbol $\Rightarrow$. **Equivalence** is designated by $\Leftrightarrow$. They are defined as follows:

$$\begin{array}{lll} 0 \oplus 0 = 0 & 0 \Rightarrow 0 = 1 & 0 \Leftrightarrow 0 = 1 \\ 0 \oplus 1 = 1 & 0 \Rightarrow 1 = 1 & 0 \Leftrightarrow 1 = 0 \\ 1 \oplus 0 = 1 & 1 \Rightarrow 0 = 0 & 1 \Leftrightarrow 0 = 0 \\ 1 \oplus 1 = 0 & 1 \Rightarrow 1 = 1 & 1 \Leftrightarrow 1 = 1 \end{array}$$

So, the result of the exclusive or is 1 if either but not both of its operands are 1. The result of the implication is 0 if and only if the first argument is 1 and the second argument is 0, because the logical TRUE must not imply the logical FALSE.

**Exercise 2.2.3.1.** Determine which of the above Boolean operations are associative and commutative. □

**Exercise 2.2.3.2.** Prove that for any Boolean value $\alpha$ and $\beta$,

(i) $\alpha \vee \beta = \neg(\neg(\alpha) \wedge \neg(\beta))$,
(ii) $\alpha \wedge \beta = \neg(\neg(\alpha) \vee \neg(\beta))$,
(iii) $\alpha \oplus \beta = \neg(\alpha \Leftrightarrow \beta)$,
(iv) $\alpha \Rightarrow \beta = \neg(\alpha) \vee \beta$, and
(v) $\alpha \Leftrightarrow \beta = (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

□

**Definition 2.2.3.3.** *A* **Boolean variable** *is any symbol to which one can associate either of the values* 0 *or* 1.

*Let* $X = \{x_1, \ldots, x_n\}$ *be a set of Boolean variables for some* $n \in \mathbb{N}$. *A* **Boolean function over** $X$ *is any mapping* $f$ *from* $\{0,1\}^n$ *to* $\{0,1\}$. *One denotes* $f$ *by* $f(x_1, x_2, \ldots, x_n)$ *if one wants to call attention to the names of its variables.*

Every argument $\alpha \in \{0,1\}^n$ of $f$ can be also viewed as a mapping $\alpha : X \to \{0,1\}$ that assigns a Boolean value to every variable $x \in X$. Because of this we call $\alpha$ an **input assignment** of $f$.

The simplest possibility to represent a Boolean function of $n$ variables is to list the values of the function for all $2^n$ possible arguments (input assignments). Figure 2.10 presents the representation of a Boolean function $f$ of three variables $x_1, x_2$, and $x_3$.

**Definition 2.2.3.4.** *Let* $f(x_1, \ldots, x_n)$ *be a Boolean function over a set of Boolean variables* $X = \{x_1, \ldots, x_n\}$. *For every argument* $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\}^n$ *(input assignment* $\alpha$ *with* $\alpha(x_i) = \alpha_i$ *for* $i = 1, 2, \ldots, n$*) such that* $f(\alpha) = f(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ *we say that* $\alpha$ **satisfies** $f$.

$$N^1(f) = \{\alpha \in \{0,1\}^n \mid f(\alpha) = 1\}$$

| $x_1$ | $x_2$ | $x_3$ | $f(x_1, x_2, x_3)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Fig. 2.10.**

*is the set of all input assignments satisfying* $f$.

$$N^0(f) = \{\beta \in \{0,1\}^n \mid f(\beta) = 0\}$$

*is the set of all input assignments that do not satisfy* $f$.

*We say that* $f$ *is* **satisfiable** *if there exists an input assignment satisfying* $f$ *(i.e.,* $|N^1(f)| \geq 1$*).*

The function $f(x_1, x_2, x_3)$ in Figure 2.10 is satisfiable. The input assignment $\alpha : \{x_1, x_2, x_3\} \to \{0,1\}$ with $\alpha(x_1) = \alpha(x_2) = \alpha(x_3) = 0$ (the argument $(0,0,0)$ of $f$) satisfies $f(x_1, x_2, x_3)$.

$$N^1(f(x_1, x_2, x_3)) = \{(0,0,0), (0,1,1), (1,0,0), (1,1,0)\}, \text{ and}$$
$$N^0(f(x_1, x_2, x_3)) = \{(0,0,1), (0,1,0), (1,0,1), (1,1,1)\}.$$

Thus, we see that another possibility to represent a Boolean function $f$ is to give $N^1(f)$ or $N^0(f)$.

**Definition 2.2.3.5.** *Let* $X = \{x_1, \ldots, x_n\}$ *be a set of* $n$ *Boolean variables,* $n \in \mathbb{N} - \{0\}$. *Let* $f(x_1, \ldots, x_n)$ *be a Boolean function over* $X$. *We say that* $\boldsymbol{f(x_1, \ldots, x_n)}$ **essentially depends on the variable** $\boldsymbol{x_i}$ *iff there exist* $n-1$ *Boolean values*

$$\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n \in \{0,1\}$$

*such that*

$$f(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 0, \alpha_{i+1}, \ldots, \alpha_n) \neq f(\alpha_1, \alpha_2, \ldots, \alpha_{i-1}, 1, \alpha_{i+1}, \ldots, \alpha_n).$$

*If* $f(x_1, \ldots, x_n)$ *does not essentially depend on a variable* $x_j$ *for some* $j \in \{1, \ldots, n\}$, *then we say the Boolean variable* $\boldsymbol{x_j}$ **is dummy for** $\boldsymbol{f}$.

For instance, the Boolean function $f(x_1, x_2, x_3)$ from Figure 2.10 essentially depends on $x_1$ because by fixing $x_2 = 1$ and $x_3 = 0$

$$f(0,1,0) = 0 \neq 1 = f(1,1,0).$$

**Exercise 2.2.3.6.** Determine whether the Boolean function $f(x_1, x_2, x_3)$ from Figure 2.10 depends essentially on $x_2$ and $x_3$. □

**Exercise 2.2.3.7.** Prove that every Boolean function $f$ with $|N^1(f)| = 1$ essentially depends on all its input variables. □

The most usual way to describe Boolean functions is the representation via Boolean formulae.

**Definition 2.2.3.8.** *Let $X$ be a countable set of Boolean variables and let $S$ be a set of unary and binary Boolean operations. The class of* **Boolean formulae over $X$ and $S$** *(***formulae** *for short) is defined recursively as follows:*

*(i) The Boolean values* 0 *and* 1 *are Boolean formulae.*
*(ii) For every Boolean variable $x \in X$, $x$ is a Boolean formula.*
*(iii) If $F$ is a Boolean formula and $\varphi$ is a unary Boolean operation from $S$, then $\varphi(F)$ is a Boolean formula.*
*(iv) If $F_1$ and $F_2$ are Boolean formulae, and $\triangle \in S$ is a binary operation, then $(F_1 \triangle F_2)$ is a Boolean formula.*
*(v) Only the expressions constructed by using (i), (ii), (iii), and (iv) are Boolean formulae over $X$ and $S$.*

An example of a Boolean formula over the set of Boolean variables $X = \{x_1, x_2, x_3, \ldots\}$ and $S = \{\vee, \wedge, \Leftrightarrow, \Rightarrow\}$ is

$$F = (((x_1 \vee x_2) \wedge x_7) \Rightarrow (x_2 \Leftrightarrow x_3)).$$

Since the operations $\vee, \wedge, \oplus$, and $\Leftrightarrow$ are commutative and associative, we may sometimes omit the parentheses. Thus, for instance, the following expressions $((x_1 \vee x_2) \vee (x_3 \vee x_4))$, $(((x_1 \vee x_2) \vee x_3) \vee x_4)$, and $x_1 \vee x_2 \vee x_3 \vee x_4$ represent the same Boolean formula. We shall also use $\bigvee_{i=1}^{n} x_i$ $[\bigwedge_{i=1}^{n} x_i, \bigoplus_{i=1}^{n} x_i]$ instead of $x_1 \vee x_2 \vee \cdots \vee x_n$ $[x_1 \wedge x_2 \wedge \cdots \wedge x_n, x_1 \oplus x_2 \oplus \cdots \oplus x_n]$.

**Definition 2.2.3.9.** *Let $S$ be a set of unary and binary Boolean operations. Let $X$ be a set of Boolean variables, and let $F$ be a formula over $X$ and $S$. Let $\alpha$ be an input assignment to $X$. The* **value of $F$ under the input assignment $\alpha$, $F(\alpha)$**, *is the Boolean value defined as follows:*

*(i)* $F(\alpha) = \begin{cases} 0 \text{ if } F = 0 \\ 1 \text{ if } F = 1 \end{cases}$
*(ii)* $F(\alpha) = \alpha(x)$ *if* $F = x$ *for an* $x \in X$,
*(iii)* $F(\alpha) = \varphi(F_1(\alpha))$ *if* $F = \varphi(F_1)$ *for some unary Boolean operation* $\varphi \in S$,
*(iv)* $F(\alpha) = F_1(\alpha) \triangle F_2(\alpha)$ *for some binary* $\triangle \in S$ *if* $F = (F_1 \triangle F_2)$ *for some formulae* $F_1$ *and* $F_2$.

*Let $f$ be a Boolean function over $X$. If, for each input assignment $\beta$ from $X$ to $\{0,1\}$, $f(\beta) = F(\beta)$, then we say* **$F$ represents $f$**. *Two formulae $F_1$ and $F_2$ are* **equivalent**, *$F_1 = F_2$, if they represent the same Boolean function (i.e., $F_1(\alpha) = F_2(\alpha)$ for every $\alpha$).*

Obviously, every formula represents exactly one Boolean function. But one Boolean function can be represented by infinitely many formulae. For instance, the formulae $x_1 \vee x_2$, $\neg(\neg(x_1) \wedge \neg(x_2))$, $(x_1 \vee x_2 \vee x_1 \vee x_2) \wedge 1$, and $\neg(x_1 \Rightarrow x_2) \vee \neg(x_2 \Rightarrow x_1) \vee \neg(x_2 \Rightarrow \neg(x_1))$ represent the same Boolean function.

In what follows, we shall use the following notation. Let $\alpha \in \{0,1\}$, and let $x$ be a Boolean variable.

$$x^{\alpha} = \begin{cases} \neg(x) & \text{if } \alpha = 0 \\ x & \text{if } \alpha = 1. \end{cases}$$

A **clause** over $\{x_1, \ldots, x_n\}$ is a formula $x_{i_1}^{\alpha_1} \vee x_{i_2}^{\alpha_2} \vee \cdots \vee x_{i_m}^{\alpha_m}$ for any $(\alpha_1, \ldots, \alpha_m) \in \{0,1\}^m$, and any $\{i_1, i_2, \ldots, i_m\} \subseteq \{1, 2, \ldots, n\}$. Note that $m$ may be different from $n$. For instance, $x_1 \vee x_3$, $\overline{x}_1 \vee x_2 \vee x_3 \vee \overline{x}_1 \vee \overline{x}_2$ are clauses over $\{x_1, x_2, x_3\}$.

**Exercise 2.2.3.10.** Prove the following equivalences between formulae:

(i) $x \vee 0 = x$, $x \wedge 1 = x$, $x \oplus 0 = x$, $x \oplus 1 = \overline{x}$,
(ii) $x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$,
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$,
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$,
(iii) $x \vee x = x \wedge x = x \vee (x \wedge y) = x \wedge (x \vee y) = x$,
(iv) $x \vee \overline{x} = x \oplus \overline{x} = 1$, and
(v) $x \wedge \overline{x} = x \oplus x = 0$.

□

**Definition 2.2.3.11.** *Let $n$ be a positive integer, and let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. For every $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\}^n$ we define the* **minterm over $X$ according to $\alpha$** *as the Boolean formula*

$$\boldsymbol{minterm_{\alpha}(x_1, \ldots, x_n)} = x_1^{\alpha_1} \wedge x_2^{\alpha_2} \wedge \cdots \wedge x_n^{\alpha_n},$$

*and the* **maxterm over $X$ according to $\alpha$** *as the clause*

$$\boldsymbol{maxterm_{\alpha}(x_1, \ldots, x_n)} = x_1^{\neg(\alpha_1)} \vee x_2^{\neg(\alpha_2)} \vee \cdots \vee x_n^{\neg(\alpha_n)}.$$

For instance, $minterm_{(0,1,0)}(x_1, x_2, x_3) = x_1^0 \wedge x_2^1 \wedge x_3^0 = \overline{x}_1 \wedge x_2 \wedge \overline{x}_3$ and $maxterm_{(0,1,0)}(x_1, x_2, x_3) = x_1^1 \vee x_2^0 \vee x_3^1 = x_1 \vee \overline{x}_2 \vee x_3$.

**Observation 2.2.3.12.** For each $\alpha \in \{0,1\}^n$, $minterm_{\alpha}(x_1, \ldots, x_n)$ takes the Boolean value 1 iff $x_i = \alpha_i$ for $i = 1, \ldots, n$ (i.e., $N^1(minterm_{\alpha}(x_1, \ldots, x_n)) = \{\alpha\}$), and $maxterm_{\alpha}(x_1, \ldots, x_n)$ takes the Boolean value 0 iff $x_i = \alpha_i$ for $i = 1, \ldots, n$ (i.e., $N^0(maxterm_{\alpha}(x_1, \ldots, x_n)) = \{\alpha\}$).

*Proof.* Obviously, for any $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\}^n$,

$$minterm_{\alpha}(\alpha_1, \ldots, \alpha_n) = \alpha_1^{\alpha_1} \wedge \alpha_2^{\alpha_2} \wedge \cdots \wedge \alpha_n^{\alpha_n} = 1$$

because $\beta^\beta = 1$ for every Boolean value $\beta$. Since $\beta^\omega = 0$ for all Boolean values $\beta \neq \omega$,

$$minterm_\alpha(\beta_1, \ldots, \beta_n) = \alpha_1^{\beta_1} \wedge \alpha_2^{\beta_2} \wedge \cdots \wedge \alpha_n^{\beta_n} = 0$$

for all vectors $(\alpha_1, \alpha_2, \ldots, \alpha_n) \neq (\beta_1, \beta_2, \ldots, \beta_n)$.

Similarly, for any $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n) \in \{0,1\}^n$,

$$maxterm_\gamma(\gamma_1, \gamma_2, \ldots, \gamma_n) = \gamma_1^{\overline{\gamma}_1} \vee \gamma_2^{\overline{\gamma}_2} \vee \cdots \vee \gamma_n^{\overline{\gamma}_n} = 0$$

because $1^0 = 0^1 = 0$ and so $\beta^{\overline{\beta}} = 0$ for each $\beta \in \{0,1\}$. Let $(\gamma_1, \ldots, \gamma_n)$, $(\beta_1, \ldots, \beta_n) \in \{0,1\}^n$ and $(\gamma_1, \ldots, \gamma_n) \neq (\beta_1, \ldots, \beta_n)$. Then there exists $i \in \{1, \ldots, n\}$ such that $\beta_i \neq \gamma_i$ (i.e., $\beta_i = \overline{\gamma}_i$). Thus,

$$maxterm_\gamma(\beta_1, \ldots, \beta_n) = \beta_1^{\overline{\gamma}_1} \vee \cdots \vee \beta_n^{\overline{\gamma}_n} = \beta_i^{\overline{\gamma}_i} = \beta_i^{\beta_i} = 1.$$

□

Using the notion of *minterm* and *maxterm* one can assign some unique formula in a special normal form to every Boolean function.

**Theorem 2.2.3.13.** *Let $f : \{0,1\}^n \to \{0,1\}$, $n \in \mathbb{N} - \{0\}$, be a Boolean function over $X = \{x_1, \ldots, x_n\}$. Then*

$$f(x_1, \ldots, x_n) = \bigvee_{\alpha \in N^1(f)} minterm_\alpha(x_1, \ldots, x_n) = \bigvee_{\alpha \in N^1(f)} (x_1^{\alpha_1} \wedge \cdots \wedge x_n^{\alpha_n}).$$

*Proof.* Let $\beta = (\beta_1, \ldots, \beta_n) \in \{0,1\}^n$ be a vector such that $f(\beta) = 1$, i.e., $\beta \in N^1(f)$. Then $minterm_\beta(\beta_1, \ldots, \beta_n) = 1$ and so

$$\bigvee_{\alpha \in N^1(f)} minterm_\alpha(x_1, \ldots, x_n) = 1.$$

If $\gamma = (\gamma_1, \ldots, \gamma_n) \in \{0,1\}^n$ is a vector such that $f(\gamma) = 0$, then $\gamma \notin N^1(f)$. Following Observation 2.2.3.12 we have $minterm_\alpha(\gamma) = 0$ for each $\alpha \in N^1(f)$, since $\alpha \neq \gamma$. So,

$$\bigvee_{\alpha \in N^1(f)} minterm_\alpha(\gamma_1, \ldots, \gamma_n) = 0.$$

□

The formula $\bigvee_{\alpha \in N^1(f)} (x_1^{\alpha_1} \wedge \cdots \wedge x_n^{\alpha_n})$ is called the **complete disjunctive normal form**, or **complete DNF**, of $f$. The complete disjunctive normal form is unique for every Boolean function $f$ because it is unambiguously determined by $N^1(f)$.

The complete DNF of the Boolean function $f(x_1, x_2, x_3)$ from Figure 2.10 is

$$(\overline{x}_1 \wedge \overline{x}_2 \wedge \overline{x}_3) \vee (\overline{x}_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \overline{x}_2 \wedge \overline{x}_3) \vee (x_1 \wedge x_2 \wedge \overline{x}_3),$$

because $N^1(f) = \{(0,0,0), (0,1,1), (1,0,0), (1,1,0)\}$.

The next assertion can be proved in an analogous way to how Theorem 2.2.3.13 was proved.

**Theorem 2.2.3.14.** *Let* $f : \{0,1\}^n \to \{0,1\}$, $n \in \mathbb{N} - \{0\}$ *be a Boolean function over* $X = \{x_1, \ldots, x_n\}$. *Then*

$$f(x_1, \ldots, x_n) = \bigwedge_{\alpha \in N^0(f)} maxterm_\alpha(x_1, \ldots, x_n) = \bigwedge_{\alpha \in N^0(f)} \left( x_1^{\overline{\alpha}_1} \vee \cdots \vee x_n^{\overline{\alpha}_n} \right).$$

**Exercise 2.2.3.15.** Prove Theorem 2.2.3.14. □

The formula $\bigwedge_{\alpha \in N^0(f)} \left( x_1^{\overline{\alpha}_1} \vee \cdots \vee x_n^{\overline{\alpha}_n} \right)$ is called the **complete conjunctive normal form**, or **complete** CNF, of $f$. The complete CNF of the Boolean function $f(x_1, x_2, x_3)$ from Figure 2.10 is

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

because $N^0(f) = \{(0,0,1), (0,1,0), (1,0,1), (1,1,1)\}$.

**Definition 2.2.3.16.** *Let* $X = \{x_1, x_2, x_3, \ldots\}$ *be a set of Boolean variables. Let* $\overline{X} = \{\overline{x} \mid x \in X\} = \{\overline{x}_1, \overline{x}_2, \overline{x}_3, \ldots\}$. *A* **literal** *(over* $X$*) is any element from* $X \cup \overline{X}$.

*Any Boolean formula (over* $X$*) consisting of a conjunction of clauses is called to be in* **conjunctive normal form**, CNF.

*For any clause* $F = x_{i_1}^{\alpha_1} \vee x_{i_2}^{\alpha_2} \vee \cdots \vee x_{i_n}^{\alpha_n}$, $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0,1\}^n$, *the* **size of** $F$ *is* $n$, *i.e., the number of its literals. For every positive integer* $k$, *a formula* $\Phi$ *is in* ***k*-conjunctive normal form**, ***k*CNF**, *if* $\Phi$ *is in* CNF and every clause of $\Phi$ has a size of at most $k$.

The formula

$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_3 \vee \overline{x}_5) \wedge x_2 \wedge (x_4 \vee x_5)$$

is in 3CNF. Every complete CNF of Boolean functions of $n$ variables is in $n$CNF.

We observe that Theorem 2.2.3.13 (as well as Theorem 2.2.3.14) implies that every Boolean function can be expressed as a formula over the set of Boolean operations $\{\vee, \wedge, \neg\}$.

**Exercise 2.2.3.17.** Prove that every Boolean function can be represented by a formula over

(i) $\{\neg, \vee\}$,
(ii) $\{\neg, \wedge\}$, and
(iii) $\{\wedge, \oplus\}$. □

**Exercise 2.2.3.18.** Find a binary Boolean operation (Boolean function of two variables) $\varphi$, such that every Boolean function can be represented by a formula over $\{\varphi\}$. □

Branching programs are currently the standard formalism for the computer representation of Boolean functions. This is because this kind of representation is often more concise than the representations by the value table or formulae, and that some special versions of branching programs can be conveniently handled in several application areas.

**Definition 2.2.3.19.** *Let $X = \{x_1, \ldots, x_n\}$, $n \in \mathbb{N} - \{0\}$, be a set of Boolean variables. A* **branching program** *(**BP**) over $X$ is a directed acyclic graph $G = (V, E)$ with the following labeling and properties:*

*(i) There is exactly one vertex with indegree 0 in $G$, and this vertex is called the* **source** *(or the* **start vertex***) of the branching program.*
*(ii) Every vertex of a nonzero outdegree in $G$ is labeled by a Boolean variable from $X$.*
*(iii) There are exactly two vertices with outdegree equal to 0. These vertices are called the* **sinks** *(or* **output vertices***) of the branching program. One of the sinks is labeled by 0 and the other one by 1.*
*(iv) Every vertex $v$ of a nonzero outdegree has the outdegree equal to two. One of the two edges leaving $v$ is labeled by 1 and the other one is labeled by 0.*

*For any input assignment $\alpha : X \to \{0,1\}$, a branching program $A$ over $X$ computes the Boolean value $A(\alpha)$ in the following way:*

*(i) $A$ starts the computation on $\alpha$ in its source.*
*(ii) If $A$ is in a vertex labeled by a Boolean variable $x \in X$, then $A$ moves via the edge labeled by $\alpha(x)$ to the next vertex.*
*(iii) If $A$ reaches a sink, then $A(\alpha)$ is the label of that sink.*

*Let $f(x_1, \ldots, x_n) : \{0,1\}^n \to \{0,1\}$ be a Boolean function. We say that a* BP *$A$* **represents** *(or* **computes***) $f$ if $A(\beta) = f(\beta)$ for every input assignment $\beta : \{x_1, \ldots, x_n\} \to \{0,1\}$.*

For instance, the branching program $A$ in Figure 2.11a follows the computation path

$$x_1 \xrightarrow{0} x_2 \xrightarrow{0} x_1 \xrightarrow{0} x_3 \xrightarrow{0} 1$$

for the input $(0,0,0)$ and so $A(0,0,0) = 1$. For the inputs $(1,1,0)$ and $(1,0,0)$ $A$ uses the same computation

$$x_1 \xrightarrow{1} x_3 \xrightarrow{0} 1.$$

One can easily observe, that $A$ computes the Boolean function $f(x_1, x_2, x_3)$ given in Figure 2.10.

Every branching program unambiguously determines the Boolean function that it represents. On the other hand, one can represent a Boolean function by different branching programs. The Boolean function $f(x_1, x_2, x_3)$ in Figure 2.10 is represented by both branching programs from Figure 2.11.

Since the general branching programs are not so easy to handle, one usually uses some restricted normal forms of them. We shall consider only one such

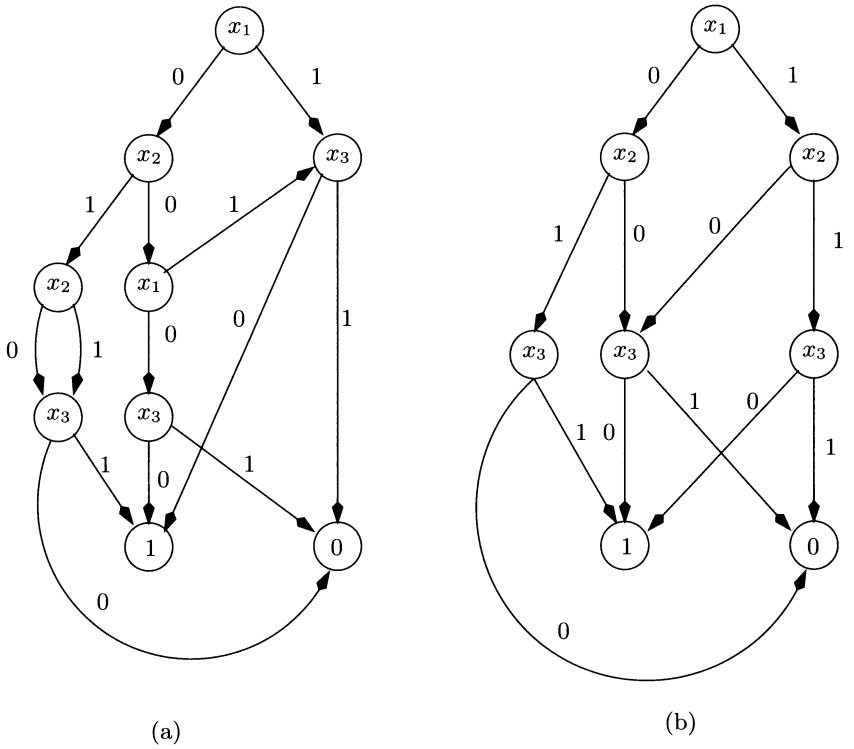



**Fig. 2.11.**

form where the restriction says that every Boolean variable may be asked for its value at most once in any computation of the branching program.

**Definition 2.2.3.20.** *Let $X$ be a set of Boolean variables, and let $A$ be a branching program over $X$. We say that $A$ is a* **one-time-only branching program** *if for every directed path $P$ of $A$ all vertices on $P$ have pairwise different labels.*

We observe that the branching program in Figure 2.11a is not a one-time-only branching program because it asks the Boolean variable $x_1$ twice on the path

$$x_1 \xrightarrow{0} x_2 \xrightarrow{0} x_1 \xrightarrow{0} x_3.$$

On the other hand, the branching program in Figure 2.11b is a one-time-only one.

**Exercise 2.2.3.21.** Construct

(a) a branching program with the minimal number of vertices for the Boolean function $f(x_1, x_2, x_3)$ from Figure 2.10.

(b) a minimal (according to the number of vertices) one-time-only branching program representing the function $f(x_1, x_2, x_3)$ from Figure 2.10.

□

**Exercise 2.2.3.22.** Construct a (one-time-only) branching program for the Boolean function

$$x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n.$$ □

### Keywords introduced in Section 2.2.3

Boolean values, Boolean operations, negation, conjunction, disjunction, implication, exclusive or, equivalence, Boolean variable, Boolean function, input assignment, satisfiability, Boolean formula, clause, minterm, maxterm, complete DNF, complete CNF, literal, conjunctive normal form (CNF), $k$CNF, branching program, one-time-only branching program

## 2.2.4 Algebra and Number Theory

The goal of this section is to recapitulate the definitions of some basic algebraic structures such as groups, semigroups, rings, and fields, and some fundamental results of number theory such as the Fundamental Theorem of Arithmetic, Prime Number Theorem, Fermat's Theorem, and Chinese Remainder Theorem. The proofs of all these results, except for Prime Number Theorem, can be expressed by elementary mathematics and so we present them. The only algorithmic part of this section is devoted to the presentation of Euclid's algorithm for the greatest common divisor. In fact, we are mainly interested in the algebraic structure $\mathbb{Z}_n$ with the set of elements $\{0, 1, 2, \ldots, n-1\}$ and the operations of multiplication and addition modulo $n$. An important observation is the fact that $\mathbb{Z}_n$ is a field if and only if $n$ is a prime. This and the fundamental theorems mentioned above are very useful for designing efficient randomized algorithms for algorithmic problems from number theory for which no polynomial-time deterministic algorithm is known. These are the most famous and transparent examples certificating the power and the usefulness of randomization in algorithmics. The knowledge of this section is crucial only for some parts of Chapter 5 on randomized algorithms, and so the reader may skip this section for now and look at it immediately before reading the related sections of Chapter 5.

A fundamental mathematical structure, also called algebraic structure or **algebra** for short, is a pair $(S, F)$, where

(i) $S$ is a set of elements, and
(ii) $F$ is a set of functions that map arguments from $S$ to an element of $S$. More precisely, $F$ is a set of **operations** on $S$, i.e., for every $f \in F$ there exists an integer $m$, such that $f$ is a function from $S^m$ into $S$. If $f : S^m \to S$, then we say that $f$ is an ***m*-ary operation on *S***.

We are not interested in structures with a mapping that for arguments from $S$ produces an element outside $S$. For our purposes we shall consider only the following fundamental sets of elements (numbers) and some fundamental binary operations corresponding to addition and multiplication:

| | |
|---|---|
| $\mathbb{N}$ | $= \{0, 1, 2, \ldots\}$ the set of all non-negative integers |
| $\mathbb{Z}$ | the set of all integers |
| $\mathbb{N}^+$ | the set of all positive integers |
| $\mathbb{N}^{\geq k}$ | $= \mathbb{N} - \{0, 1, \ldots, k-1\}$ for every positive integer $k$ |
| $\mathbb{Z}_n$ | $= \{0, 1, 2, \ldots, n-1\}$ |
| $\mathbb{Q}$ | the set of rational numbers |
| $\mathbb{Q}^+$ | the set of positive rational numbers |
| $\mathbb{R}$ | the set of real numbers |
| $\mathbb{R}^+$ | the set of positive real numbers |
| $\mathbb{R}^{\geq 0}$ | the set of non-negative real numbers |

If $F = \{f_1, \ldots, f_k\}$ is a finite set of operations of an algebra $(S, F)$, we simply write $(S, f_1, f_2, \ldots, f_k)$ instead of $(S, \{f_1, f_2, \ldots, f_k\})$ in what follows. For some binary operations $f$ and $g$ we use the notation $\cdot$ and $+$ if $f$ should be interpreted as a "version" of multiplication and if $g$ should be interpreted as a "version" of addition[7] in the algebra considered. Thus, instead of $f(x, y)$ $[g(x, y)]$ we simply write $x \cdot y$ $[x + y]$.

**Definition 2.2.4.1.** *A* **group** *is an algebra* $(S, *)$, *where*

*(i)* $*$ *is a binary operation,*
*(ii)* $*$ *is associative, i.e.,* $\forall x, y, z \in S : (x * y) * z = x * (y * z)$,
*(iii) there exists an* $e \in S$ *(called the* **neutral element according to** $*$ **in** $S$*), such that* $\forall x \in S$ :

$$e * x = x = x * e.$$

*(iv)* $\forall x \in S$ *there exists an element* $i(x) \in S$ *such that*

$$i(x) * x = e = x * i(x),$$

*where* $i(x)$ *is called the* **inverse element of** $x$ **according to** $*$.

*A group is said to be* **commutative** *if* $x * y = y * x$ *for all* $x, y \in S$.

In what follows, if $*$ is considered to be multiplication, then the neutral element is denoted by 1. If $*$ is considered to be addition, then the neutral element is denoted by 0.

**Example 2.2.4.2.** $(\mathbb{Z}, +)$ with $+$ as the standard addition of $(\mathbb{Z}, +)$ is a commutative group. The neutral element is 0, and $i(x) = -x$ for every $x \in \mathbb{Z}$. □

[7] Note that this interpretation does not necessarily mean that the considered operations $\cdot$ and $+$ are commutative.

**Exercise 2.2.4.3.** Prove that we cannot build a group on $\mathbb{N}$ by using the standard addition $+$ on $\mathbb{N}$. □

As already mentioned above we are mainly interested in $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$. To define a "natural" addition and multiplication on $\mathbb{Z}_n$ one needs the notion of "divisibility" and "remainder of the division". Let $a$, $d$ be non-negative integers, and let $k$ be a positive integer. If $a = k \cdot d$, then we say that $\boldsymbol{d}$ **divides** $\boldsymbol{a}$ or that $a$ is a **multiple** of $d$, and write $\boldsymbol{d|a}$. By agreement, every positive integer divides 0. Obviously, if $a$ and $d$ are positive integers and $d|a$, then $d \leq a$.

If $d|a$ we also say that $d$ is a **divisor of** $a$. For every positive integer $a$ the integers 1 and $a$ are so-called **trivial divisors of** $\boldsymbol{a}$. Nontrivial divisors of $a$ are called **factors of** $\boldsymbol{a}$. For instance, the factors of 24 are 2, 3, 4, 6, 8, 12.

The definition of the operations on $\mathbb{Z}_n$ is based on the elementary division of a positive integer $a$ by a positive integer $b$ for $b < a$. Using the school algorithm for division one can write

$$a = q \cdot b + r, \tag{2.9}$$

where $q$ is the result of the division procedure and $r < b$ is the remainder of the division. One can easily prove that for every integer $a$ and every positive integer $b$, there are unique integers $q$ and $r$ such that the equality (2.9) is true. In what follows we use the notation $\boldsymbol{a}$ **div** $\boldsymbol{b}$ for $q$ and $\boldsymbol{a}$ **mod** $\boldsymbol{b}$ for $r$. For instance, 21 div 5 = 4 and 21 mod 5 = 1. For every positive integer $n$, we define the operations $\oplus_n$ and $\odot_n$ on $\mathbb{Z}_n$ as follows:

For all $x, y \in \mathbb{Z}_n$ :

$$\boldsymbol{x} \oplus_{\boldsymbol{n}} \boldsymbol{y} = (x + y) \bmod n,$$
$$\boldsymbol{x} \odot_{\boldsymbol{n}} \boldsymbol{y} = (x \cdot y) \bmod n.$$

For instance, $7 \oplus_{13} 10 = (7 + 10) \bmod 13 = 17 \bmod 13 = 4$, $6 \odot_{11} 7 = (6 \cdot 7) \bmod 11 = 42 \bmod 11 = 9$.

Since the remainder of the division by $n$ is smaller than $n$, $x \oplus_n y$, $x \odot_n y \in \mathbb{Z}_n$ for all $x, y \in \mathbb{Z}_n$. So, $\odot_n$ and $\oplus_n$ are binary operations on $\mathbb{Z}_n$.

**Example 2.2.4.4.** Consider $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ and the operation $\oplus_n$ as defined above:

Then 0 can be considered as the neutral element according to $\oplus_n$. For every $x \in \mathbb{Z}_n$, define $i(x) = (n - x) \bmod n$. We observe that, for every $x$,

$$x \oplus_n i(x) = (x + (n - x) \bmod n) \bmod n = n \bmod n = 0.$$

Since $\oplus_n$ is associative and commutative, $(\mathbb{Z}_n, \oplus_n)$ with the neutral element 0 is a commutative group. □

The crucial point is that if one has a structure (algebra) like a group $(S, +)$, where there exists an inverse element $i(x)$ according to addition for

every $x \in S$, then one also has the **subtraction** in this structure.[8] One can define it as

$$\boldsymbol{x - y} = x + i(y).$$

In what follows we shall also use the notation $\boldsymbol{-y}$ for $i(y)$, where $i(y)$ is the inverse of $y$ according to addition.

A similar situation appears if one considers a group $(S, \cdot)$, where $\cdot$ stands for multiplication. If $i(x)$ is the inverse of $x$ according to $\cdot$, then one also has **division** in the group. Division is defined by

$$\boldsymbol{x/y} = x \cdot i(y),$$

and we often use the notation $\boldsymbol{y^{-1}}$ instead of $i(y)$.

If one considers the set $\mathbb{Z}$ and the standard multiplication operation one can easily observe that it is impossible to build a group. The neutral element according to multiplication is 1 and for no $x \in \mathbb{Z}$, $x \neq 1$, are we able to find an element $y$ such that $x \cdot y = 1$. Thus, we do not have inverse elements according to the multiplication in $\mathbb{Z}$, and we cannot define division in such a structure.

In what follows we also use the abbreviation $\boldsymbol{a^i}$ inductively defined for every $a \in S$ and every $i \in \mathbb{Z}$ in a group $(S, *)$ with the neutral element $e$ as follows:

(i) $a^0 = e$, $a^1 = a$, and $a^{-1} = i(a)$, where $a^0$ is called the **trivial power of** $\boldsymbol{a}$;
(ii) $a^{i+1} = a * a^i$ for every positive integer $i$, and $a^i$ is called a **nontrivial power of** $\boldsymbol{a}$ if $i \geq 1$;
(iii) $a^{-j} = (i(a))^j$ for every positive integer $j$.

An element $g \in S$ is called a **generator** of a group $(S, *)$ if $S = \{g^i \mid i \in \mathbb{Z}\}$. If a group has a generator, then the group is called **cyclic**. For instance, $(\mathbb{Z}_n, \oplus)$ is a cyclic group for every $n \in \mathbb{N}^+$ because $1^i = i$ for every $i \in \{1, 2, \ldots, n-1\}$, and $1^0 = 0$, i.e., 1 is a generator of $(\mathbb{Z}_n, \oplus_n)$. Observe that 2 is a generator of $(\mathbb{Z}_n, \oplus_n)$ if and only if $n$ is an odd integer.

**Definition 2.2.4.5.** *A* **semigroup** *is any algebra $(S, *)$, where $*$ is an associative, binary operation on $S$.*

*A* **monoid** *is an algebra $(M, *)$ where*

*(i) $*$ is a binary associative operation, and*
*(ii) there exists an $e \in S$ such that $\forall x \in S : e * x = x = x * e$.*

*A* **monoid** *is commutative if $\forall x, y \in M$, $x * y = y * x$.*

[8] Observe that subtraction considered as a function from $S^2$ to $S$ is not associative. This is the reason why one does not use the notion "operation" for it. The same holds for division.

**Example 2.2.4.6.** $(\mathbb{N}, +)$ with the neutral element 0, and $(\mathbb{Z}, \cdot)$ with the neutral element 1 are commutative monoids. $(\Sigma^*, \cdot)$, where $\Sigma$ is an alphabet (a finite set of symbols), $\Sigma^*$ is the set of all finite sequences over elements of $\Sigma$, and $\cdot$ is the concatenation of two sequences is a (noncommutative) monoid. The neutral element of $(\Sigma^*, \cdot)$ is the empty sequence $\lambda$. □

Now, we want to consider algebras that contain both addition and multiplication.

**Definition 2.2.4.7.** *A* **ring** *is an algebra* $(R, +, \cdot)$ *such that*

*(i)* $(R, +)$ *is a commutative group,*
*(ii)* $(R, \cdot)$ *is a semigroup, and*
*(iii) addition and multiplication are related by the* **distributive laws:**

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

*A* **ring** $(R, +, \cdot)$ *with neutral element* 0 *according to* $+$ *is called* **zero division free** *if for* $\forall x, y \in R - \{0\}$, $x \cdot y \neq 0$.

**Example 2.2.4.8.** Each of the sets $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ with standard addition and multiplication is a zero division free ring. □

Because of (i) of Definition 2.2.4.7 we see that subtraction is well defined in every ring. But in general, division is not available for rings. To obtain the possibility to divide in rings one has to add some additional requirements.

**Definition 2.2.4.9.** *A* **field** *is an algebra* $(R, +, \cdot)$ *satisfying the following conditions:*

*(i)* $(R, +, \cdot)$ *is a zero division free ring, with the neutral element* 0 *according to* $+$,
*(ii)* $a \cdot b = b \cdot a$ *for all* $a, b \in R$,
*(iii) there exists an element* $1 \in R$ *such that* $1 \cdot a = a \cdot 1 = a$ *for all* $a \in R - \{0\}$,
*(iv) for every* $b \in R - \{0\}$ *there exists* $i(b) \in R$ *such that* $b \cdot i(b) = 1$.

$\mathbb{Q}$ and $\mathbb{R}$, with respect to addition and multiplication, build fields, but for $\mathbb{Z}$ it is impossible to define division. In what follows we observe that $\mathbb{Z}_n$ can be used to build a field for some $n$s, but not for every $n$. The reason for this behavior will be explained later.

**Example 2.2.4.10.** Consider $\mathbb{Z}_n$ for every positive integer $n$, with the operations $\oplus_n$ and $\odot_n$. As we already observed, $(\mathbb{Z}_n, \oplus_n)$ is a commutative group. One can easily verify that the distributive laws for $\oplus_n$ and $\odot_n$ hold, too. Obviously, $(\mathbb{Z}_n, \odot_n)$ is a semigroup. Thus, the only properties in question are the zero divisor freeness and the existence of inverse elements according to $\odot_n$.

If $n = 12$, then $3 \odot_{12} 4 = 3 \cdot 4 \bmod 12 = 0$. Since $3 \neq 0$ and $4 \neq 0$ the ring $(\mathbb{Z}_{12}, \oplus_{12}, \odot_{12})$ is not zero division free. Obviously, this is true for every $n$ that can be written as $a \cdot b$, $a, b \in \{2, 3, \ldots, n-1\}$.

Now, let us look for the inverse elements of the elements of $\mathbb{Z}_{12}$ according to $\odot_{12}$. Since $1 \odot_{12} 1 = 1 \cdot 1 \bmod 12 = 1$, 1 can always be considered to be the inverse of 1, i.e., $1^{-1} = 1/1 = 1$ in $\mathbb{Z}_{12}$. Next, consider the element 2. Since $2 \odot_{12} a$ is even for every $a \in \mathbb{Z}_n$, $2 \cdot a \bmod 12$ is even too. Thus, for every $a \in \mathbb{Z}_{12}$, $2 \odot a \neq 1$. The conclusion is that there is no inverse element of 2 according to $\odot_{12}$.

Now we consider $\mathbb{Z}_5$. Realizing all 16 multiplications of nonzero elements, one can see that $(\mathbb{Z}_5, \oplus_5, \odot_5)$ is zero divisor free. In what follows we see the every nonzero element has its inverse according to $\odot_5$.

$1 \odot_5 1 = 1 \cdot 1 \bmod 5 = 1$, i.e., $1^{-1} = 1$

$2 \odot_5 3 = 2 \cdot 3 \bmod 5 = 1$, i.e., $2^{-1} = 3$ and $3^{-1} = 2$ because $2 \odot_5 3 = 3 \odot_5 2$

$4 \odot_5 4 = 4 \cdot 4 \bmod 5 = 16 \bmod 5 = 1$, i.e., $4^{-1} = 4$.

Thus, $(\mathbb{Z}_5, \oplus_5, \odot_5)$ is a field. □

In what follows, we usually omit the full description of algebras as tuples. For instance, we shall speak shortly about a field $\mathbb{Q}$, $\mathbb{Z}_5$, or $\mathbb{R}$ assuming that the reader automatically considers the standard operations connected with them.

Next, we shall deal with integers, and especially with primes.

**Definition 2.2.4.11.** *A* **prime** *is an integer $p > 1$, which has no factors (divisors*[9] *other than itself and one).*

*An integer $b > 1$ that is not a prime (i.e., $b = a \cdot c$ for some $a, c > 1$) is called* **composite**.

The smallest primes are the numbers $2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots$. The importance of the class of primes is that every positive integer greater that 1 can be expressed as a product of primes (if a number is not itself a prime, it may be successively factorized until all the factors are primes).

**Observation 2.2.4.12.** For every integer $a \in \mathbb{N}^{\geq 2}$, there exists an integer $k \geq 1$, primes $p_1, p_2, \ldots, p_k$, and $i_1, i_2, \ldots, i_k \in \mathbb{N}^{\geq 1}$ such that

$$a = p_1^{i_1} \cdot p_2^{i_2} \cdot \cdots \cdot p_k^{i_k}.$$

For instance, $720 = 5 \cdot 144 = 5 \cdot 2 \cdot 72 = 5 \cdot 2 \cdot 2 \cdot 36 = 5 \cdot 2 \cdot 2 \cdot 3 \cdot 12 = 5 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 4 = 5 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 2 \cdot 2 = 2^4 \cdot 3^2 \cdot 5$.

[9] Remember that an integer $a \neq 1$ is said to be a **factor** (or a **nontrivial divisor**) of an integer $b$ if there is some integer $c \neq 1$ such that $b = a \cdot c$ (i.e., $b \bmod a = 0$).

One of the first questions that arises concerning the class of primes is whether there are infinite many different primes or whether the cardinality of the class of all primes is an integer. The following assertion provides the answer to this question.

**Theorem 2.2.4.13.** *There are infinitely many different primes.*

*Proof.* We present the old Euclid's proof by the indirect method. Assume there are only finitely many different primes, say $p_1, p_2, \ldots, p_n$. Any other number is composite, and must be divisible by at least one of the primes $p_1, p_2, \ldots, p_n$. Consider the number

$$a = p_1 \cdot p_2 \cdot \cdots \cdot p_n + 1.$$

Since $a > p_i$ for every $i \in \{1, 2, \ldots, n\}$, $a$ must be composite, so, there exists an $i \in \{1, 2, \ldots, n\}$ such that $p_i$ is a factor of $a$. But $a \bmod p_j = 1$ for every $j \in \{1, 2, \ldots, n\}$, so, our initial assumption that there is only a finite number of primes leads to this contradiction, and hence its contrary must be true. □

Observe that the proof of Theorem 2.2.4.13 does not provide any method for constructing an arbitrary large sequence of primes. This is because $a = p_1 \cdot p_2 \cdot \cdots \cdot p_n + 1$ does not need to be a prime (for instance, $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30031 = 59 \cdot 509$ is a composite number), and if $a$ is a prime, then it does not need to be the $(n+1)$th prime ($2 \cdot 3 + 1 = 7$ but 7 is the 4th prime number, not the 3rd one). The only right conclusion of the proof of Theorem 2.2.4.13 is that there must exist a prime $p$ greater than $p_n$ and smaller than or equal to $a$. Therefore, one can search in the interval $(p_n, a]$ for the $(n+1)$th smallest prime.[10]

As already mentioned above the primes are important because each integer can be expressed as a product of primes. The Fundamental Theorem of Arithmetics claims that every integer $a > 1$ can be factorized into primes in only one way.

**Theorem 2.2.4.14 (The Fundamental Theorem of Arithmetics).** *For every integer $a \in \mathbb{N}^{\geq 2}$, $a$ can be expressed as a product of nontrivial powers of distinct primes*[11]

$$a = p_1^{i_1} \cdot p_2^{i_2} \cdot \cdots \cdot p_k^{i_k}$$

*and up to a rearrangement of the factors, this prime factorization is unique.*

*Proof.* The fact that $a$ can be expressed as a product of primes has been formulated in Observation 2.2.4.12. We have to prove that the decomposition of $a$ into a product of primes is unique. We again give an indirect method.

Let $m$ be the smallest integer from $\mathbb{N}^{\geq 2}$ such that

$$m = p_1 \cdot p_2 \cdot \cdots \cdot p_r = q_1 \cdot q_2 \cdot \cdots \cdot q_s, \tag{2.10}$$

[10] Note that up till now no simple (efficiently computable) formula yielding an infinite sequence of primes is known.

[11] $p_i \neq p_j$ for $i \neq j$.

where $p_1 \leq p_2 \leq \cdots \leq p_r$ and $q_1 \leq q_2 \leq \cdots \leq q_s$ are primes such that there exists $i \in \{1, \ldots, r\}$ such that $p_i \notin \{q_1, q_2, \ldots, q_s\}$.[12]

First, we observe that $p_1$ cannot be equal to $q_1$ (If $p_1 = q_1$, then $m/p_1 = p_2 \cdot \cdots \cdot p_r = q_2 \cdot \cdots \cdot q_s$ is an integer with two different prime factorizations. Since $m/p_1 < m$ this is the contradiction with the minimality of $m$).

Without loss of generality we assume $p_1 < q_1$. Consider the integer

$$m' = m - (p_1 q_2 q_3 \ldots q_s). \tag{2.11}$$

By substituting for $m$ the two expressions of (2.10) into (2.11) we may write the integer $m'$ in either of the two forms:

$$m' = (p_1 p_2 \ldots p_r) - (p_1 q_2 \ldots q_s) = p_1(p_2 p_3 \ldots p_r - q_2 q_3 \ldots q_s) \tag{2.12}$$

$$m' = (q_1 q_2 \ldots q_s) - (p_1 q_2 \ldots q_s) = (q_1 - p_1)(q_2 q_3 \ldots q_s). \tag{2.13}$$

Since $p_1 < q_1$, it follows from (2.13) that $m'$ is a positive integer, while (2.11) implies that $m' < m$. Hence, the prime decomposition of $m'$ must be unique.[13] Equality (2.12) implies that $p_1$ is a factor of $m'$. So, following (2.12) and the fact that (2.13) is a unique decomposition of $m'$, $p_1$ is a factor of either $(q_1 - p_1)$ or $(q_2 q_3 \ldots q_s)$. The latter is impossible because $q_s \geq q_{s-1} \geq \cdots \geq q_3 \geq q_2 \geq q_1 > p_1$ and $q_1, \ldots, q_s$ are primes. Hence $p_1$ must be a factor of $q_1 - p_1$, i.e., there exists an integer $a$ such that

$$q_1 - p_1 = p_1 \cdot a. \tag{2.14}$$

But (2.14) implies $q_1 = p_1(a+1)$ and so $p_1$ is a factor of $q_1$. This is impossible because $p_1 < q_1$ and $q_1$ is a prime. □

**Corollary 2.2.4.15.** *If a prime $p$ is a factor of an integer $a \cdot b$, then $p$ must be a factor of either $a$ or $b$.*

*Proof.* Assume that $p$ is a factor of neither $a$ nor $b$. Then the product of the prime decomposition of $a$ and $b$ does not contain $p$. Since $p$ is a factor of $a \cdot b$, there exists an integer $t$ such that $a \cdot b = p \cdot t$. The product of $p$ and the prime decomposition of $t$ yields a prime decomposition of $a \cdot b$ that contains $p$. Thus, we have two different prime decompositions of $a \cdot b$ which contradicts the Fundamental Theorem of Arithmetics. □

A large effort in number theory has been devoted to finding a simple mathematical formula yielding the sequence of all primes. Unfortunately, these attempts have not succeeded up till now and it is questionable whether such a formula exists. On the other hand, another important question:

*"How many primes are contained in $\{1, 2, \ldots, n\}$?"*

[12] So, the case $r = s = 1$ cannot happen because $m = p_1 = q_1$ contradicts $p_1 \notin \{q_1\}$.

[13] Aside from the order of the factors.

has been (at least) approximately answered. The next deep result is one of the most fundamental assertions of number theory and it has numerous applications (see Section 5.2 for some applications in algorithmics). The difficulty of the proof of this assertion is beyond the elementary level of this chapter and we omit it here.

Let ***Prim(n)*** denote the number of primes among integers $1, 2, 3, \ldots, n$.

**Theorem 2.2.4.16 (Prime Number Theorem).**

$$\lim_{n\to\infty} \frac{Prim(n)}{n/\ln n} = 1.$$

□

In other words the Prime Number Theorem says that the density $\frac{Prim(n)}{n}$ of the primes among the first $n$ integers tends to $\frac{1}{\ln n}$ as $n$ increases. The following table shows that $\frac{1}{\ln n}$ is a good "approximation" of $\frac{Prim(n)}{n}$ already for "small" $n$ (for which we are able to compute $Prim(n)$ exactly with a computer).

| $n$ | $\frac{Prim(n)}{n}$ | $\frac{1}{\ln n}$ | $\frac{Prim(n)}{n/\ln n}$ |
|---|---|---|---|
| $10^3$ | 0.168 | 0.145 | 1.159 |
| $10^6$ | 0.0885 | 0.0724 | 1.084 |
| $10^9$ | 0.0508 | 0.0483 | 1.053 |

Note that for $n \geq 100$ one can prove that

$$1 \leq \frac{Prim(n)}{n/\ln n} \leq 1.23,$$

and this is already very useful in the design of randomized algorithms.

In what follows we use the Gauss' congruence notation

$$\boldsymbol{a \equiv b \pmod{d}}$$

for the fact

$$a \bmod d = b \bmod d,$$

and say that ***a* and *b* are congruent modulo *d***. If $a$ is not congruent to $b$ modulo $d$, we shall write

$$\boldsymbol{a \not\equiv b \pmod{d}}.$$

**Exercise 2.2.4.17.** Prove that, for all positive integer $a \geq b$, the following statements are equivalent:

(i) $a \equiv b \pmod{d}$,
(ii) $a = b + nd$ for some integer $n$,
(iii) $d$ divides $a - b$. □

**Definition 2.2.4.18.** *For all integers $a$ and $b$, $a \neq 0$ or $b \neq 0$, the* **greatest common divisor of $a$ and $b$, $gcd(a,b)$**, *and the* **lowest common multiple of $a$ and $b$, $lcm(a,b)$**, *are defined as follows:*

$$\boldsymbol{gcd(a,b)} = \max\{d \mid d \text{ divides both } a \text{ and } b, \text{ i.e., } a \equiv b \equiv 0 (\text{mod } d)\}$$

*and*

$$\boldsymbol{lcm(a,b)} = \frac{a \cdot b}{gcd(a,b)} = \min\{c \mid a|c \text{ and } b|c\}.$$

*By convention, $gcd(0,0) = lcm(0,0) = 0$. We say that $a$ and $b$ are* **coprimes** *if $gcd(a,b) = 1$, i.e., if $a$ and $b$ have no common factor.*

For instance, the greatest common divisor of 24 and 30 is 6. It is clear that if one has the prime factorizations of two numbers $a = p_1 \cdot \cdots \cdot p_r \cdot q_1 \cdot \cdots \cdot q_s$ and $b = p_1 \cdot \cdots \cdot p_r \cdot h_1 \cdot \cdots \cdot h_m$, where $\{q_1, \ldots, q_s\} \cap \{h_1, \ldots, h_m\} = \emptyset$, then $gcd(a,b) = p_1 \cdot p_2 \cdot \cdots \cdot p_r$. For instance, $60 = 2 \cdot 2 \cdot 3 \cdot 5$, $24 = 2 \cdot 2 \cdot 2 \cdot 3$, and so $gcd(60,24) = 2 \cdot 2 \cdot 3 = 12$. Unfortunately, we do not know how to efficiently[14] compute the prime factorization of a given integer, and so this idea is not helpful if one wants to compute $gcd(a,b)$ for two given integers $a, b$. A very efficient computation of $gcd(a,b)$ is provided by the well-known Euclid's algorithm.[15] To explain it we present some simple assertions. First, we give an important property of common divisors.

**Lemma 2.2.4.19.** *Let $d > 0$, $a$, $b$ be some integers.*

*(i) If $d|a$ and $d|b$ then*

$$d|(ax + by)$$

*for any integers $x$ and $y$.*

*(ii) For any two positive integers $a$, $b$, $a|b$ and $b|a$ imply $a = b$.*

*Proof.* (i) If $d|a$ and $d|b$, then $a = n \cdot d$ and $b = m \cdot d$ for some integers $n$ and $m$. So,

$$ax + by = n \cdot d \cdot x + m \cdot d \cdot y = d(n \cdot x + m \cdot y).$$

(ii) $a|b$ implies $a \leq b$ and $b|a$ implies $b \leq a$. So, $a = b$. □

The following properties of *gcd* are obvious.

**Observation 2.2.4.20.** For all positive integers $a$, $b$, and $k$,

(i) $gcd(a,b) = gcd(b,a)$,
(ii) $gcd(a,0) = a$,
(iii) $gcd(a,ka) = a$ for every integer $k$,
(iv) if $k|a$ and $k|b$, then $k|gcd(a,b)$.

[14] There is no known polynomial-time algorithm for this task.
[15] Euclid's algorithm is one of the oldest algorithms (circa 300 B.C.).

**Exercise 2.2.4.21.** Prove that the *gcd* operator is associative, i.e., that for all integers $a$, $b$, and $c$,

$$gcd(a, gcd(b, c)) = gcd(gcd(a, b), c).$$

□

Euclid's algorithm is based on the following recursive property of *gcd*.

**Theorem 2.2.4.22.** *For any non-negative integer $a$ and any positive integer $b$,*

$$gcd(a, b) = gcd(b, a \bmod b).$$

*Proof.* We shall prove that $gcd(a, b)$ and $gcd(b, a \bmod b)$ divide each other, and so by the fact (ii) of Lemma 2.2.4.19 they must be equal.

(1) We first prove that $gcd(a, b)$ divides $gcd(b, a \bmod b)$. Following the definition of *gcd*, $gcd(a, b)|a$ and $gcd(a, b)|b$. We know (cf. (2.9)) that we can write $a$ as

$$a = (a \text{ div } b) \cdot b + a \bmod b$$

for some non-negative integer $a$ div $b$. Thus

$$a \bmod b = a - (a \text{ div } b) \cdot b.$$

This means that $a \bmod b$ is a linear combination of $a$ and $b$ and so every common divisor of $a$ and $b$ must divide $a \bmod b$ due to the fact (i) of Lemma 2.2.4.19. Thus,

$$gcd(a, b)|(a \bmod b).$$

Finally, due to the fact (iv) of Observation 2.2.4.20

$$gcd(a, b)|b \text{ and } gcd(a, b)|(a \bmod b)$$

imply

$$gcd(a, b)|gcd(b, a \bmod b).$$

(2) We shall prove that $gcd(b, a \bmod b)$ divides $gcd(a, b)$. Obviously,

$$gcd(b, a \bmod b)|b \text{ and } gcd(b, a \bmod b)|(a \bmod b).$$

Since

$$a = (a \text{ div } b) \cdot b + a \bmod b,$$

we see that $a$ is a linear combination of $b$ and $a \bmod b$.
Due to the assertion (i) of Lemma 2.2.4.19,

$$gcd(b, a \bmod b)|a.$$

Following the assertion (iv) of Observation 2.2.4.20,

$$gcd(b, a \bmod b)|b \text{ and } gcd(b, a \bmod b)|a$$

together imply

$$gcd(b, a \bmod b)|gcd(a, b).$$

□

Theorem 2.2.4.22 directly implies the correctness of the following recursive algorithm for *gcd*.

**Algorithm 2.2.4.23.** EUCLID'S ALGORITHM

*Euclid*$(a, b)$
Input: two positive integers $a$, $b$.
Recursive Step: **if** $b = 0$ **then return** $(a)$
**else return** *Euclid*$(b, a \bmod b)$.

**Example 2.2.4.24.** Consider the computation of EUCLID'S ALGORITHM for $a = 12750$ and $b = 136$.

$$\begin{aligned} Euclid(12750, 136) &= Euclid(136, 102) \\ &= Euclid(102, 34) \\ &= Euclid(34, 0) \\ &= 34. \end{aligned}$$

Thus, $gcd(12750, 136) = 34$. □

Obviously, EUCLID'S ALGORITHM cannot recurse indefinitely, since the second argument strictly decreases in each recursive call. Moreover, the above example shows that three recursive calls are enough to compute *gcd* of 12750 and 136. Obviously, the computational complexity of EUCLID'S ALGORITHM is proportional to the number of recursive calls it makes. The assertion of the following exercise shows that this algorithm is very efficient.

**Exercise 2.2.4.25.** Let $a \geq b \geq 0$ be two integers. Prove that the number of recursive calls of *Euclid*$(a, b)$ is in $O(\log_2 b)$.

There are many situations of both theoretical and practical interest when one asks whether a given integer $p$ is a prime or not. Following the definition of a prime one can decide this by looking at all numbers $2, \ldots, \lfloor \sqrt{p} \rfloor$ and testing whether one of them is a factor of $p$. If one wants to check that $p$ is a prime in this way, $\Omega(\sqrt{p})$ divisions have to be performed. But the best computer cannot perform $\sqrt{p}$ divisions if $p$ is a number consisting of hundreds of digits, and we need to work with such large numbers in practice. This is one of the reasons why people have searched for other characterizations[16] of primes.

[16] Equivalent definitions.

**Theorem 2.2.4.26.** *For every positive integer $p \geq 2$,*

$$p \text{ is a prime if and only if } \mathbb{Z}_p \text{ is a field.}$$

*Proof.* In Example 2.2.4.10 we have shown that if $p$ is a composite ($p = a \cdot b$ for $a > 1$, $b > 1$) then $\mathbb{Z}_p$ with $\oplus_n, \odot_n$ cannot build a field since $a \odot_n b = 0$.

We showed already in Example 2.2.4.4 that $(\mathbb{Z}_n, \oplus_n)$ is a commutative group for every positive integer $n$. So, one can easily see that $(\mathbb{Z}_n, \oplus_n, \odot_n)$ is a ring for all $n \in \mathbb{N}$. Since $\odot_n$ is commutative and $1 \odot_n a = a$ for every $a \in \mathbb{Z}_n$, it is sufficient to prove that the primality of $n$ implies the existence of an inverse element $a^{-1}$ for every $a \in \mathbb{Z}_n - \{0\}$.

Now, let $p$ be a prime. For every $a \in \mathbb{Z}_p - \{0\}$, consider the following $p-1$ multiples of $a$:

$$m_0 = 0 \cdot a, \quad m_1 = 1 \cdot a, \quad m_2 = 2 \cdot a, \quad \ldots \quad m_{p-1} = (p-1) \cdot a.$$

First, we prove that no two of these integers can be congruent modulo $p$. Let there exist two different $r, s \in \{0, 1, \ldots, p-1\}$, such that

$$m_r \equiv m_s (\text{mod } p).$$

Then $p$ is a factor of $m_r - m_s = (r-s) \cdot a$. But this cannot occur because $r - s < p$ (i.e., $p$ is not a factor of $r-s$) and $a < p$.[17] So, $m_0, m_1, m_2, \ldots, m_{p-1}$ are pairwise different in $\mathbb{Z}_p$.

Therefore, the numbers $m_1, m_2, \ldots, m_{p-1}$ must be respectively congruent to the numbers $1, 2, 3, \ldots, p-1$, in some arrangement. So,

$$\{0 \odot_p a, 1 \odot_p a, \ldots, a \odot_p (p-1)\} = \{0, 1, \ldots, p-1\}.$$

Now we are ready because it implies that there exists $b \in \mathbb{Z}_p$ such that $a \odot b = 1$, i.e., $b$ is the inverse element of $a$ in $\mathbb{Z}_p$. Since we have proved it for every $a \in \mathbb{Z}_p - \{0\}$, $\mathbb{Z}_p$ is a field. □

**Exercise 2.2.4.27.** Prove that if $p$ is a prime, then $(\{1, 2, \ldots, p-1\}, \odot_p)$ is a cyclic group. □

**Exercise 2.2.4.28.** (*) Define, for every positive integer $n$,

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n - \{0\} \mid gcd(a, n) = 1\}.$$

Prove that

(i) $(\mathbb{Z}_n^*, \odot_n)$ forms a group.
(ii) the group $(\mathbb{Z}_n^*, \odot_n)$ is cyclic if and only if either $n = 2, 4, p^k$ or $2p^k$, for some non-negative integer $k$ and an odd prime $p$. □

[17] Note that this argument works because of the Fundamental Theorem of Arithmetics.

This nice characterization of primes by Theorem 2.2.4.26 leads directly to the following important result of the number theory.

**Theorem 2.2.4.29 (Fermat's Theorem).** *For every prime $p$ and every integer $a$ such that $gcd(a,p) = 1$*

$$a^{p-1} \equiv 1 \pmod{p}.$$

*Proof.* Consider again[18] the numbers

$$m_1 = 1 \cdot a, \quad m_2 = 2 \cdot a, \quad \ldots \quad m_{p-1} = (p-1) \cdot a.$$

We claim by almost the same argument as in the proof above that no two of these integers can be congruent modulo $p$. Let there exist two different integers $r, s \in \{1, 2, \ldots, p-1\}$, $r > s$, such that

$$m_r \equiv m_s \pmod{p}.$$

Then $p$ is a factor of $m_r - m_s = (r-s) \cdot a$. This cannot occur because $r - s < p$ and $p$ is not a factor of $a$ according to our assumption $gcd(a,p) = 1$. Thus,

$$|\{m_1 \bmod p, m_2 \bmod p, \ldots, m_{p-1} \bmod p\}| = p - 1.$$

Now, we claim that none of the numbers $m_1, m_2, \ldots, m_{p-1}$ is congruent to 0 mod $p$. Since $\mathbb{Z}_p$ is a field, $m_r = r \cdot a \equiv 0 \pmod{p}$ forces either $a \equiv 0 \pmod{p}$ or $r \equiv 0 \pmod{p}$. But for every $r \in \{1, 2, \ldots, p-1\}$, $m_r = r \cdot a$, $r < p$, and $gcd(a,p) = 1$ (i.e., $p$ is a factor of neither $r$ nor $a$).

The conclusion is

$$\{m_1 \bmod p, m_2 \bmod p, \ldots, m_{p-1} \bmod p\} = \{1, 2, \ldots, p-1\}. \tag{2.15}$$

Finally, consider the following number

$$m_1 \cdot m_2 \cdot \cdots \cdot m_{p-1} = 1 \cdot a \cdot 2 \cdot a \cdot \cdots \cdot (p-1) \cdot a = 1 \cdot 2 \cdot \cdots \cdot (p-1) \cdot a^{p-1}. \tag{2.16}$$

Following (2.15) we get

$$1 \cdot 2 \cdot \cdots \cdot (p-1) \cdot a^{p-1} \equiv 1 \cdot 2 \cdot \cdots \cdot (p-1) \pmod{p},$$

i.e.,

$$1 \cdot 2 \cdot \cdots \cdot (p-1) \cdot \left(a^{p-1} - 1\right) \equiv 0 \pmod{p}.$$

Since $1 \cdot 2 \cdot \cdots \cdot (p-1) \not\equiv 0 \pmod{p}$ and $\mathbb{Z}_p$ is a field (i.e., $(\mathbb{Z}_p, \oplus_p, \odot_p)$ is zero division free), we obtain

$$a^{p-1} - 1 \equiv 0 \pmod{p}.$$

□

[18] As in the proof of Theorem 2.2.4.26

**Exercise 2.2.4.30.** Check Fermat's Theorem for $p = 5$ and $a = 9$. □

A nice consequence of Fermat's Theorem is a method for computing the inverse element according to multiplication.

**Corollary 2.2.4.31.** *Let $p$ be a prime. Then, for every $a \in \mathbb{Z}_p - \{0\}$,*

$$a^{-1} = a^{p-2} \bmod p.$$

*Proof.* $a \cdot a^{p-2} = a^{p-1} \equiv 1 \pmod{p}$ due to Fermat's Theorem. □

In what follows we frequently use the notation $-1$ (the inverse of 1 according to addition) for $p - 1$ in $\mathbb{Z}_p$. The following theorem provides a nice equivalent definition of primality.

**Theorem 2.2.4.32.** *Let $p > 2$ be an odd integer.*

$$p \text{ is a prime} \Leftrightarrow a^{(p-1)/2} \bmod p \in \{1, -1\} \text{ for all } a \in \mathbb{Z}_p - \{0\}.$$

*Proof.* (i) Let $p = 2p' + 1$, $p' \geq 1$, be a prime. Following Fermat's Theorem $a^{p-1} \equiv 1 \pmod{p}$ for every $a \in \mathbb{Z}_p - \{0\}$. Since

$$a^{p-1} = a^{2p'} = \left(a^{p'} - 1\right) \cdot \left(a^{p'} + 1\right) + 1,$$

we can write

$$\left(a^{p'} - 1\right) \cdot \left(a^{p'} + 1\right) \equiv 0 \pmod{p}. \tag{2.17}$$

Since $\mathbb{Z}_p$ is a field, (2.17) implies

$$\left(a^{p'} - 1\right) \equiv 0 \pmod{p} \text{ or } \left(a^{p'} + 1\right) \equiv 0 \pmod{p}. \tag{2.18}$$

Inserting $p' = (p-1)/2$ into (2.18) we finally obtain

$$a^{(p-1)/2} \equiv 1 \pmod{p} \quad \text{or} \quad a^{(p-1)/2} \equiv -1 \pmod{p}.$$

(ii) Let, for all $a \in \mathbb{Z}_p - \{0\}$, $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. It is sufficient to show that $\mathbb{Z}_p$ is a field. Obviously, $a^{p-1} = a^{(p-1)/2} \cdot a^{(p-1)/2}$.
If $a^{(p-1)/2} \equiv 1 \pmod{p}$, then $a^{p-1} \equiv 1 \pmod{p}$.
If $a^{(p-1)/2} \equiv -1 \equiv (p-1) \pmod{p}$, then

$$a^{p-1} \equiv (p-1)^2 \equiv p^2 - 2p + 1 \equiv 1 \pmod{p}.$$

So, for every $a \in \mathbb{Z}_p - \{0\}$, $a^{p-2} \bmod p$ is the inverse element $a^{-1}$ of $a$. To prove that $\mathbb{Z}_p$ is a field it remains to show that if $a \cdot b \equiv 0 \pmod{p}$ for some $a, b \in \mathbb{Z}_p$ then $a \equiv 0 \pmod{p}$ or $b \equiv 0 \pmod{p}$. Let $a \cdot b \equiv 0 \pmod{p}$, and let $b \not\equiv 0 \pmod{p}$. Then there exists $b^{-1} \in \mathbb{Z}_p$ such that $b \cdot b^{-1} \equiv 1 \pmod{p}$. Finally,

$$a = a \cdot (b \cdot b^{-1}) = (a \cdot b) \cdot b^{-1} \equiv 0 \cdot b^{-1} \equiv 0 \pmod{p}.$$

In this way we proved that $\mathbb{Z}_p$ is zero division free, and so $\mathbb{Z}_p$ is a field. □

In order to use an extension of Theorem 2.2.4.32 for primality testing in Chapter 5, we shall investigate the properties of $\mathbb{Z}_n$ when $n$ is composite. Let, for instance, $n = p \cdot q$ for two primes $p$ and $q$. We know that $\mathbb{Z}_p$ and $\mathbb{Z}_q$ are fields. Now, consider the direct product $\mathbb{Z}_p \times \mathbb{Z}_q$. The elements of $\mathbb{Z}_p \times \mathbb{Z}_q$ are pairs $(a_1, a_2)$, where $a_1 \in \mathbb{Z}_p$ and $a_2 \in \mathbb{Z}_q$. We define addition in $\mathbb{Z}_p \times \mathbb{Z}_q$ as

$$(a_1, a_2) \oplus_{p,q} (b_1, b_2) = ((a_1 + b_1) \bmod p, (a_2 + b_2) \bmod q),$$

and multiplication as

$$(a_1, a_2) \odot_{p,q} (b_1, b_2) = ((a_1 \cdot b_1) \bmod p, (a_2 \cdot b_2) \bmod q).$$

The idea is to show that $\mathbb{Z}_n$ and $\mathbb{Z}_p \times \mathbb{Z}_q$ are isomorphic, i.e., there exists a bijection $h : \mathbb{Z}_n \to \mathbb{Z}_p \times \mathbb{Z}_q$ such that

$$h(a \oplus_n b) = h(a) \oplus_{p,q} h(b) \text{ and } h(a \odot_n n) = h(a) \odot_{p,q} h(b)$$

for all $a, b \in \mathbb{Z}_n$. If one finds such a $h$, then one can view $\mathbb{Z}_n$ as $\mathbb{Z}_p \times \mathbb{Z}_q$. We define $h$ simply as follows:

$$\text{For all } a \in \mathbb{Z}_n,\ h(a) = (a \bmod p, a \bmod q).$$

One can simply verify that $h$ is injective.[19] For each $a, b \in \mathbb{Z}_n$, $a$ and $b$ can be written as

$$\begin{aligned} a &= a_1' \cdot p + a_1 = a_2' \cdot q + a_2 \quad \text{for some } a_1 < p \text{ and } a_2 < q, \\ b &= b_1' \cdot p + b_1 = b_2' \cdot q + b_2 \quad \text{for some } b_1 < p \text{ and } b_2 < q, \end{aligned}$$

i.e., $h(a) = (a_1, a_2)$ and $h(b) = (b_1, b_2)$. So,

$$\begin{aligned} h(a \oplus_n b) &= h(a + b \bmod n) = ((a + b) \bmod p, (a + b) \bmod q) \\ &= ((a_1 + b_1) \bmod p, (a_2 + b_2) \bmod q) \\ &= (a_1, a_2) \oplus_{p,q} (b_1, b_2) = h(a) \oplus_{p,q} h(b). \end{aligned}$$

Similarly,

$$\begin{aligned} h(a \odot_n b) &= h(a \cdot b \bmod n) = ((a \cdot b) \bmod p, (a \cdot b) \bmod q) \\ &= ((a_1' \cdot b_1' \cdot p^2 + (a_1 \cdot b_1' + a_1' \cdot b_1) \cdot p + a_1 \cdot b_1) \bmod p, \\ &\quad (a_2' \cdot b_2' \cdot q^2 + (a_2' \cdot b_2 + a_2 \cdot b_2') \cdot q + a_2 \cdot b_2) \bmod q) \\ &= ((a_1 \cdot b_1) \bmod p, (a_2 \cdot b_2) \bmod q) \\ &= (a_1, a_2) \odot_{p,q} (b_1, b_2) = h(a) \odot_{p,q} h(b). \end{aligned}$$

In general, $n = p_1 \cdot p_2 \cdot \cdots \cdot p_k$ for primes $p_1, p_2, \ldots, p_k$, and one considers the isomorphism between $\mathbb{Z}_n$ and $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_n}$. This isomorphism is called Chinese remainder and we formulate it in the next theorem.

[19] In fact, we do not need to require that $p$ and $q$ are primes; it is sufficient to assume that $p$ and $q$ are coprimes.

**Theorem 2.2.4.33 (Chinese Remainder Theorem, first version).** *Let* $m = m_1 \cdot m_2 \cdot \cdots \cdot m_k$, $k \in \mathbb{N}^+$, *where* $m_i \in \mathbb{N}^{\geq 2}$ *are pairwise coprimes (i.e.,* $gcd(m_i, m_j) = 1$ *for* $i \neq j$*). Then, for any sequence of integers* $r_1 \in \mathbb{Z}_{m_1}$, $r_2 \in \mathbb{Z}_{m_2}, \cdots, r_k \in \mathbb{Z}_{m_k}$, *there is an integer* $r$ *such that*

$$r \equiv r_i \pmod{m_i}$$

*for every* $i \in \{1, \ldots, k\}$, *and this integer* $r$ *is unique in* $\mathbb{Z}_m$.

*Proof.* We first show that there exists at least one such $r$. Since $gcd(m_i, m_j) = 1$ for $i \neq j$, $gcd(\frac{m}{m_l}, m_l) = 1$ for every $l \in \{1, 2, \ldots, k\}$. It follows that there exists a multiplicative inverse $n_i$ for $m/m_i$ in the group $\mathbb{Z}_{m_i}$. Consider, for $i = 1, \ldots, k$,

$$e_i = n_i \frac{m}{m_i} = m_1 \cdot m_2 \cdot \cdots \cdot m_{i-1} \cdot n_i \cdot m_{i+1} \cdot m_{i+2} \cdot \cdots \cdot m_k.$$

For every $j \in \{1, \ldots, k\} - \{i\}$, $\frac{m}{m_i} \equiv 0 \pmod{m_j}$, and so

$$e_i \bmod m_j = 0.$$

Since $n_i$ is the multiplicative inverse for $m/m_i$ in $\mathbb{Z}_{m_i}$,

$$e_i \bmod m_i = n_i \frac{m}{m_i} \bmod m_i = 1.$$

Now, we set

$$r \equiv \left( \sum_{i=1}^{k} r_i \cdot e_i \right) \pmod{m}$$

and we see that $r$ has the required properties.

To see that $r$ is uniquely determined modulo $m$, let us assume that there exist two integers $x$ and $y$ satisfying

$$y \equiv x \equiv r_i \pmod{m_i}$$

for every $i \in \{1, \ldots, k\}$. Then,

$$x - y \equiv 0 \pmod{m_i}$$

for every $i \in \{1, \ldots, k\}$, since $m = m_1 \cdot m_2 \cdot \cdots \cdot m_k$ and $gcd(m_i, m_j) = 1$ for $i \neq j$, $x \equiv y \pmod{m}$. □

The first version of the Chinese Remainder Theorem can be viewed as a statement about solutions of certain equations. The second version can be regarded as a theorem about the structure of $\mathbb{Z}_n$. Because the proof idea has been already explained for the isomorphism between $\mathbb{Z}_{p \cdot q}$ and $\mathbb{Z}_p \times \mathbb{Z}_q$ above, we leave the proof of the second version of the Chinese Remainder Theorem as an exercise to the reader.

**Theorem 2.2.4.34 (Chinese Remainder Theorem, second version).** *Let $m = m_1 \cdot m_2 \cdot \cdots \cdot m_k$, $k \in \mathbb{N}^+$, where $m_i \in \mathbb{N}^{\geq 2}$ are pairwise coprimes (i.e., $gcd(m_i, m_j) = 1$ for $i \neq j$). Then $\mathbb{Z}_m$ is isomorphic to $\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_k}$.*

**Exercise 2.2.4.35.** Prove the second version of the Chinese Remainder Theorem. □

The above stated theorems are fundamental results from number theory and we will use them in Chapter 5 to design efficient randomized algorithms for primality testing. For the development of (randomized) algorithms for algorithmic problems arising from number theory, one often needs fundamental results on (finite) groups, especially on $\mathbb{Z}_n^*$. Hence we present some of the basic assertions in what follows.

Since $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid gcd(a, n) = 1\}$, we first look for some basic properties of greatest common divisors. We start with an equivalent definition of the greatest common divisor of two numbers.

**Theorem 2.2.4.36.** *Let $a, b \in \mathbb{N} - \{0\}$, and let*

$$Com(a, b) = \{ax + by \mid x, y \in \mathbb{Z}\}$$

*be the set of linear combinations of $a$ and $b$. Then*

$$gcd(a, b) = \min\{d \in Com(a, b) \mid d \geq 1\},$$

*i.e., $gcd(a, b)$ is the smallest positive integer from $Com(a, b)$.*

*Proof.* Let $h = \min\{d \in Com(a, b) \mid d \geq 1\}$, and let $h = ax + by$ for some $x, y \in \mathbb{Z}$. We prove $h = gcd(a, b)$ by proving the inequalities $h \leq gcd(a, b)$ and $h \geq gcd(a, b)$ separately.

First we show that $h$ divides both $a$ and $b$ and so it divides $gcd(a, b)$. Following the definition of modulo $n$, we have

$$a \bmod h = a - \lfloor a/h \rfloor \cdot h = a - \lfloor a/h \rfloor \cdot (ax + by) = a \cdot (1 - \lfloor a/h \rfloor x) + b \cdot (-\lfloor a/h \rfloor y)$$

and so $a \bmod h$ is a linear combination of $a$ and $b$. Since $h$ is the smallest positive linear combination of $a$ and $b$, and $a \bmod h < h$, we obtain

$$a \bmod h = 0, \text{ i.e., } h \text{ divides } a.$$

The same argumentation provides the fact that $h$ divides $b$. Thus

$$h \leq gcd(a, b).$$

Since $gcd(a, b)$ divides both $a$ and $b$, $gcd(a, b)$ must divide $au + bv$ for all $u, v \in \mathbb{Z}$, i.e., $gcd(a, b)$ divides every element in $Com(a, b)$. Since $h \in Com(a, b)$,

$$gcd(a, b) \text{ divides } h, \text{ i.e., } gcd(a, b) \leq h.$$

□

**Theorem 2.2.4.37.** *Let $a, n \in \mathbb{N} - \{0\}$, $n \geq 2$, and let $gcd(a, n) = 1$. Then the congruence*

$$ax \equiv 1 \pmod{n}$$

*has a solution $x \in \mathbb{Z}_n$.*

*Proof.* Following Theorem 2.2.4.36, there exists $u, v \in \mathbb{Z}$ such that

$$a \cdot u + n \cdot v = 1 = gcd(a, n).$$

Observe, that for every $k \in \mathbb{Z}$

$$a \cdot u + n \cdot v = a \cdot (u + kn) + n \cdot (v - ka),$$

and so

$$a \cdot (u + kn) + n \cdot (v - ka) = 1.$$

For sure there is a $l \in \mathbb{Z}$ such that $a + ln \in \mathbb{Z}_n$. Set $x = u + ln$. Since

$$a(u + ln) + n(v - la) \equiv a(u + ln) \equiv \pmod{n},$$

$x$ is a solution of the congruence $ax \equiv 1 \pmod{n}$. □

**Exercise 2.2.4.38.** Let $a, n$ satisfy the assumptions of Theorem 2.2.4.37. Prove that the solution $x$ of the congruence $ax \equiv 1 \pmod{n}$ is unique in $\mathbb{Z}_n$.

**Exercise 2.2.4.39.** Let $a, n \in \mathbb{N} - \{0\}$, $n \geq 2$, and let $gcd(a, n) = 1$. Prove, that for every $b \in \mathbb{Z}_n$, the congruence $ax \equiv b \pmod{n}$ has a unique solution $x \in \mathbb{Z}_n$.

Now, we are ready to prove one of the most useful facts for developing randomized algorithms for primality testing. Later we extend the following assertion for any positive integer $n$.

**Theorem 2.2.4.40.** *For every prime $n$, $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is a commutative group.*

*Proof.* Let $a, b$ be arbitrary elements of $\mathbb{Z}_n^*$. Following the definition of $\mathbb{Z}_n^*$, we have $gcd(a, n) = gcd(b, n) = 1$. Since

$$Com(ab, n) \subseteq Com(a, n) \cap Com(b, n),$$

Theorem 2.2.4.36 implies that the number $1 \in Com(ab, n)$, and so $gcd(ab, n) = 1$. Thus, $ab \in \mathbb{Z}_n^*$, i.e., $\mathbb{Z}_n^*$ is closed under multiplication modulo $n$. This implies that $(\mathbb{Z}_n^*, \odot_{\bmod n})$ is an algebra.

Obviously, 1 is the identity element with respect to $\odot_{\bmod n}$ and $\odot_{\bmod n}$ is an associative and commutative operation.

Theorem 2.2.4.37 ensures the existence of an inverse element $x = a^{-1}$ for every $a \in \mathbb{Z}_n^*$ (i.e., for every $a \in \mathbb{Z}_n$ with $gcd(a, n) = 1$). Following Definition 2.2.4.1, the proof is completed. □

**Exercise 2.2.4.41.** Let $(A, *)$ be a group. Prove the following facts.

(i) For every $a \in A$, $a = (a^{-1})^{-1}$.
(ii) For all $a, b, c \in A$,
$a * b = c * b$ implies $a = c$, and
$b * a = b * c$ implies $a = c$.
(iii) For all $a, b, c \in A$,

$$a \neq b \Leftrightarrow a * c \neq b * c \Leftrightarrow c * a \neq c * b.$$

□

**Definition 2.2.4.42.** *Let $(A, \circ)$ be a group with the identity element 1. For every $a \in A$, the* **order of $a$, $order(a)$** *is the smallest $r \in \mathbb{N} - \{0\}$, such that*

$$a^r = 1,$$

*if such an $r$ exists. If $a^i \neq 1$ for all $i \in \mathbb{N} - \{0\}$, then we set $order(a) = \infty$.*

We show that Definition 2.2.4.42 is consistent, i.e., that every element of a finite group $(A, \circ)$ has a finite order. Consider the $|A| + 1$ elements

$$a^0, a^1, a^2, \ldots, a^{|A|}$$

from $A$. There must exist $0 \leq i \leq j \leq |A|$ such that

$$a^i = a^j.$$

This implies

$$1 = a^i \cdot (a^{-1})^i = a^j \cdot (a^{-1})^i = a^{(j-i)}$$

and so $order(a) \leq j - i$, i.e., $order(a) \in \{1, 2, \ldots, |A|\}$.

**Definition 2.2.4.43.** *Let $(A, \circ)$ be a group. An algebra $(H, \circ)$ is called a* **subgroup of $A$** *if $(H, \circ)$ is a group and $H \subseteq A$.*

For instance $(\mathbb{Z}, +)$ is a subgroup of $(\mathbb{Q}, +)$, and $(\{1\}, \odot_{\text{mod } 5})$ is a subgroup of $(\mathbb{Z}_5^*, \odot_{\text{mod } 5})$. But $(\mathbb{Z}_5^*, \odot_{\text{mod } 5})$ is not a subgroup of $(\mathbb{Z}_7^*, \odot_{\text{mod } 7})$ because $\odot_{\text{mod } 5}$ and $\odot_{\text{mod } 7}$ are different operations ($4 \odot_{\text{mod } 5} 4 = 1$ and $4 \odot_{\text{mod } 7} 4 = 2$).

**Lemma 2.2.4.44.** *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. Then the identity elements of both groups are the same.*

*Proof.* Let $e_H$ be the identity of $(H, \circ)$, and let $e_A$ be the identity of $(A, \circ)$. Since $e_H$ is the identity of $(H, \circ)$,

$$e_H \circ e_H = e_H. \tag{2.19}$$

Since $e_A$ is the identity of $(A, \circ)$ and $e_H \in A$,

$$e_A \circ e_H = e_H. \tag{2.20}$$

Thus, the left sides of the equalities (2.19) and (2.20) are the same, i.e.,

$$e_H \circ e_H = e_A \circ e_H. \tag{2.21}$$

If $e_H^{-1}$ is the inverse of $e_H$ in $(A, \circ)$, then multiplying (2.21) by $e_H^{-1}$ we obtain

$$e_H = e_H \circ e_H \circ e_H^{-1} = e_A \circ e_H \circ e_H^{-1} = e_A.$$

□

**Theorem 2.2.4.45.** *Let $(A, \circ)$ be a finite group. Every algebra $(H, \circ)$ with $H \subseteq A$ is a subgroup of $(A, \circ)$.*

*Proof.* Let $H \subseteq A$, and let $(H, \circ)$ be an algebra. To prove that $(H, \circ)$ is a subgroup of $(A, \circ)$, it is sufficient to show that $(H, \circ)$ is a group, i.e., that $e_A$ is the identity of $(H, \circ)$ and that every $b \in H$ has its inverse $b^{-1}$ in $H$.

Let $b$ be an arbitrary element of $H$. Since $b \in A$ and $A$ is finite, $order(b) \in \mathbb{N} - \{0\}$. Thus

$$b^{order(b)} = e_A.$$

Since $b^i \in H$ for all positive integers $i$ (remember that $H$ is closed under $\circ$), $e_A \in H$. Since

$$e_A \circ d = d$$

for every $d \in A$ and $H \subseteq A$, $e_A$ is the identity of $(H, \circ)$, too.

Since $b^{order(b)-1} \in H$ for any $b \in H$ and

$$e_A = b^{order(b)} = b \circ b^{order(b)-1},$$

$b^{order(b)-1}$ is the inverse element of $b$ in $(H, \circ)$. □

Theorem 2.2.4.45 is a useful instrument when working with groups, because in order to prove that $(H, \circ)$ is a subgroup of a finite group $(A, \circ)$, it is sufficient to show that $H \subseteq A$ and that $H$ is closed under $\circ$.

Note that the assumption of Theorem 2.2.4.45 that $A$ is finite is essential, because $(\mathbb{N}, +)$ is an algebra, but $(\mathbb{N}, +)$ is not a subgroup of the group $(\mathbb{Z}, +)$.

**Exercise 2.2.4.46.** Let $(H, \circ)$ and $(G, \circ)$ be two subgroups of a group $(A, \circ)$. Prove that $(H \cap G, \circ)$ is a subgroup of $(A, \circ)$. □

**Lemma 2.2.4.47.** *Let $(A, \circ)$ be a group with the identity $e$ and let $a \in A$ be an element with a finite order. Then, for $H(a) = \{e, a, a^2, \ldots, a^{order(a)-1}\}$, $(H(a), \circ)$ is the smallest subgroup of $(A, \circ)$ that contains $a$.*

*Proof.* First, we prove that $H(a)$ is closed under $\circ$. Let $a^i$ and $a^j$ be two arbitrary elements of $H(a)$. If $i + j < order(a)$, then

$$a^i \circ a^j = a^{i+j} \in H(a).$$

If $i + j > order(a)$, then

$$a^i \circ a^j = a^{i+j} = a^{order(a)} \circ a^{i+j-order(a)}$$
$$= e \circ a^{i+j-order(a)} = a^{i+j-order(a)} \in H(a).$$

Following the definition of $H(a)$, $e \in H(a)$. For every element $a^i \in H(a)$,

$$e = a^{order(a)} = a^i \circ a^{order(a)-i},$$

i.e., $a^{order(a)-i}$ is the inverse element to $a^i$.

Since every algebra $(G, \circ)$ with $a \in G$ must contain $H(a)$, $(H(a), \circ)$ is the smallest subgroup of $(A, \circ)$ that contains $a$. □

**Definition 2.2.4.48.** *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. For every $b \in A$, the set*

$$\boldsymbol{H \circ b} = \{h \circ b \mid h \in H\}$$

*is called a* **right coset** *of $H$ in $(A, \circ)$, and the set*

$$\boldsymbol{b \circ H} = \{b \circ h \mid h \in H\}$$

*is called a* **left coset** *of $H$ in $(A, \circ)$. If $H \circ b = b \circ H$, then $H \circ b$ is called a* **coset** *of $H$ in $(A, \circ)$.*

For instance, $(\{7 \cdot a \mid a \in \mathbb{Z}\}, +)$ is a subgroup of $(\mathbb{Z}, +)$. Let $B_7 = \{7 \cdot a \mid a \in \mathbb{Z}\}$. Then

$$B_7 + i = i + B_7 = \{7 \cdot a + i \mid a \in \mathbb{Z}\} = \{b \in \mathbb{Z} \mid b \bmod 7 = i\}$$

are cosets of $B_7$ in $(\mathbb{Z}, +)$ for $i = 0, 1, \ldots, 6$. Observe that $\{B_7 + i \mid i = 0, 1, \ldots, 6\}$ is a partition of $\mathbb{Z}$ into 7 disjoint classes.

**Observation 2.2.4.49.** If $(H, \circ)$ is a subgroup of a commutative group $(A, \circ)$, then all right cosets (left cosets) of $H$ in $(A, \circ)$ are cosets.

An important fact about cosets $H \circ b$ is, that their size is always equal to the size of $H$.

**Theorem 2.2.4.50.** *Let $(H, \circ)$ be a subgroup of $(A, \circ)$. Then the following facts hold.*

*(i) $H \circ h = H$ for all $h \in H$.*
*(ii) For all $b, c \in A$,*

$$\text{either } H \circ b = H \circ c \text{ or } H \circ b \cap H \circ c = \emptyset.$$

*(iii) If $H$ is finite, then*

$$|H \circ b| = |H|$$

*for all $b \in A$.*

*Proof.* We prove these three claims separately. Let $e$ be the identity of $(A, \circ)$ and $(H, \circ)$.

(i) Let $h \in H$. Since $H$ is closed under $\circ$, we obtain $a \circ h \in H$ for every $a \in H$, i.e.,

$$H \circ h \subseteq H.$$

Since $(H, \circ)$ is a group, $h^{-1} \in H$. Let $b$ be an arbitrary element of $H$. Then

$$b = b \circ e = b \circ \underbrace{(h^{-1} \circ h)}_{e} = \underbrace{(b \circ h^{-1})}_{\in H} \circ h \in H \circ h, \text{ i.e.,}$$

$$H \subseteq H \circ h.$$

Thus $H \circ h = H$.

(ii) Let $H \circ b \cap H \circ c \neq \emptyset$ for some $b, c \in A$. Then there exists $a_1, a_2 \in H$, such that

$$a_1 \circ b = a_2 \circ c.$$

This implies $c = a_2^{-1} \circ a_1 \circ b$, where $a_2^{-1} \in H$. Then

$$H \circ c = H \circ (a_2^{-1} \circ a_1 \circ b) = H \circ (a_2^{-1} \circ a_1) \circ b. \tag{2.22}$$

Since $a_2^{-1}, a_1 \in H$, the element $a_2^{-1} \circ a_1$ belongs to $H$, too. This implies, because of (i), that

$$H \circ (a_2^{-1} \circ a_1) = H. \tag{2.23}$$

Thus, combining (2.22) and (2.23) we obtain

$$H \circ c = H \circ (a_2^{-1} \circ a_1) \circ b = H \circ b.$$

(iii) Let $H$ be finite, and let $b \in A$. Since $H \circ b = \{h \circ b \mid h \in H\}$, we immediately have

$$|H \circ b| \leq |H|.$$

Let $H = \{h_1, h_2, \ldots, h_k\}$ for some $k \in \mathbb{N}$. We have to show that

$$|\{h_1 \circ b, h_2 \circ b, \ldots, h_k \circ b\}| \geq k, \text{ i.e., that } h_i \circ b \neq h_j \circ b$$

for all $i, j \in \{1, \ldots, k\}$ with $i \neq j$. Since $(A, \circ)$ is a group, $b^{-1} \in A$ and so $h_i \circ b = h_j \circ b$ would imply $h_i \circ b \circ b^{-1} = h_j \circ b \circ b^{-1}$. Thus,

$$h_i = h_i \circ (b \circ b^{-1}) = h_j \circ (b \circ b^{-1}) = h_j$$

would contradict the assumption $h_i \neq h_j$.

□

As a consequence of Theorem 2.2.4.50 we obtain that one can partition the set $A$ of every group $(A, \circ)$, that has a proper subgroup $(H, \circ)$, into pairwise disjoint subsets of $A$, which are the left (right) cosets of $H$ in $(A, \circ)$.

**Theorem 2.2.4.51.** *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. Then $\{H \circ b \mid b \in A\}$ is a partition of $A$.*

*Proof.* The claim (ii) of Theorem 2.2.4.50 shows that $H \circ b \cap H \circ c = \emptyset$ or $H \circ b = H \circ c$. So, it remains to show that $A \subseteq \bigcup_{b \in A} H \circ b$. But this is obvious because the identity $e$ of $(A, \circ)$ is also the identity of $(H, \circ)$, and so $b = e \circ b \in H \circ b$ for every $b \in A$. □

**Definition 2.2.4.52.** *Let $(H, \circ)$ be a subgroup of a group $(A, \circ)$. We define the* **index of $H$ in $(A, \circ)$** *as*

$$\boldsymbol{Index_H(A)} = |\{H \circ b \mid b \in A\}|,$$

*i.e., as the number of different right cosets of $H$ in $(A, \circ)$.*

The following theorem of Lagrange is the main reason for our study of group theory. It provides a powerful instrument for proving that there are not too many "bad" elements with some special properties in a group $(A, \circ)$, because it is sufficient to show that all "bad" elements are in a proper subgroup of $(A, \circ)$. The following assertion claims that the size of any proper subgroup of $(A, \circ)$ is at most $|A|/2$.

**Theorem 2.2.4.53 (Lagrange's Theorem).** *For any subgroup $(H, \circ)$ of a finite group $(A, \circ)$,*

$$|A| = Index_H(A) \cdot |H|,$$

*i.e., $|H|$ divides $|A|$.*

*Proof.* Following Theorem 2.2.4.51, $A$ can be divided in $Index_H(A)$ right cosets, which are pairwise disjoint and all of the same size $|H|$. □

**Corollary 2.2.4.54.** *Let $(H, \circ)$ be a subalgebra of a finite group $(A, \circ)$. If $H \subset A$, then*

$$|H| \leq |A|/2.$$

*Proof.* Theorem 2.2.4.45 ensures that $(H, \circ)$ is a subgroup of $(A, \circ)$. Following Lagrange's Theorem

$$|A| = Index_H(A) \cdot |H|.$$

Since $H \subset A$, $1 \leq |H| < |A|$ and so

$$Index_H(A) \geq 2.$$

□

**Corollary 2.2.4.55.** *Let $(A, \circ)$ be a finite group. Then, for every element $a \in A$, the order of $a$ divides $|A|$.*

*Proof.* Let $a$ be an arbitrary element of $A$. Following Lemma 2.2.4.47 $(H(a), \circ)$ with $H(a) = \{e, a, a^2, \ldots, a^{order(a)-1}\} = \{a, a^2, \ldots, a^{order(a)}\}$ is a subgroup of $(A, \circ)$.
Since $|H(a)| = order(a)$, Lagrange's Theorem implies

$$|A| = Index_{H(a)}(A) \cdot order(a).$$

□

In Chapter 5 we will often work with the sets

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid gcd(a,n) = 1\}$$

for $n \in \mathbb{N} - \{0\}$. In Theorem 2.2.4.40 we proved that $(\mathbb{Z}_p^*, \odot_p)$ is a commutative group for every prime $p$. Note that $\mathbb{Z}_p^* = \mathbb{Z}_p - \{0\} = \{1, 2, \ldots, p-1\}$ for every prime $p$. We prove now, that $(\mathbb{Z}_n^*, \odot_n)$ is a group for every $n \in \mathbb{N} - \{0\}$.

**Theorem 2.2.4.56.** *For every $n \in \mathbb{N} - \{0\}$, $(\mathbb{Z}_n^*, \odot_n)$ is a commutative group.*

*Proof.* First of all we have to show that $\mathbb{Z}_n^*$ is closed under $\odot_n$. Let $a, b$ be arbitrary elements of $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid gcd(a,n) = 1\}$. We have to show that $a \odot_n b \in \mathbb{Z}_n^*$, i.e., that $gcd(a \odot_n b, n) = 1$. Let us assume the contrary, i.e., that $gcd(a \odot_n b, n) = k$ for some $k \geq 2$. Then $n = k \cdot v$ for some $v \in \mathbb{N} - \{0\}$ and $a \cdot b \bmod n = k \cdot d$ for some $d \in \mathbb{N} - \{0\}$. This implies

$$a \cdot b \bmod kv = kd, \text{ i.e., } a \cdot b = kv \cdot s + kd$$

for some $s \in \mathbb{N}$. Since $kvs + kd = k(vs + d)$, $k$ divides $a \cdot b$. Let $k = p \cdot m$, where $p$ is a prime. Obviously, either $p$ divides $a$ or $p$ divides $b$. But this is the contradiction to the facts $n = p \cdot m \cdot v$, $gcd(a,n) = 1$, and $gcd(b,n) = 1$. Thus, $a \cdot b \bmod p \in \mathbb{Z}_n^*$.

Clearly, 1 is the identity element of $(\mathbb{Z}_n^*, \odot_n)$. Let $a$ be an arbitrary element of $\mathbb{Z}_n^*$. To prove that there exists an inverse $a^{-1}$ with $a \odot_n a^{-1} = 1$, it is sufficient to show that

$$|\{a \odot_n 1, a \odot_n 2, \ldots, a \odot_n (n-1)\}| = n - 1,$$

which implies $1 \in \{a \odot_n 1, \ldots, a \odot_n (n-1)\}$. Let us assume the contrary. Let there exist $i, j, i > j$, such that

$$a \odot_n i = a \odot_n j, \text{ i.e., } a \cdot i \equiv a \cdot j \pmod{n}.$$

This implies

$$a \cdot i = nk_1 + z \text{ and } a \cdot j = n \cdot k_2 + z$$

for some $k_1, k_2, z \in \mathbb{N}$ with $z < n$. Then

$$a \cdot i - a \cdot j = nk_1 - nk_2 = n(k_1 - k_2),$$

and so

$$a \cdot (i - j) = n(k_1 - k_2), \text{ i.e., } n \text{ divides } a \cdot (i - j).$$

Since $gcd(a,n) = 1$, $n$ must divide $(i-j)$. But this is impossible, because $i - j < n$. Thus, we can infer that $(\mathbb{Z}_n^*, \odot_n)$ is a group. Since $\odot_n$ is a commutative operation, $(\mathbb{Z}_n^*, \odot_n)$ is a commutative group. □

Next we show that $\mathbb{Z}_n^*$ contains all elements of $\mathbb{Z}_n$ that have an inverse element with respect to $\odot_{\text{mod } n}$, i.e., that

$$\begin{aligned}\mathbb{Z}_n^* &= \{a \in \mathbb{Z}_n \mid gcd(a,n) = 1\} \\ &= \{a \in \mathbb{Z}_n \mid \exists a^{-1} \in \mathbb{Z}_n \text{ such that } a \odot_{\text{mod } n} a^{-1} = 1\}.\end{aligned} \tag{2.24}$$

Theorem 2.2.4.56 implies that $\mathbb{Z}_n^* \subseteq \{a \in \mathbb{Z}_n \mid \exists a^{-1} \in \mathbb{Z}_n\}$ because $(\mathbb{Z}_n^*, \odot_{\text{mod } n})$ is a group. Thus, the following lemma completes the proof of (2.24).

**Lemma 2.2.4.57.** *Let $a \in \mathbb{Z}_n$. If there exists an $a^{-1} \in \mathbb{Z}_n$ such that $a \odot_n a^{-1} = 1$, then*

$$gcd(a,n) = 1.$$

*Proof.* Theorem 2.2.4.36 claims that

$$gcd(a,n) = \min\{d \in \mathbb{N} - \{0\} \mid d = ax + by \text{ for } x, y \in \mathbb{Z}\}.$$

Let there exist an element $a^{-1}$ with $a \odot_{\text{mod } n} a^{-1} = 1$. So $a \cdot a^{-1} \equiv 1 \pmod{n}$, i.e.,

$$a \cdot a^{-1} = k \cdot n + 1 \qquad \text{for a } k \in \mathbb{N}.$$

Choosing $x = a^{-1}$ and $y = -k$ we obtain

$$a \cdot a^{-1} + n \cdot (-k) = k \cdot n + 1 - k \cdot n = 1 \in Com(a,n)$$

and so $gcd(a,n) = 1$. □

We conclude this section by proving a fundamental result of the group theory. Let $\varphi(n) = |\mathbb{Z}_n^*|$ for any $n \in \mathbb{N}$ be the so-called **Euler's number**.

**Theorem 2.2.4.58 (Euler's Theorem).** *For all positive integers $n \geq 1$*

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

*for all $a \in \mathbb{Z}_n^*$.*

*Proof.* Let $a$ be an arbitrary element from $\mathbb{Z}_n^*$. Corollary 2.2.4.55 implies that $order(a)$ in $(\mathbb{Z}_n^*, \odot_n)$ divides $|\mathbb{Z}_n^*| = \varphi(n)$. Since $a^{order(a)} \equiv 1 \pmod{n}$ we obtain

$$\begin{aligned}a^{\varphi(n)} \bmod n &= (a^{order(a)})^{\varphi(n)/order(a)} \bmod n \\ &= (a^{order(a)} \bmod n)^{\varphi(n)/order(a)} \bmod n \\ &= 1^{\varphi(n)/order(a)} \bmod n = 1\end{aligned}$$

□

**Exercise 2.2.4.59.** Apply Euler's Theorem in order to give an alternative (algebraic) proof of Fermat's Theorem.

Keywords introduced in Section 2.2.4

group, semigroup, ring, field, prime, greatest common divisor, Euclid's algorithm, generator of a group, cyclic group, order of a group element, coset, index of a group

Summary of Section 2.2.4

The main assertions presented in Section 2.2.4 are:

- There are infinitely many primes and the number of primes from $\{2, 3, 4, \ldots, n\}$ is approximately $\frac{n}{\ln n}$ (Prime Number Theorem).
- Every integer greater than 1 can be expressed as a product of nontrivial powers of distinct primes and this prime factorization is unique (Fundamental Theorem of Arithmetics).
- $p$ is prime if and only if $a^{(p-1)/2} \bmod p \in \{1, -1\}$ for every $a \in \{1, 2, \ldots, p-1\}$.
- If $n = p_1 \cdot p_2 \cdot \cdots \cdot p_k$ for primes $p_1, \ldots, p_k$, then $\mathbb{Z}_n$ is isomorphic to $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_k}$ (Chinese Remainder Theorem).
- If $(H, \circ)$ is a subgroup of a group $(A, \circ)$, then $|H|$ divides $|A|$.
- The order of every element $a$ of a group $(A, \circ)$ divides $|A|$.
- For every $\mathbb{Z}_n^*, n \in \mathbb{N}$, and every $a \in \mathbb{Z}_n^*$, $a^{\varphi(n)} \bmod n = 1$ (Euler's Theorem).

### 2.2.5 Probability Theory

The probability theory has been developed to study experiments with uncertain outcomes. The fundamental notions of probability theory are "sample space" and "elementary event". A **sample space** $S$ is the set of all basic events that may happen in some experiment. Every element of $S$ is called an **elementary event**. Intuitively, in some experiment, the sample space is the set of all results (events) that may be the outcomes of the experiment. For instance, in flipping a coin one can consider the following two outcomes: "head" and "tail". Thus, $\{head, tail\}$ is the sample space and *head* and *tail* are the elementary events. If one flips three coins (one after each other or at once), then the sample space is $\{(x, y, z) \mid x, y, z \in \{head, tail\}\}$. The sequences $(head, head, tail)$, $(tail, head, tail)$, and $(head, head, head)$ are examples of elementary events. Intuitively, an elementary event is an event that cannot be expressed as a collection of smaller, more fundamental events, i.e., an elementary event cannot be partitioned into pieces from the point of view of the experiment considered.

Another example of an experiment is a fixed (randomized) algorithm. During the computation on an input $x$ such a randomized algorithm $A$ may have a choice from several possibilities on how to continue. Thus, it may execute different computations depending on the choices. From this point of view, the sample space $S_{A,x}$ is the set of all possible sequences of random choices, or equivalently the set of all computations of $A$ on $x$ (one computation for each sequence of random choices). Thus, every computation of $A$ on $x$ can be considered as an elementary event.

For our purposes it is sufficient to consider that $S$ is countable. So, all following definitions of fundamental notions assume the countability of $S$. An **event** is any subset of the sample space $S$. The event $S$ is called the **certain event**, and the event $\emptyset$ is called the **null event**. Two events $S_1, S_2 \subseteq S$ are called **mutually exclusive** if $S_1 \cap S_2 = \emptyset$. For instance, for $S = \{(x,y,z) \mid x,y,z \in \{head, tail\}\}$, $S_1 = \{(head, head, head), (tail, head, tail)\}$ and $S_2 = \{(tail, head, head), (head, tail, head), (head, head, tail)\}$ are two events that are mutually exclusive. Considering $S_{A,x}$, one may be interested in computations of $A$ on $x$, in which the correct output is computed. Then, one considers the set $Cor(A,x)$ of all computations providing the right output as an event. $Cor(A,x)$ is mutually exclusive to the event consisting of all computations that finish with a wrong output.

In what follows we have the following scenario. For a finite sample space $S$, we want to assign a probability to every elementary event. This assignment should correspond to the reality, i.e., to the experiment executed. To be fair, we have (among others) the following requirements on this assignment:

(i) Every elementary event has some non-negative probability.
(ii) The sum of the probabilities of all elementary events gives certainty (denoted by 1 in probability theory).
(iii) There is a fair way to compute the probability of any event $S_1 \subseteq S$ (among others, the probability of $S_1$ plus the probability of the complement $S - S_1$ must be certainty).

This results in the following definition.

**Definition 2.2.5.1.** *A* **probability distribution** ***Prob*** *on a sample space $S$ is a mapping from events of $S$ to real numbers ($Prob : 2^S \rightarrow \mathbb{R}^{\geq 0}$) such that the following* **probability axioms** *are satisfied:*

*(1) $Prob(\{x\}) \geq 0$ for every elementary event $x$,*
*(2) $Prob(S) = 1$,*
*(3) $Prob(X \cup Y) = Prob(X) + Prob(Y)$ for any two mutually exclusive events $X$ and $Y$ ($X \cap Y = \emptyset$).*
*(If one considers an infinite $S$, then one requires $Prob\left(\bigcup_{i=1}^{\infty} X_i\right) = \sum_{i=1}^{\infty} Prob(X_i)$ for every countable infinite sequence of mutually exclusive events $X_1, X_2, X_3, \ldots$)*

***Prob*$(X)$** *is called the* **probability of the event $X$**.

If one considers $S = \{head, tail\}$ for a fair coin flipping, then $Prob(\{head\}) = Prob(\{tail\}) = 1/2$. One can simply observe that the probabilities of the elementary events unambiguously determine the probabilities of all events in every sample space. The proof of the following simple observation is left to the reader.

**Exercise 2.2.5.2.** Prove for all events $X, Y$ of a sample space $S$ and every probability distribution $Prob$ on $S$ that

(i) $Prob(\emptyset) = 0$,
(ii) if $X \subseteq Y$, then $Prob(X) \leq Prob(Y)$,
(iii) $Prob(S - X) = 1 - Prob(X)$,
(iv) $Prob(X \cup Y) = Prob(X) + Prob(Y) - Prob(X \cap Y) \leq Prob(X) + Prob(Y)$.

□

In what follows we always consider that $S$ is finite or countably infinite. In this case we speak about **discrete probability distribution**. If, for every elementary event $x$ of a finite $S$,

$$Prob(\{x\}) = \frac{1}{|S|},$$

then *Prob* is called the **uniform probability distribution on $S$**.

**Example 2.2.5.3.** Consider the sample space

$$S = \{(x, y, z) \mid x, y, z \in \{head, tail\}\}$$

for the experiment of flipping three coins. If we consider fair coins, it means that $Prob(\{a\}) = \frac{1}{|S|} = \frac{1}{8}$ for every elementary event $a \in S$.

What is the probability of getting at least one head? This event is $Head = \{(head, head, head), (head, head, tail), (head, tail, head), (head, tail, tail), (tail, head, tail), (tail, tail, head), (tail, head, head)\}$. Thus,

$$Prob(Head) = \sum_{a \in Head} Prob(\{a\}) = \frac{7}{8}.$$

A more convenient way to evaluate $Prob(Head)$ is to say that $S - Head = \{(tail, tail, tail)\}$, and so $Prob(S - Head) = \frac{1}{8}$ directly implies

$$Prob(Head) = 1 - Prob(S - Head) = \frac{7}{8}.$$

□

**Exercise 2.2.5.4.** Let $n \geq k \geq 0$ be two integers. Consider the experiments of flipping $n$ coins and the corresponding sample space $S = \{(x_1, x_2, \ldots, x_n) \mid x_i \in \{head, tail\}$ for $i = 1, \ldots, n\}$. What is the probability of getting exactly $k$ heads if *Prob* is the uniform probability distribution on $S$? □

The notions defined above are suitable when looking at an experiment only once, i.e., at the very end. But sometimes we can obtain partial information about the outcome of the experiment by some intermediate observation. For instance, we flip three coins one after each other and we look at the result of the first coin flipping. Knowing this result we ask what the probability is of getting at least two heads in the whole experiment. Or, somebody tells you that the result $(x, y, z)$ contains at least one head and knowing this fact you have to estimate the probability that $(x, y, z)$ contains at least two heads. The tasks of this kind result in the following definition of conditional probabilities.

**Definition 2.2.5.5.** *Let $S$ be a sample space with a probability distribution Prob. The* **conditional probability** *of an event $X \subseteq S$ given that another event $Y \subseteq S$ occurs (with certainty) is*

$$\boldsymbol{Prob(X|Y)} = \frac{Prob(X \cap Y)}{Prob(Y)},$$

*whenever $Prob(Y) \neq 0$. We also say that $Prob(X|Y)$ is* **the probability of $X$ given $Y$**.

Observe that the definition of the conditional probability is natural. $X \cap Y$ consists of elementary events that are in both $X$ and $Y$. Since we know that $Y$ happens, it is clear that no event from $X - Y$ can happen. Dividing $Prob(X \cap Y)$ by $Prob(Y)$ we normalize the probabilities of all elementary events in $Y$ since

$$\sum_{e \in Y} \frac{Prob(\{e\})}{Prob(Y)} = \frac{1}{Prob(Y)} \cdot \sum_{e \in Y} Prob(\{e\}) = \frac{1}{Prob(Y)} \cdot Prob(Y) = 1.$$

Intuitively it means that we exchange $S$ by $Y$, because $Y$ happens with certainty. Thus, the conditional probability of $X$ given $Y$ is the ratio of the probability of the event $X \cap Y$ to the probability of $Y$.

**Example 2.2.5.6.** Let us again consider the experiment of flipping three coins. Let $X$ be the event that the result contains at least two heads, and let $Y$ be the event that the result contains at least one head. Since $X \cap Y = X$ we obtain

$$Prob(X|Y) \underset{def.}{=} \frac{Prob(X \cap Y)}{Prob(Y)} = \frac{Prob(X)}{Prob(Y)} = \frac{\frac{4}{8}}{\frac{7}{8}} = \frac{4}{7}.$$

□

**Definition 2.2.5.7.** *Let $S$ be a sample space with a probability distribution Prob. Two events $X, Y \subseteq S$ are* **independent** *if*

$$Prob(X \cap Y) = Prob(X) \cdot Prob(Y).$$

The following observation relates independence with conditional probability, and it provides an equivalent definition of the independence of two events.

**Observation 2.2.5.8.** Let $S$ be a sample space with a probability distribution $Prob$. Let $X, Y \subseteq S$, and let $Prob(Y) \neq 0$. Then $X$ and $Y$ are independent if and only if $Prob(X|Y) = Prob(X)$.

*Proof.* (i) If $X$ and $Y$ are independent, then $Prob(X \cap Y) = Prob(X) \cdot Prob(Y)$. So,

$$Prob(X|Y) \underset{def.}{=} \frac{Prob(X \cap Y)}{Prob(Y)} = \frac{Prob(X) \cdot Prob(Y)}{Prob(Y)} = Prob(X).$$

(ii) Let $Prob(X|Y) = Prob(X)$ and $Prob(Y) \neq 0$. Then

$$Prob(X) = Prob(X|Y) \underset{def.}{=} \frac{Prob(X \cap Y)}{Prob(Y)},$$

which directly implies $Prob(X \cap Y) = Prob(X) \cdot Prob(Y)$. □

Due to Observation 2.2.5.8 we see that if two events $X$ and $Y$ are independent, then the knowledge that $X$ ($Y$) occurs with certainty does not change the probability $Prob(Y)$ ($Prob(X)$). This corresponds to our intuitive meaning of the independence of two events $X$ and $Y$ that if one knows that the experiment results in an event $X$ we cannot obtain any partial information about the correspondence between this result and the event $Y$.

**Example 2.2.5.9.** Consider our standard experiment of flipping three coins. Let

$$X = \{(head, head, head), (head, head, tail), (head, tail, head), (head, tail, tail)\}$$

be the event that the result of the first coin flipping is *head*. Obviously, $Prob(X) = \frac{4}{8} = \frac{1}{2}$. Let

$$Y = \{(head, tail, head), (head, tail, tail), (tail, tail, head), (tail, tail, tail)\}$$

be the event that the result of the second coin flipping is *tail*. Clearly, $Prob(Y) = \frac{1}{2}$. Since $X \cap Y = \{(head, tail, head), (head, tail, tail)\}$,

$$Prob(X \cap Y) = \frac{2}{8} = \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2} = Prob(X) \cdot Prob(Y).$$

Thus, $X$ and $Y$ are independent and this corresponds to our intuition because the result of the first coin flipping does not have any influence on the result of the second coin flipping. □

**Exercise 2.2.5.10.** Determine all pairs of independent events of the experiments from the above example. □

**Exercise 2.2.5.11.** Let $S$ be a sample space with a probability distribution $Prob$. Prove that for all events $A_1, A_2, \ldots, A_n \subseteq S$ such that $Prob(A_1) \neq \emptyset, Prob(A_1 \cap A_2) \neq \emptyset, \ldots, Prob(A_1 \cap A_2 \cap \cdots \cap A_{n-1}) \neq \emptyset$,

$$\begin{aligned} Prob(A_1 \cap A_2 \cap \cdots \cap A_n) = {} & Prob(A_1) \cdot Prob(A_2|A_1) \\ & \cdot Prob(A_3|A_1 \cap A_2) \cdot \cdots \\ & \cdot Prob(A_n|A_1 \cap A_2 \cap \cdots \cap A_{n-1}). \end{aligned}$$ □

**Theorem 2.2.5.12 (Bayes' Theorem).** *Let $S$ be a sample space with a probability distribution $Prob$. For every two events $X, Y \subseteq S$ with nonzero probability,*

$$(i)\ Prob(X|Y) = \frac{Prob(X) \cdot Prob(Y|X)}{Prob(Y)},$$

$$(ii)\ Prob(X|Y) = \frac{Prob(X) \cdot Prob(Y|X)}{Prob(X) \cdot Prob(Y|X) + Prob(S-X) \cdot Prob(Y|S-X)}.$$

*Proof.* (i) From the definition of the conditional probability we obtain

$$Prob(X \cap Y) = Prob(Y) \cdot Prob(X|Y) = Prob(X) \cdot Prob(Y|X).$$

The last equality implies directly

$$Prob(X|Y) = \frac{Prob(X) \cdot Prob(Y|X)}{Prob(Y)}.$$

(ii) To get the equality (ii) from (i) we show that $Prob(Y)$ can be expressed as

$$Prob(X) \cdot Prob(Y|X) + Prob(S-X) \cdot Prob(Y|S-X).$$

Since $Y = (Y \cap X) \cup (Y \cap (S-X))$ and $(Y \cap X) \cap (Y \cap (S-X)) = \emptyset$,

$$\begin{aligned} Prob(Y) &= Prob(Y \cap X) + Prob(Y \cap (S-X)) \\ &= Prob(X) \cdot Prob(Y|X) + Prob(S-X) \cdot Prob(Y|S-X). \end{aligned}$$

□

Now, we define a notion that is crucial for the analysis of the behavior of randomized algorithms. Let $S$ be a sample space.[20] Any function $F$ from $S$ to $\mathbb{R}$ is called a (**discrete**) **random variable** on $S$. This means that we associate a real number with every elementary event of $S$ (outcome of the experiment). To see a motivation for this notion one can consider the work of a randomized algorithm $A$ on a fixed input $x$ as the experiment and one run (computation) of $A$ as an elementary event. If $F$ assigns the length (time complexity) of $C$ to every run $C$ of $A$, then we can analyze the "expected" time complexity of $A$ by the probability distribution induced by $F$ on $\mathbb{R}$. Then one can ask what the probability is that $A$ computes the output in a given time $t$ or vice versa – what is the smallest time $t'$ such that $A$ finishes the work in time $t'$ with the probability of at least $1/2$. Another possibility to define $F$ is that $F$ assigns 1 to a particular run $C$ of $A$ on $x$ if the output produced in this run is correct, and $F$ assigns 0 to $C$ if the output is wrong. Then the "average" (expected) value of $F$ gives us the information about the reliability of the algorithm $A$. Thus, the free choice of a random variable provides a powerful instrument for the analysis of the behavior of the considered random experiment. Note that an appropriate choice of random variables does not only decide which properties of the experiment will be investigated, but it also may influence the success of this analytic approach as well as the difficulty (efficiency) of the execution of this analysis.

[20] Remember that we assume $S$ is either finite or countably infinite.

**Definition 2.2.5.13.** *Let $S$ be a sample space with a probability distribution Prob, and let $F$ be a random variable on $S$. For every $x \in \mathbb{R}$, we define the event $F = x$ by*

$$Event(F = x) = \{s \in S \mid F(s) = x\}.$$

*The function $f_F : \mathbb{R} \to [0,1]$ defined by*

$$\boldsymbol{f_F(x)} = Prob(Event(F = x)),$$

*is called the* **probability density function** *of the random variable $F$.*

*The* **distribution function** *of $F$ is a function $Dis_F : \mathbb{R} \to [0,1]$ defined by*

$$\boldsymbol{Dis_F(x)} = Prob(F \leq x) = \sum_{y \leq x} Prob(Event(F = y)).$$

In what follows, we use the notation $F = x$ instead of $Event(F = x)$, and so the notation $Prob(F = x)$ instead of $Prob(Event(F = x))$.

**Observation 2.2.5.14.** Let $S$, $Prob$, and $F$ be as in Definition 2.2.5.13. Then, for every $x \in \mathbb{R}$,

(i) $f_F(x) = Prob(F = x) = \sum_{\{s \in S \mid F(s) = x\}} Prob(\{s\})$,
(ii) $Prob(F = x) \geq 0$, and
(iii) $\sum_{y \in \mathbb{R}} Prob(F = y) = 1$.

□

**Example 2.2.5.15.** Consider the experiment of rolling three 6-sided dices. The outcome of rolling one dice is one of the numbers 1, 2, 3, 4, 5, and 6 and we consider that the probability distribution is uniform. So, $S = \{(a,b,c) \mid a,b,c \in \{1,2,3,4,5,6\}\}$ and $Prob(\{s\}) = \frac{1}{6^3} = \frac{1}{216}$ for every $s \in S$. Define the random variable $F$ to be the sum of the values on all three dices. For instance, $F((3,1,5)) = 3+1+5 = 9$. The probability of the event $F = 5$ is

$$\begin{aligned} Prob(F = 5) &= \sum_{\substack{s \in S \\ F(s)=5}} Prob(\{s\}) = \sum_{\substack{a+b+c=5 \\ a,b,c \in \{1,2,\ldots,6\}}} Prob(\{(a,b,c)\}) \\ &= Prob(\{(1,1,3)\}) + Prob(\{(1,3,1)\}) \\ &\quad + Prob(\{(3,1,1)\}) + Prob(\{(1,2,2)\}) \\ &\quad + Prob(\{(2,1,2)\}) + Prob(\{(2,2,1)\}) \\ &= 6 \cdot \frac{1}{216} = \frac{1}{36}. \end{aligned}$$

Let $G$ be the random variable defined by $G((a,b,c)) = \max\{a,b,c\}$ for every elementary event $(a,b,c) \in S$.

$$Prob(G = 3) = \sum_{\substack{s \in S \\ G(s)=3}} Prob(\{s\}) = \sum_{\substack{\max\{a,b,c\}=3 \\ a,b,c \in \{1,2,\ldots,6\}}} Prob(\{(a,b,c)\})$$

$$
\begin{aligned}
&= \sum_{b,c\in\{1,2\}} Prob(\{(3,b,c)\}) + \sum_{a,c\in\{1,2\}} Prob(\{(a,3,c)\}) \\
&\quad + \sum_{a,b\in\{1,2\}} Prob(\{(a,b,3)\}) + \sum_{a\in\{1,2\}} Prob(\{(3,3,a)\}) \\
&\quad + \sum_{b\in\{1,2\}} Prob(\{(3,b,3)\}) + \sum_{a,c\in\{1,2\}} Prob(\{(a,3,3)\}) \\
&\quad + Prob(\{(3,3,3)\}) \\
&= \frac{4}{216} + \frac{4}{216} + \frac{4}{216} + \frac{2}{216} + \frac{2}{216} + \frac{2}{216} + \frac{1}{216} = \frac{19}{216}.
\end{aligned}
$$

□

Thus, we have seen that one can define several random variables on the same sample space.

**Definition 2.2.5.16.** *Let $S$ be a sample space with a probability distribution Prob, and let $X$ and $Y$ be two random variables on $S$. The* **joint probability density function** *of $X$ and $Y$ is the function $f_{X,Y} : \mathbb{R} \times \mathbb{R} \to [0,1]$ defined by*

$$f_{X,Y}(x,y) = Prob(X = x \text{ and } Y = y) = Prob(Event(X = x) \cap Event(Y = y)).$$

*$X$ and $Y$ are* **independent** *if, for all $x, y \in \mathbb{R}$,*

$$Prob(X = x \text{ and } Y = y) = Prob(X = x) \cdot Prob(Y = y).$$ □

Note that the above definition of the independence of $X$ and $Y$ is a natural extension of the notion of the independence of two events. Obviously,

$$Prob(X = x) = \sum_{y\in\mathbb{R}} Prob(X = x \text{ and } Y = y),$$

and

$$Prob(Y = y) = \sum_{x\in\mathbb{R}} Prob(X = x \text{ and } Y = y).$$

Applying the notion of conditional probabilities

$$Prob(X = x | Y = y) = \frac{Prob(X = x \text{ and } Y = y)}{Prob(Y = y)}.$$

Thus, if $Event(X = x)$ and $Event(Y = y)$ are independent, then $Prob(X = x|Y = y) = Prob(X = x)$ and we obtain the definition of the independence of $X$ and $Y$.

The simplest and most useful characterization of the distribution of a random variable is the average of the values it takes on. This average value will be called the expected value in what follows. A good algorithmic motivation for the study of the expected value may be the relation to the analysis of the behavior and the time complexity of randomized algorithms.

**Definition 2.2.5.17.** *Let $S$ be a sample space with a probability distribution Prob, and let $X$ be a random variable on $S$. The* **expected value** *(or* **expectation** *of $X$) is*

$$\boldsymbol{E[X]} = \sum_{x \in \mathbb{R}} x \cdot Prob(X = x)$$

*if the sum is finite or converges absolutely.*

**Exercise 2.2.5.18.** Let $S$ be a sample space with a probability distribution *Prob*, and let $X$ be an random variable on $S$. Prove that

$$E[X] = \sum_{s \in S} X(s) \cdot Prob(\{s\}).$$

□

**Example 2.2.5.19.** Consider the experiment of rolling one 6-sided dice, and the random variable $F$ defined by $F(a) = a$ for $a \in S = \{1, 2, \ldots, 6\}$.

$$E[F] = \sum_{a \in S} F(a) \cdot Prob(F = a) = \sum_{a \in S} a \cdot \frac{1}{6} = \frac{1}{6} \cdot \sum_{a \in S} a = \frac{1}{6} \cdot \sum_{i=1}^{6} i = \frac{21}{6} = \frac{7}{2}.$$ □

In what follows, a discrete random variable is called an **indicator variable** if it takes only values 0 and 1. An indicator variable $X$ is used to denote the occurrence or nonoccurrence of an event $E$, where

$$E = \{s \in S \mid X(s) = 1\} \text{ and } S - E = \{s \in S \mid X(s) = 0\}.$$

The above mentioned variable $F$, with $F(C) = 1$ if the run $C$ of a randomized algorithm computes the right output, is an example of an indicator variable.

If $X$ is a random variable, and $g$ a function from $\mathbb{R}$ to $\mathbb{R}$, then $g(X)$ is a random variable, too. If the expectation of $g(X)$ is defined, then clearly[21]

$$E[g(X)] = \sum_{x \in \mathbb{R}} g(x) \cdot Prob(X = x).$$

Particularly, if $g(x) = r \cdot X$, then

$$E[g(x)] = E[r \cdot X] = r \cdot E[X].$$

**Observation 2.2.5.20.** Let $S$ be a sample space with a probability distribution *Prob*, and let $X$ and $Y$ be two random variables. Then

$$E[X + Y] = E[X] + E[Y].$$ □

The property of expectations presented in Observation 2.2.5.20 is called **linearity of expectation.**

[21] See Exercise 2.2.5.18.

**Theorem 2.2.5.21.** *Let $S$ be a sample space with a probability distribution Prob. For any two independent random variables $X$ and $Y$ with defined $E[X]$ and $E[Y]$, respectively,*

$$E[X \cdot Y] = E[X] \cdot E[Y].$$

*Proof.*

$$\begin{aligned} E[X \cdot Y] &= \sum_{z \in \mathbb{R}} z \cdot Prob(X \cdot Y = z) \\ &= \sum_{x} \sum_{y} x \cdot y \cdot Prob(X = x \text{ and } Y = y) \\ &= \sum_{x} \sum_{y} xy Prob(X = x) \cdot Prob(Y = y) \\ &= \left( \sum_{x} x Prob(X = x) \right) \cdot \left( \sum_{y} y Prob(Y = y) \right) \\ &= E[X] \cdot E[Y]. \end{aligned}$$

□

**Definition 2.2.5.22.** *Let $S$ be a sample space with a probability distribution Prob. Let $X_1, X_2, \ldots, X_n$, $n \in \mathbb{N}^+$ be random variables on $S$. We say that $X_1, X_2, \ldots, X_n$ are* **mutually independent** *if, for all $x_1, x_2, \ldots, x_n \in \mathbb{R}$,*

$$\begin{aligned} &Prob(X_1 = x_1 \text{ and } X_2 = x_2 \text{ and } \ldots \text{and } X_n = x_n) = \\ &Prob(X_1 = x_1) \cdot Prob(X_2 = x_2) \cdot \cdots \cdot Prob(X_n = x_n). \end{aligned}$$

**Exercise 2.2.5.23.** Prove the following generalization of Theorem 2.2.5.21. For any $n$ random variables $X_1, X_2, \ldots, X_n$ that are mutually independent,

$$E[X_1 \cdot X_2 \cdot \cdots \cdot X_n] = E[X_1] \cdot E[X_2] \cdot \cdots \cdot E[X_n].$$

□

**Example 2.2.5.24.** Consider the experiment of consecutively rolling three 6-sided dices and the random variable $F$ defined by $F((a, b, c)) = 3a + 2b + c$ for every $(a, b, c) \in S$. We want to compute $E[F]$ without working with the sum $\sum_x x \cdot Prob(F = x)$ consisting of 215 additions. We define three random variables $F_1$, $F_2$, and $F_3$ as follows:

$$F_1(a, b, c) = a, F_2(a, b, c) = b, \text{ and } F_3(a, b, c) = c.$$

Obviously, $F = 3F_1 + 2F_2 + F_3$. Since $E[F_i] = 7/2$ for $i = \{1, 2, 3\}$ as computed in the previous example, we obtain

$$E[F] = 3 \cdot E[F_1] + 2 \cdot E[F_2] + E[F_3] = 6 \cdot \frac{7}{2} = 21.$$

□

The following two examples illustrate the usefulness of the notions random variable and expectation for practical purposes in algorithmics. The first

example shows how the investigation of $E[X]$ of a randomized variable $X$ can lead to the design of an efficient algorithm for a given task. The second example shows how to use these notions to analyze the complexity of a randomized algorithm.

**Example 2.2.5.25.** Let $F = F(x_1, \ldots, x_n)$ be a formula in conjunctive normal form over a set $\{x_1, \ldots, x_n\}$ of $n$ variables. Our aim is to find an assignment to $\{x_1, \ldots, x_n\}$ such that as many as possible clauses are satisfied. Using a simple probabilistic consideration we show that there exists an input assignment that satisfies at least half of the clauses.

Let $F$ consist of $m$ clauses, i.e., $F = F_1 \wedge F_2 \wedge \cdots \wedge F_m$. Suppose the following experiment. We choose the values of $x_1, x_2, \ldots, x_n$ randomly with $Prob(x_i = 1) = Prob(x_i = 0) = 1/2$ for $i = 1, 2, \ldots, n$. Now, we define $m$ random variables $Z_1, \ldots, Z_m$ where, for $i = 1, \ldots, m$, $Z_i(\alpha) = 1$ if $F_i$ is satisfied by the assigned $\alpha$ and $Z_i(\alpha) = 0$ otherwise. For every clause of $k$ distinct literals, the probability that it is not satisfied by a random variable assignment to the set of variables is $2^{-k}$, since this event takes place if and only if each literal gets the value 0, and the Boolean values are assigned independently to distinct literals in any clause. This implies that the probability that a clause with $k$ literals is satisfied is at least $1 - 2^{-k} \geq 1/2$ for every $k \geq 1$, i.e.,

$$E[Z_i] \geq 1/2$$

for all $i = 1, \ldots, m$.

Now, we define a random variable $Z$ as $Z = \sum_{i=1}^{m} Z_i$. Obviously, $Z$ counts the number of satisfied clauses. Because of the linearity of expectation

$$E[Z] = E\left[\sum_{i=1}^{m} Z_i\right] = \sum_{i=1}^{m} E[Z_i] \geq \sum_{i=1}^{m} \frac{1}{2} = \frac{m}{2}.$$

Thus, we are sure that there exists an assignment satisfying at least one half of the clauses of $F$. The following algorithm outputs an assignment whose expected number of satisfied clauses is at least $m/2$.

**Algorithm 2.2.5.26** (RANDOM ASSIGNMENT).

Input: A formula $F = F(x_1, \ldots, x_n)$ over $n$ variables $x_1, \ldots, x_n$ in CNF.
Step 1: Choose uniformly at random $n$ Boolean values $a_1, \ldots, a_n$ and set $x_i = a_i$ for $i = 1, \ldots, n$.
Step 2: Evaluate each clause of $F$ and set $Z :=$ the number of satisfied clauses.
Output: $(a_1, a_2, \ldots, a_n), Z$.

□

**Exercise 2.2.5.27.** Let $F$ be a formula in CNF whose every clause consists of at least $k$ distinct variables, $k \geq 2$. Which lower bound on $E[Z]$ can be proved in this case? □

**Example 2.2.5.28.** Consider the task[22] of sorting a set $S$ of $n$ elements into an increasing order. One of the well-known recursive algorithms for this task is the following RANDOMIZED QUICKSORT RQS($S$).

**Algorithm 2.2.5.29.** RQS($S$)
Input: A set $S$ of numbers.
Step 1: Choose an element $a$ uniformly at random from $S$.
{every element in $S$ has the probability $\frac{1}{|S|}$ of being chosen}
Step 2: $S_< := \{b \in S \mid b < a\}$;
$S_> := \{c \in S \mid c > a\}$;
Step 3: **output**(RQS($S_<$), $a$, RQS($S_>$)).
Output: The sequence of elements of $S$ in increasing order.

The goal of this example is to show that the notions of random variables and expectation can be helpful to estimate the "average" (expected) complexity of this algorithm. As usual for sorting, the complexity is measured in the number of comparisons of pairs of elements of $S$. Let $|S| = n$ for a positive integer $n$.

We observe that the complexity of Step 2 is exactly $|S| - 1 = n - 1$. Intuitively, the best random choices of elements from $S$ are choices dividing $S$ into two approximately equal sized sets $S_<$ and $S_>$. In the terminology of recursion this means that the original problems of the size $n$ are reduced to two problems of size $n/2$. So, if $T(n)$ denotes the complexity for this kind of choices, then

$$T(n) \leq 2 \cdot T(n/2) + n - 1.$$

Following Section 2.2.2 we already know that the solution of this recurrence is $T(n) \in O(n \cdot \log n)$. A very bad sequence of random choices is when the smallest element of the given set is always chosen. In this case the number of comparisons is

$$T(n) = \sum_{i=1}^{n-1} i \in \Theta(n^2).$$

Since one can still show that, for the recurrence inequality

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3}{4} \cdot n\right) + n - 1,$$

$T(n) \in O(n \log n)$, RQS will behave well also if the size $|S_<|$ of $S_<$ very roughly approximates $|S_>|$. But this happens when the probability is at least $1/2$ because at least half of the elements are good choices. This is the reason for our hope that algorithm RQS behaves very well in the average. In what follows we carefully analyze the expected complexity of RQS.

[22] One of the fundamental computing problems

Let $s_1, s_2, \ldots, s_n$ be the output[23] of the algorithm RQS. Our experiment is the sequence of random choices of RQS. We define the random variable $X_{ij}$ by

$$X_{ij}(C) = \begin{cases} 1 \text{ if } s_i \text{ and } s_j \text{ are compared in the run } C \text{ of RQS} \\ 0 \text{ otherwise} \end{cases}$$

for all $i, j \in \{1, \ldots, n\}$, $i < j$. Obviously, the random variable

$$T = \sum_{i=1}^{n} \sum_{j>i} X_{ij}$$

counts the total number of comparisons. So,

$$E[T] = E\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} E[X_{ij}] \tag{2.25}$$

is the expected complexity of the algorithm RQS.[24] It remains to estimate $E[X_{ij}]$.

Let $p_{ij}$ denote the probability that $s_i$ and $s_j$ are compared in an execution. Since $X_{ij}$ is either 1 or 0,

$$E[X_{ij}] = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}.$$

Now, consider for every $i, j \in \{1, \ldots, n\}$, $i < j$, the subsequence

$$s_i, s_{i+1}, \ldots, s_{i+j-1}, s_j.$$

If some $s_d$ with $i < d < j$ was randomly chosen by RQS($S$) before either $s_i$ or $s_j$ has been randomly chosen, then $s_i$ and $s_j$ were not compared.[25] If $s_i$ or $s_j$ has been randomly chosen to play the splitting role before any of the elements from $\{s_{i+1}, s_{i+2}, \ldots, s_{i+j-1}\}$ have been randomly chosen, then $s_i$ and $s_j$ were compared in the corresponding run[26] of RQS($S$). Since each of the elements of $\{s_i, s_{i+1}, \ldots, s_j\}$ is equal likely to be the first in the sequence of random choices,

$$p_{ij} = \frac{2}{j - i + 1}. \tag{2.26}$$

Inserting (2.26) into (2.25) we finally obtain

$$E[T] = \sum_{i=1}^{n} \sum_{j>i} p_{ij}$$

[23] That is, $s_1 < s_2 < \cdots < s_n$.

[24] Note that (2.25) holds because of the linearity of expectation.

[25] This is because $s_i \in S_<$ and $s_j \in S_>$ according to $d$.

[26] In the run corresponding to this sequence of random choices.

$$
\begin{aligned}
&= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} \\
&\leq \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2 \sum_{i=1}^{n} Har(n) \\
&= 2n \cdot Har(n) \approx 2 \cdot n \cdot \ln n + \Theta(n).
\end{aligned}
$$

Thus, as expected, the expected time complexity of RQS is in $O(n \log n)$. □

Keywords introduced in Section 2.2.5

sample space, event, probability distribution, conditional probability, random variable, probability density function, distribution function, expected value (expectation), independence of events, independence of random variables, linearity of expectation

## 2.3 Fundamentals of Algorithmics

### 2.3.1 Alphabets, Words, and Languages

All data are represented as strings of symbols. The kind of data representation is often important for the efficiency of algorithm implementations. Here, we present some elementary fundamentals of formal language theory. We do not need to deal too much with details of data representation because we consider algorithms on an abstract design level and do not often work with the details of implementation. The main goal of this section is to give definitions of notions that are sufficient for fixing the representation of some input data and thus to precisely formalize the definitions of some fundamental algorithmic problems. We also need the terms defined here for the abstract considerations of the complexity theory in Section 2.3.3 and for proving lower bounds on polynomial-time inapproximability in Section 4.4.2.

**Definition 2.3.1.1.** *Any non-empty, finite set is called an* **alphabet**. *Every element of an alphabet $\Sigma$ is called a* **symbol** *of $\Sigma$.*

An alphabet has the same meaning for algorithmics as for natural languages – it is a collection of signs or symbols used in a more or less uniform fashion by a number of people in order to represent information and so to

communicate with each other. Thus, alphabets are used for communication between human and machine, between computers, and in algorithmic information processing. A symbol of an alphabet is often considered as a possible content of the computer word. Fixing an alphabet means to fix all possible computer words in this interpretation. Examples of alphabets are

$$\begin{aligned} \Sigma_{bool} &= \{0,1\}, \\ \Sigma_{lat} &= \{a,b,c,\ldots,z\}, \\ \Sigma_{logic} &= \{0,1,(,),\wedge,\vee,\neg,x\}. \end{aligned}$$

**Definition 2.3.1.2.** *Let $\Sigma$ be an alphabet. A* **word** *over $\Sigma$ is any finite sequence of symbols of $\Sigma$. The* **empty word $\lambda$** *is the only word consisting of zero symbols. The set of all words over the alphabet $\Sigma$ is denoted by $\Sigma^*$.*

The interpretation of the notion "word over $\Sigma$" is a text consisting of symbols of $\Sigma$ rather than a term representing a notion. So, the contents of a book can be considered as a word over some alphabet including the symbol blank and symbols of $\Sigma_{lat}$.

$w = 0,1,0,0,1,0$ is a word over $\Sigma_{bool}$. In what follows we usually omit the commas and represent $w$ simply by 010010. So $abcxyzef$ is a word over $\Sigma_{lat}$. For $\Sigma = \{a,b\}$, $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, \ldots\}$.

**Definition 2.3.1.3.** *The* **length of a word $w$** *over an alphabet $\Sigma$, denoted by $|w|$, is the number of symbols in $w$ (i.e., the length of $w$ as a sequence). For every word $w \in \Sigma^*$, and every symbol $a \in \Sigma$, $\#_a(w)$ is the number of occurrences of the symbol $a$ in the word $w$.*

For the word $w = 010010$, $|w| = 6$, $\#_0(w) = 4$, and $\#_1(w) = 2$. We observe that for every alphabet $\Sigma$ and every word $w \in \Sigma^*$,

$$|w| = \sum_{a \in \Sigma} \#_a(w).$$

**Definition 2.3.1.4.** *Let $\Sigma$ be an alphabet. Then, for any $n \in \mathbb{N}$,*

$$\Sigma^n = \{x \in \Sigma^* \mid |x| = n\}.$$

For instance, $\{a,b\}^3 = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$. We define $\Sigma^+ = \Sigma^* - \{\lambda\}$.

**Definition 2.3.1.5.** *Given two words $v$ and $w$ over an alphabet $\Sigma$, we define the* **concatenation of $v$ and $w$**, *denoted by $vw$ (or by $v \cdot w$) as the word that consists of the symbols of $v$ in the same order, followed by the symbols of $w$ in the same order.*

*For every word $w \in \Sigma^*$, we define*

*(i)* $w^0 = \lambda$, *and*
*(ii)* $w^{n+1} = w \cdot w^n = ww^n$ *for every positive integer $n$.*

*A* **prefix** *of a word $w \in \Sigma^*$ is any word $v$ such that $w = vu$ for some word $u$ over $\Sigma$. A* **suffix** *of a word $w \in \Sigma^*$ is any word $u$ such that $w = xu$ for some word $x \in \Sigma^*$. A* **subword** *of a word $w$ over $\Sigma$ is any word $z \in \Sigma^*$ such that $w = uzv$ for some words $u, v \in \Sigma^*$.*

The word *abbcaa* is the concatenation of the words *ab* and *bcaa*. The words *abbcaa*, *a*, *ab*, *bca*, and *bbcaa* are examples of subwords of $abbcaa = ab^2ca^2$. The words *a*, *ab*, $ab^2$, $ab^2c$, $ab^2ca$, and $ab^2ca^2$ are all prefixes of *abbcaa*. *caa* and $a^2$ are examples of suffixes of *abbcaa*.

**Exercise 2.3.1.6.** Prove that, for every alphabet $\Sigma$, $(\Sigma^*, \cdot)$, where $\cdot$ is the operation of concatenation, is a monoid. □

In what follows we use words to code data and so to represent input and output data, as well as the contents of the computer memory. Since the complexity of algorithms is measured according to the input length, the first step in the complexity analysis is to fix the alphabet and the data representation over this alphabet. This automatically determines the length of every input. We usually code integers as binary words. For every $u = u_n u_{n-1} \dots u_2 u_1 \in \Sigma_{bool}^n$, $u_i \in \Sigma_{bool}$ for $i = 1, \dots, n$,

$$\textbf{\textit{Number}}(u) = \sum_{i=1}^{n} u_i \cdot 2^{i-1}$$

is the integer coded by $u$. Thus, for instance, $Number(000) = Number(0) = 0$, and $Number(1101) = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 1 + 0 + 4 + 8 = 13$.

One can use the alphabet $\{0, 1, \#\}$ to code graphs. If $M_G = [a_{ij}]_{i,j=1,\dots,n}$ is an adjacency matrix of a graph $G$ of $n$ vertices, then the word

$$a_{11}a_{12} \dots a_{1n}\#a_{21}a_{22} \dots a_{2n}\# \dots \#a_{n1}a_{n2} \dots a_{nn}$$

can be used to code $G$.

**Exercise 2.3.1.7.** Design a representation of graphs by words over $\Sigma_{bool}$. □

**Exercise 2.3.1.8.** Design a representation of weighted graphs, where weights are some positive integers, using the alphabet $\{0, 1, \#\}$. □

One can use the alphabet $\Sigma_{logic}$ to represent formulae over a variable set $X = \{x_1, x_2, x_3, \dots\}$ and operations $\vee$, $\wedge$, and $\neg$. Since we have infinitely many variables, we cannot use symbols $x_i$ as symbols of the alphabet. We code a variable $x_j$ by $xbin(j)$, where $bin(j)$ is the shortest word[27] over $\Sigma_{bool}$ such that $Number(bin(j)) = j$, and $x$ is a symbol of $\Sigma_{logic}$. Then, the code of a formula $\Phi$ can be obtained by simply exchanging $x_i$ with $xbin(i)$ for every occurrence of $x_i$ in $\Phi$. For instance, the formula

[27] Thus, the first (most significant) bit of $bin(j)$ is 1.

$$\Phi = (x_1 \vee \overline{x}_4 \vee x_7) \wedge (x_2 \vee \overline{x}_1) \wedge (x_4 \wedge \overline{x}_8)$$

is represented by the word

$$w_\Phi = (x1 \vee \neg(x100) \vee x111) \wedge (x10 \vee \neg(x1)) \wedge (x100 \wedge \neg(x1000))$$

over $\Sigma_{logic} = \{0, 1, (, ), \wedge, \vee, \neg, x\}$.

**Definition 2.3.1.9.** *Let $\Sigma$ be an alphabet. Every set $L \subseteq \Sigma^*$ is called a* **language** *over $\Sigma$. The* **complement of the language $L$ according to $\Sigma$** *is $L^C = \Sigma^* - L$.*

*Let $\Sigma_1$ and $\Sigma_2$ be alphabets, and let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be languages. The* **concatenation** *of $L_1$ and $L_2$ is*

$$L_1 L_2 = L_1 \circ L_2 = \{uv \in (\Sigma_1 \cup \Sigma_2)^* \mid u \in L_1 \text{ and } v \in L_2\}.$$

$\emptyset$, $\{\lambda\}$, $\{a, b\}$, $\{a, b\}^*$, $\{ab, bba, b^{10}a^{20}\}$, $\{a^n b^{2^n} \mid n \in \mathbb{N}\}$ are examples of languages over $\{a, b\}$. Observe that $L \cdot \emptyset = \emptyset \cdot L = \emptyset$ and $L \cdot \{\lambda\} = \{\lambda\} \cdot L = L$ for every language $L$. $U = \{1\} \cdot \{0, 1\}^*$ is the language of binary representations of all positive integers.

A language can be used to describe a set of consistent input instances of an algorithmic problem. For instance, the set of all representations of formulae in CNF as a set of words over $\Sigma_{logic}$ or the set $\{u_1 \# u_2 \# \ldots \# u_m \mid u_i \in \{0, 1\}^m, m \in \mathbb{N}\}$ as the set of representations of all directed graphs over $\{0, 1, \#\}$ are examples of such languages. But words can also be used to code programs and so one can consider the language of codes of all correct programs in a given programming language. Languages can be also used to describe so-called decision problems, but this is the topic of the next section.

The last definition of this section shows how one can define a linear ordering on words over some alphabet $\Sigma$, provided one has a linear ordering on the symbols of $\Sigma$.

**Definition 2.3.1.10.** *Let $\Sigma = \{s_1, s_2, \ldots, s_m\}$, $m \geq 1$, be an alphabet, and let $s_1 < s_2 < \cdots < s_m$ be a linear ordering on $\Sigma$. We define the* **canonical ordering** *on $\Sigma^*$ as follows. For all $u, v \in \Sigma^*$,*

$$\begin{aligned} u < v \text{ if } & |u| < |v| \\ \text{or } & |u| = |v|, u = x s_i u', \text{ and } v = x s_j v' \\ & \text{for some } x, u', v' \in \Sigma^*, \text{ and } i < j. \end{aligned}$$

Keywords introduced in Section 2.3.1

alphabet, symbol, word, empty word, the length of a word, concatenation, prefix, suffix, subword, binary representation of integers, language, the complement of a language, canonical ordering of words

### 2.3.2 Algorithmic Problems

Thousands of algorithmic problems classified according to different points of view are considered in the literature on algorithmics. In this book we deal with hard problems only. We consider a problem to be hard if there is no known deterministic algorithm (computer program) that solves it efficiently. Efficiently means in a low-degree polynomial time. Our interpretation of hardness here is connected to the current state of our knowledge in algorithmics rather than to the unknown, real difficulty of the problems considered. Thus, a problem is hard if one would need years or thousands of years to solve it by deterministic programs for an input of a realistic size appearing in the current practice. This book provides a handbook of algorithmic methods that attack hard problems. Thousands of problems of great practical relevance are very hard from this point of view. Fortunately, we do not need to define and to consider all of them. There are some crucial, paradigmatic problems such as the traveling salesperson problem, linear (integer) programming, set cover problem, knapsack problem, satisfiability problem, and primality testing that are pattern problems in the sense that solving most of the hard problems can be reduced to solving some of the paradigmatic problems.

The goal of this chapter is to define some of these fundamental pattern problems. The methods for solving them are the topic of the next chapters. Every algorithm (computer program) can be viewed as an execution of a mapping from a subset of $\Sigma_1^*$ to $\Sigma_2^*$ for some alphabets $\Sigma_1$ and $\Sigma_2$. So, every (algorithmic) problem can be considered as a function from $\Sigma_1^*$ to $\Sigma_2^*$ or as a relation on $\Sigma_1^* \times \Sigma_2^*$ for some alphabets $\Sigma_1$ and $\Sigma_2$. We usually do not need to work with this kind of formalism because we consider two classes of problems only – decision problems (to decide for a given input whether it has a prescribed property) and optimization problems (to find the "best" solution from the set of solutions determined by some constraints). In what follows we define the fundamental problems that will be the objects of the algorithmic design in the subsequent chapters. We start with decision problems. If $A$ is an algorithm and $x$ is an input, then $\boldsymbol{A(x)}$ denotes the output of $A$ for the input $x$.

**Definition 2.3.2.1.** *A* **decision problem** *is a triple* $(L, U, \Sigma)$ *where* $\Sigma$ *is an alphabet and* $L \subseteq U \subseteq \Sigma^*$. *An algorithm* $A$ **solves** *(***decides***) the decision problem* $(L, U, \Sigma)$ *if, for every* $x \in U$,

*(i)* $A(x) = 1$ *if* $x \in L$, *and*
*(ii)* $A(x) = 0$ *if* $x \in U - L$ $(x \notin L)$.

We see that any algorithm $A$ solving a decision problem $(L, U, \Sigma)$ computes a function from $U$ to $\{0, 1\}$. The output "1" is interpreted as the answer "yes" to the question whether a given input belongs to $L$ (whether the input has the property corresponding to the specification of the language $L$), and the output "0" is equivalent to the answer "no".

An equivalent form of a description of a decision problem is the following form that specifies the input-output behavior.

**Problem $(L, U, \Sigma)$**

Input: An $x \in U$.
Output: "yes" if $x \in L$,
"no" otherwise.

For many decision problems $(L, U, \Sigma)$ we assume $U = \Sigma^*$. In that case we shall use the short notation $(L, \Sigma)$ instead of $(L, \Sigma^*, \Sigma)$.

Next we present the fundamental decision problems that will be studied in Chapter 5.

**PRIMALITY TESTING.**

Informally, primality testing is to decide, for a given positive integer, whether it is prime or not. Thus, primality testing is a decision problem $(\text{PRIM}, \Sigma_{bool})$, where

$$\text{PRIM} = \{w \in \{0,1\}^* \mid \mathit{Number}(w) \text{ is a prime}\}.$$

Another description of this problem is

**Primality testing**

Input: An $x \in \Sigma_{bool}^*$
Output: "yes" if $\mathit{Number}(x)$ is a prime,
"no" otherwise.

One can easily observe that primality testing can also be considered for other integer representations. Using $\Sigma_k = \{0, 1, 2, \ldots, k-1\}$ and the $k$-ary representation of integers we obtain $(\text{PRIM}_k, \Sigma_k)$, where

$$\text{PRIM}_k = \{x \in \Sigma_k^* \mid x \text{ is the } k\text{-ary representation of a prime}\}.$$

From the point of view of computational hardness it is not essential whether we consider $(\text{PRIM}, \Sigma_{bool})$ or $(\text{PRIM}_k, \Sigma_k)$ for some constant $k$ because one has efficient algorithms for transferring any $k$-ary representation of an integer to its binary representation, and vice versa. But this does not mean that the representation of integers does not matter for primality testing. If one represents an integer $n$ as

$$\#bin(p_1)\#bin(p_2)\#\ldots\#bin(p_l)$$

over $\{0, 1, \#\}$, where $n = p_1 \cdot p_2 \cdot \cdots \cdot p_l$ and $p_i$s are the nontrivial prime factors of $n$, then the problem of primality testing becomes easy. This sensibility of hardness of algorithmic problems according to the representation of their inputs is sometimes the reason for taking an exact formal description of the problem that fixes the data representation, too. For primality testing we always consider $(\text{PRIM}, \Sigma_{bool})$ as the formal definition of this decision problem.

**EQUIVALENCE PROBLEM FOR POLYNOMIALS.**

The problem is to decide, for a given prime $p$ and two polynomials $p_1(x_1, \ldots, x_m)$ and $p_2(x_1, \ldots, x_m)$ over the field $\mathbb{Z}_p$, whether $p_1$ and $p_2$ are equivalent, i.e., whether $p_1(x_1, \ldots, x_m) - p_2(x_1, \ldots, x_m)$ is identical 0. The crucial point is that the polynomials are not necessarily given in a normal form such as

$$a_0 + a_1x_1 + a_2x_2 + a_{12}x_1x_2 + a_1^2x_1^2 + a_2^2x_2^2 + \cdots$$

but in an arbitrary form such as

$$(x_1 + 3x_2)^2 \cdot (2x_1 + 4x_4) \cdot x_3^2.$$

A normal form may be exponentially long in the length of another representation and so the obvious way to compare two polynomials by transferring them to their normal forms and comparing their coefficients is not efficient.

We omit the formal definition of the representation of polynomials in an arbitrary form over the alphabet $\Sigma_{pol} = \{0, 1, (, ), \exp, +, \cdot\}$, because it can be done in a similar way as how one represents formulae over $\Sigma_{logic}$. The equivalence problem for polynomials can be defined as follows.

**EQ-POL**

Input: A prime $p$, two polynomials $p_1$ and $p_2$ over variables from $X = \{x_1, x_2, \ldots\}$.
Output: "yes" if $p_1 \equiv p_2$ in the field $\mathbb{Z}_p$,
"no" otherwise.

**EQUIVALENCE PROBLEM FOR ONE-TIME-ONLY BRANCHING PROGRAMS.**

The equivalence problem for one-time-only branching programs, EQ-1BP, is to decide, for two given one-time-only branching programs $B_1$ and $B_2$, whether $B_1$ and $B_2$ represent the same Boolean function. One can represent a branching program in a similar way as a directed weighted graph[28] and so we omit the formal description of branching program representation.[29]

**EQ-1BP**

Input: One-time-only branching program $B_1$ and $B_2$ over a set of Boolean variables $X = \{x_1, x_2, x_3, \ldots\}$.
Output: "yes" if $B_1$ and $B_2$ are equivalent (represent the same Boolean function),
"no" otherwise.

[28] Where not only the edges have some labels, but also the vertices are labeled.

[29] Remember that the formal definition of branching programs was given in Section 2.2.3 (Definitions 2.2.3.19, 2.2.3.20, Figure 2.11).

**SATISFIABILITY PROBLEM.**

The satisfiability problem is to decide, for a given formula in the CNF, whether it is satisfiable or not. Thus, the **satisfiability problem** is the decision problem $(\text{SAT}, \Sigma_{logic})$, where

$$\textbf{SAT} = \{w \in \Sigma^+_{logic} \mid w \text{ is a code of a satisfiable formula in CNF}\}.$$

We also consider specific subproblems of SAT where the length of clauses of the formulae in CNF is bounded. For every positive integer $k \geq 2$, we define the ***k*-satisfiability problem** as the decision problem $(k\text{SAT}, \Sigma_{logic})$, where

$$\boldsymbol{k}\textbf{SAT} = \{w \in \Sigma^+_{logic} \mid w \text{ is a code of a satisfiable formula in } k\text{CNF}\}.$$

In what follows we define some decidability problems from graph theory.

**CLIQUE PROBLEM.**

The clique problem is to decide, for a given graph $G$ and a positive integer $k$, whether $G$ contains a clique of size $k$ (i.e., whether the complete graph $K_k$ of $k$ vertices is a subgraph of $G$). Formally, the **clique problem** is the decision problem $(\text{CLIQUE}, \{0,1,\#\})$, where

$$\textbf{CLIQUE} = \{x\#w \in \{0,1,\#\}^* \mid x \in \{0,1\}^* \text{ and } w \text{ represents a graph that contains a clique of size } \mathit{Number}(x)\}.$$

An equivalent description of the clique problem is the following one.

**Clique Problem**

Input: A positive integer $k$ and a graph $G$.
Output: "yes" if $G$ contains a clique of size $k$,
"no" otherwise.

**VERTEX COVER PROBLEM.**

The vertex cover problem is to decide, for a given graph $G$ and a positive integer $k$, whether $G$ contains a vertex cover of cardinality $k$. Remember that a vertex cover of $G = (V, E)$ is any set $S$ of vertices of $G$ such that each edge from $E$ is incident to at least one vertex in $S$.

Formally, the **vertex cover problem** (**VCP**) is the decision problem $(\text{VCP}, \{0,1,\#\})$, where

$$\textbf{VCP} = \{u\#w \in \{0,1,\#\}^+ \mid u \in \{0,1\}^+ \text{ and } w \text{ represents a graph that contains a vertex cover of size } \mathit{Number}(u)\}.$$

**HAMILTONIAN CYCLE PROBLEM.**

The Hamiltonian cycle problem is to determine, for a given graph $G$, whether $G$ contains a Hamiltonian cycle or not. Remember that a Hamiltonian cycle of $G$ of $n$ vertices is a cycle of length $n$ in $G$ that contains every vertex of $G$.

Formally, the **Hamiltonian cycle problem** (**HC**) is the decision problem $(\text{HC}, \{0,1,\#\})$, where

$$\textbf{HC} = \{w \in \{0,1,\#\}^* \mid w \text{ represents a graph that contains a Hamiltonian cycle}\}.$$

**EXISTENCE PROBLEMS IN LINEAR PROGRAMMING.**

Here, we consider problems of deciding whether a given system of linear equations has a solution. Following the notation of Section 2.2.1 a system of linear equations is given by the equality

$$A \cdot X = b,$$

where $A = [a_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$ is an $m \times n$ matrix, $X = (x_1, x_2, \ldots, x_n)^{\mathsf{T}}$, and $b = (b_1, \ldots, b_m)^{\mathsf{T}}$ is an $m$-dimensional column vector. The $n$ elements $x_1, x_2, \ldots, x_n$ of $X$ are called unknowns (variables). In what follows we consider that all elements of $A$ and $b$ are integers. Remember, that

$$Sol(A,b) = \{X \subseteq \mathbb{R}^n \mid A \cdot X = b\}$$

denotes the set of all real-valued solutions of the linear equations system $A \cdot X = b$. In what follows we are interested in deciding whether $Sol(A,b)$ is empty or not (i.e., whether there exist a solution to $A \cdot X = b$) for given $A$ and $b$. More precisely, we consider several specific decision problems by restricting the set $Sol(A,b)$ to subsets of solutions over $\mathbb{Z}^n$ or $\{0,1\}^n$ only, or even considering the linear equations over some finite fields instead of $\mathbb{R}$. Let

$$\boldsymbol{Sol_S(A,b)} = \{X \subseteq S^n \mid A \cdot X = b\}$$

for any subset $S$ of $\mathbb{R}$.

First of all observe that the problem of deciding whether $Sol(A,b) = \emptyset$ is one of the fundamental tasks of linear algebra and that it can be solved efficiently. The situation essentially changes if one searches for integer solutions or Boolean solutions. Let $\langle A,b \rangle$ denote a representation of a matrix $A$ and a vector $b$ over the alphabet $\{0,1,\#\}$, assuming all elements of $A$ and $b$ are integers.

The **problem of the existence of a solution of linear integer programming** is to decide whether $Sol_{\mathbb{Z}}(A,b) = \emptyset$ for given $A$ and $b$. Formally, this decision problem is (SOL-IP, $\{0,1,\#\}$), where

$$\textbf{SOL-IP} = \{\langle A,b \rangle \in \{0,1,\#\}^* \mid Sol_{\mathbb{Z}}(A,b) \neq \emptyset\}.$$

The **problem of the existence of a solution of 0/1-linear programming** is to decide whether $Sol_{\{0,1\}}(A,b) = \emptyset$ for given $A$ and $b$. Formally, this decision problem is (SOL-0/1-IP, $\{0,1,\#\}$), where

$$\textbf{SOL-0/1-IP} = \{\langle A,b\rangle \in \{0,1,\#\}^* \mid Sol_{\{0,1\}}(A,b) \neq \emptyset\}.$$

All existence problems mentioned above consider computing over the field $\mathbb{R}$. We are interested in solving the system of linear equations $A \cdot X = b$ over a finite field $\mathbb{Z}_p$ for a prime $p$. So, all elements of $A$ and $b$ are from $\mathbb{Z}_p = \{0,1,\ldots,p-1\}$, all solutions have to be from $(\mathbb{Z}_p)^n$, and the linear equations are congruences modulo $p$ (i.e., the operation of addition is $\oplus_{\bmod p}$ and the operation of multiplication is $\odot_{\bmod p}$).

The **problem of the existence of a solution of linear programming modulo $p$** is the decision problem (SOL-IP$_p$, $\{0,1,\ldots,p-1,\#\}$) where

$$\begin{aligned}\textbf{SOL-IP}_p = \{\langle A,b\rangle \in \{0,1,\ldots,p-1,\#\}^* \mid &\text{ if } A \text{ is an } m \times n \text{ matrix}\\ &\text{over } \mathbb{Z}_p, m,n \in \mathbb{N}-\{0\}, \text{ and } b \in \mathbb{Z}_p^m, \text{ then there}\\ &\text{exists } X \in (\mathbb{Z}_p)^n \text{ such that } AX \equiv b \pmod p\}.\end{aligned}$$

In what follows, we define some fundamental optimization problems. We start with a general framework that describes the formalism for the specification of optimization problems.

Roughly, a problem instance $x$ of an optimization problem specifies a set of constraints. These constraints unambiguously determine the set $\mathcal{M}(x)$ of feasible solutions for the problem instance $x$. Note that $\mathcal{M}(x)$ may be empty or infinite. The objective, determined by the specification of the problem, is to find a solution from $\mathcal{M}(x)$ that is the "best" one among all solutions in $\mathcal{M}(x)$. Note that there may exist several (even infinitely many) best solutions among the solutions in $\mathcal{M}(x)$.

**Definition 2.3.2.2.** *An* **optimization problem** *is a 7-tuple* $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$, *where*

*(i)* $\Sigma_I$ *is an alphabet, called the* **input alphabet** *of* $U$,
*(ii)* $\Sigma_O$ *is an alphabet, called the* **output alphabet** *of* $U$,
*(iii)* $L \subseteq \Sigma_I^*$ *is the* **language of feasible problem instances**,
*(iv)* $L_I \subseteq L$ *is the* **language of the (actual) problem instances of** $U$,
*(v)* $\mathcal{M}$ *is a function from* $L$ *to* $Pot(\Sigma_O^*)$,[30] *and, for every* $x \in L$, $\mathcal{M}(x)$ *is called the* **set of feasible solutions** *for* $x$,
*(vi) cost is the* **cost function** *that, for every pair* $(u,x)$, *where* $u \in \mathcal{M}(x)$ *for some* $x \in L$, *assigns a positive real number* $cost(u,x)$,
*(vii)* $goal \in \{minimum, maximum\}$.

[30] Remember that $Pot(S)$ is the set of all subsets of the set $S$, i.e., the power set of $S$.

*For every $x \in L_I$, a feasible solution $y \in \mathcal{M}(x)$ is called* **optimal for $x$ and $U$** *if*

$$cost(y, x) = goal\{cost(z, x) \mid z \in \mathcal{M}(x)\}.$$

*For an optimal solution $y \in \mathcal{M}(x)$, we denote $cost(y, x)$ by $\boldsymbol{Opt_U(x)}$. $U$ is called a* **maximization problem** *if goal = maximum, and $U$ is a* **minimization problem** *if goal = minimum. In what follows $\boldsymbol{Output_U(x)} \subseteq \mathcal{M}(x)$ denotes the set of all optimal solutions for the instance $x$ of $U$.*

*An algorithm $A$ is* **consistent** *for $U$ if, for every $x \in L_I$, the output $A(x) \in \mathcal{M}(x)$. We say that an algorithm $B$* **solves** *the optimization problem $U$ if*

*(i) $B$ is consistent for $U$, and*
*(ii) for every $x \in L_I$, $B(x)$ is an optimal solution for $x$ and $U$.*

Let us explain the informal meaning of the formal definition of an optimization problem $U$ as a 7-tuple $(\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$. $\Sigma_I$ has the same meaning as the alphabet of decision problems and it is used to code (represent) the inputs. Similarly, $\Sigma_O$ is the alphabet used to code outputs. On the level of algorithm design used here we usually do not need to specify $\Sigma_I$ or $\Sigma_O$ and the coding of inputs and outputs, because these details do not have any essential influence on the hardness of the problems considered. But this formal specification may be useful in the classification of the optimization problems according to their computational difficulty, and especially for proving lower bounds on their polynomial-time approximability.

The language $L$ is the set of codes of all problem instances (inputs) for which $U$ is well defined. $L_I$ is the set of actual problem instances (inputs) and one measures the computational hardness of $U$ according to inputs of $L_I$. In general, one can simplify the definition of $U$ by removing $L$ and the definition will work as well as Definition 2.3.2.2. The reason to put this additional information into the definition of optimization problems is that the hardness of many optimization problems is very sensible according to the specification of the set of considered problem instances ($L_I$). Definition 2.3.2.2 enables one to conveniently measure the increase or decrease of the hardness of optimization problems according to the changes of $L_I$ by a fixed $L$.

**Definition 2.3.2.3.** *Let $U_1 = (\Sigma_I, \Sigma_O, L, L_{I,1}, \mathcal{M}, cost, goal)$ and $U_2 = (\Sigma_I, \Sigma_O, L, L_{I,2}, \mathcal{M}, cost, goal)$ be two optimization problems. We say that $U_1$ is a* **subproblem** *of $U_2$ if $L_{I,1} \subseteq L_{I,2}$.*

The function $\mathcal{M}$ is determined by the constraints given by the problem instances and $\mathcal{M}(x)$ is the set of all objects (solutions) satisfying the constraints given by $x$. The cost function assigns the cost $cost(\alpha, x)$ to every solution $\alpha$ from $\mathcal{M}(x)$. If the input instance $x$ is fixed, we often use the short notion $\boldsymbol{cost(\alpha)}$ instead of $cost(\alpha, x)$. If *goal = minimum* [= *maximum*], then an optimal solution is any solution from $\mathcal{M}(x)$ with the minimal [maximal] cost.

To make the definitions of specific optimization problems transparent, we often leave out the specification of coding the data over $\Sigma_I$ and $\Sigma_O$. We define the problems simply by specifying

- the set of actual problem instances $L_I$,
- the constraints given by the input instances, and so $\mathcal{M}(x)$ for every $x \in L_I$,
- the cost function,
- the goal.

**TRAVELING SALESPERSON PROBLEM.**

Traveling salesperson problem is the problem of finding a Hamiltonian cycle (tour) of the minimal cost in a complete weighted graph. The formal definition follows.

**Traveling Salesperson Problem (TSP)**

Input: A weighted complete graph $(G, c)$, where $G = (V, E)$ and $c : E \to \mathbb{N}$. Let $V = \{v_1, \ldots, v_n\}$ for some $n \in \mathbb{N} - \{0\}$.

Constraints: For every input instance $(G, c)$, $\mathcal{M}(G, c) = \{v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1} \mid (i_1, i_2, \ldots, i_n)$ is a permutation of $(1, 2, \ldots, n)\}$, i.e., the set of all Hamiltonian cycles of $G$.

Costs: For every Hamiltonian cycle $H = v_{i_1} v_{i_2} \ldots v_{i_n} v_{i_1} \in \mathcal{M}(G, c)$, $cost((v_{i_1}, v_{i_2}, \ldots v_{i_n}, v_{i_1}), (G, c)) = \sum_{j=1}^{n} c(\{v_{i_j}, v_{i_{(j \bmod n)+1}}\})$, i.e., the cost of every Hamiltonian cycle $H$ is the sum of the weights of all edges of $H$.

Goal: *minimum*.

If one wants to specify $\Sigma_I$ and $\Sigma_O$ one can take $\{0, 1, \#\}$ for both. The input can be a code of the adjacency matrix of $(G, c)$ and the Hamiltonian paths can be coded as permutations of the set of vertices.

The following adjacency matrix represents the problem instance of TSP depicted in Figure 2.12.

$$\begin{pmatrix} 0 & 1 & 1 & 3 & 8 \\ 1 & 0 & 2 & 1 & 2 \\ 1 & 2 & 0 & 7 & 1 \\ 3 & 1 & 7 & 0 & 1 \\ 8 & 2 & 1 & 1 & 0 \end{pmatrix}$$

Observe that there are $4!/2 = 12$ Hamiltonian tours in $K_5$. The cost of the Hamiltonian tour $H = v_1, v_2, v_3, v_4, v_5, v_1$ is

$$\begin{aligned} cost(H) &= c(\{v_1, v_2\}) + c(\{v_2, v_3\}) + c(\{v_3, v_4\}) + c(\{v_4, v_5\}) + c(\{v_5, v_1\}) \\ &= 1 + 2 + 7 + 1 + 8 = 19. \end{aligned}$$

The unique optimal Hamiltonian tour is

$$H_{Opt} = v_1, v_2, v_4, v_5, v_3, v_1 \text{ with } cost(H_{Opt}) = 5.$$

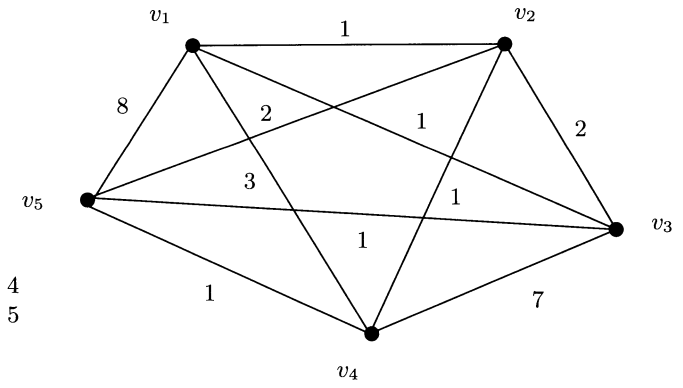



**Fig. 2.12.**

Now we define two subproblems of TSP.

The **metric traveling salesperson problem**, **△-TSP**, is a subproblem of TSP such that every problem instance $(G, c)$ of △-TSP satisfies the triangle inequality

$$c(\{u, v\}) \leq c(\{u, w\}) + c(\{w, v\})$$

for all vertices $u, w, v$ of $G$.

The problem instance depicted in Figure 2.12 does not satisfy the triangle inequality because

$$7 = c(\{v_3, v_4\}) > c(\{v_3, v_5\}) + c(\{v_5, v_4\}) = 1 + 1 = 2.$$

The **geometrical traveling salesperson problem** (**Euclidean TSP**) is a subproblem of TSP such that, for every problem instance $(G, c)$ of TSP, the vertices of $G$ can be embedded in the two-dimensional Euclidean space in such a way that $c(\{u, v\})$ is the Euclidean distance between the points assigned to the vertices $u$ and $v$ for all $u, v$ of $G$. A simplified specification of the set of input instances of the geometric TSP is to say that the input is a set of points in the plane and the cost of the connection between any two points is defined by their Euclidean distance.

Since the two-dimensional Euclidean space is a metric space, the Euclidean distance satisfies the triangle inequality and so the geometrical TSP is a subproblem of △-TSP.

### MAKESPAN SCHEDULING PROBLEM.

The problem of makespan scheduling (MS) is to schedule $n$ jobs with designated processing times on $m$ identical machines in such a way that the whole processing time is minimized. Formally, we define MS as follows.

**Makespan Scheduling Problem (MS)**

Input: Positive integers $p_1, p_2, \ldots, p_n$ and an integer $m \geq 2$ for some $n \in \mathbb{N} - \{0\}$.
{$p_i$ is the processing time of the $i$th job on any of the $m$ available machines}.

Constraints: For every input instance $(p_1, \ldots, p_n, m)$ of MS,
$\mathcal{M}(p_1, \ldots, p_n, m) = \{S_1, S_2, \ldots, S_m \mid S_i \subseteq \{1, 2, \ldots, n\}$ for $i = 1, \ldots, m$, $\bigcup_{k=1}^{m} S_k = \{1, 2, \ldots, n\}$, and $S_i \cap S_j = \emptyset$ for $i \neq j\}$.
{$\mathcal{M}(p_1, \ldots, p_n, m)$ contains all partitions of $\{1, 2, \ldots, n\}$ into $m$ subsets. The meaning of $(S_1, S_2, \ldots, S_m)$ is that, for $i = 1, \ldots, m$, the jobs with indices from $S_i$ have to be processed on the $i$th machine}.

Costs: For each $(S_1, S_2, \ldots, S_m) \in \mathcal{M}(p_1, \ldots, p_n, m)$,
$cost((S_1, \ldots, S_m), (p_1, \ldots, p_n, m)) = \max\{\sum_{l \in S_i} p_l \mid i = 1, \ldots, m\}$.

Goal: *minimum*.

An example of scheduling seven jobs with the processing times 3, 2, 4, 1, 3, 3, 6, respectively, on 4 machines is given in Figure 4.1 in Section 4.2.1.

**COVER PROBLEMS.**

Here, we define the minimum vertex cover problem[31] (MIN-VCP), its weighted version, and the set cover problem (SCP). The minimum vertex cover problem is to cover all edges of a given graph $G$ with a minimal number of vertices of $G$.

**Minimum Vertex Cover Problem (MIN-VCP)**

Input: A graph $G = (V, E)$.

Constraints: $\mathcal{M}(G) = \{S \subseteq V \mid$ every edge of $E$ is incident to at least one vertex of $S\}$.

Cost: For every $S \in \mathcal{M}(G)$, $cost(S, G) = |S|$.

Goal: *minimum*.

Consider the graph $G$ given in Figure 2.13.

$$\mathcal{M}(G) = \{\{v_1, v_2, v_3, v_4, v_5\}, \{v_1, v_2, v_3, v_4\}, \{v_1, v_2, v_3, v_5\}, \{v_1, v_2, v_4, v_5\}$$
$$\{v_1, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_1, v_3, v_4\}, \{v_2, v_4, v_5\}, \{v_2, v_3, v_5\}\}.$$

The optimal solutions are $\{v_1, v_3, v_4\}$, $\{v_2, v_4, v_5\}$, and $\{v_2, v_3, v_5\}$ and so $Opt_{\mathrm{VCP}}(G) = 3$. Observe that there is no vertex cover of cardinality 2 because

[31] Observe that we have two versions of vertex cover problems. One version is the decision problem defined by the language VCP above and the second version MIN-VCP is the minimization problem considered here.

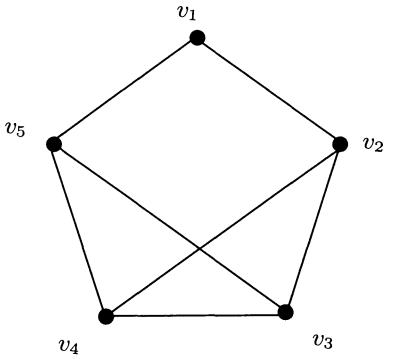



**Fig. 2.13.**

to cover the edges of the cycle $v_1, v_2, v_3, v_4, v_5, v_1$ one needs at least three vertices.

**Set Cover Problem (SCP)**

Input: $(X, \mathcal{F})$, where $X$ is a finite set and $\mathcal{F} \subseteq Pot(X)$ such that $X = \bigcup_{S \in \mathcal{F}} S$.
Constraints: For every input $(X, \mathcal{F})$,
$\mathcal{M}(X, \mathcal{F}) = \{C \subseteq \mathcal{F} \mid X = \bigcup_{S \in C} S\}$.
Costs: For every $C \in \mathcal{M}(X, \mathcal{F})$, $cost(C, (X, \mathcal{F})) = |C|$.
Goal: *minimum*.

Later we shall observe that MIN-VCP can be viewed as a special subproblem of SCP because, for a given graph $G = (V, E)$, one can assign the set $S_v$ of all edges adjacent to $v$ to every vertex $v$ of $G$. For the graph in Figure 2.13 it results in the instance $(E, \mathcal{F})$ of SCP where

$\mathcal{F} = \{S_{v_1}, S_{v_2}, S_{v_3}, S_{v_4}, S_{v_5}\}$,
$S_{v_1} = \{\{v_1, v_2\}, \{v_1, v_5\}\}$, $S_{v_2} = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_2, v_4\}\}$,
$S_{v_3} = \{\{v_3, v_2\}, \{v_3, v_5\}, \{v_3, v_4\}\}$, $S_{v_4} = \{\{v_3, v_4\}, \{v_2, v_4\}, \{v_4, v_5\}\}$, and
$S_{v_5} = \{\{v_1, v_5\}, \{v_3, v_5\}, \{v_4, v_5\}\}$.

The last cover problem that we consider is the weighted generalization of MIN-VCP.

**Weighted Minimum Vertex Cover Problem (WEIGHT-VCP)**

Input: A weighted graph $G = (V, E, c)$, $c : V \to \mathbb{N} - \{0\}$.
Constraints: For every input instance $G = (V, E, c)$,
$\mathcal{M}(G) = \{S \subseteq V \mid S \text{ is a vertex cover of } G\}$.
Cost: For every $S \in \mathcal{M}(G)$, $G = (V, E, c)$,
$cost(S, (V, E, c)) = \sum_{v \in S} c(v)$.
Goal: *minimum*.

### MAXIMUM CLIQUE PROBLEM.

The maximum clique problem (MAX-CL) is to find a clique of the maximal size in a given graph $G$.

**Maximum Clique Problem (MAX-CL)**

Input: A graph $G = (V, E)$
Constraints: $\mathcal{M}(G) = \{S \subseteq V \mid \{\{u, v\} \mid u, v \in S, u \neq v\} \subseteq E\}$.
$\{\mathcal{M}(G)$ contains all complete subgraphs (cliques) of $G\}$
Costs: For every $S \in \mathcal{M}(G)$, $cost(S, G) = |S|$.
Goal: *maximum*.

To present a specific input instance consider the graph $G$ depicted in Figure 2.13.

$$\mathcal{M}(G) = \{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\},$$
$$\{v_1, v_2\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\},$$
$$\{v_2, v_3, v_4\}, \{v_3, v_4, v_5\}\}.$$

The optimal solutions are $\{v_2, v_3, v_4\}$ and $\{v_3, v_4, v_5\}$ and so $Opt_{\text{MAX-CL}}(G) = 3$.

### CUT PROBLEMS.

We introduce the maximum cut problem (MAX-CUT) and the minimum cut problem (MIN-CUT). Remember that a cut of a graph $G = (V, E)$ is any partition of $V$ into $(V_1, V_2)$ such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$.

**Maximum Cut Problem (MAX-CUT)**

Input: A graph $G = (V, E)$.
Constraints:
$$\mathcal{M}(G) = \{(V_1, V_2) \mid V_1 \cup V_2 = V, V_1 \neq \emptyset \neq V_2, \text{and} V_1 \cap V_2 = \emptyset\}.$$
Costs: For every cut $(V_1, V_2) \in \mathcal{M}(G)$,

$$cost((V_1, V_2), G) = |E \cap \{\{u, v\} \mid u \in V_1, v \in V_2\}|.$$

Goal: *maximum*.

The **minimum cut problem (MIN-CUT)** can be defined in the same way as MAX-CUT. The only difference is that the goal of MIN-CUT is minimum.

The only optimal solution of **MIN-CUT** for the graph $G$ in Figure 2.13 is $(\{v_1\}, \{v_2, v_3, v_4, v_5\})$, and the optimal solutions of MAX-CUT for the graph $G$ are $(\{v_1, v_2, v_3\}, \{v_4, v_5\})$, $(\{v_1, v_2, v_5\}, \{v_3, v_4\})$, and $(\{v_1, v_4, v_5\}, \{v_2, v_3\})$. So, $Opt_{\text{MIN-CUT}}(G) = 2$ and $Opt_{\text{MAX-CUT}}(G) = 4$.

**KNAPSACK PROBLEM.**

First, we define the simple knapsack problem(SKP). This optimization task can be described as follows. One has a knapsack whose weight capacity is bounded by a positive integer $b$ (for instance, by $b$ pounds) and $n$ objects of weights $w_1, w_2, \ldots, w_n$, $n \in \mathbb{N} - \{0\}$. The aim is to pack some objects in the knapsack in such a way that the contents of the knapsack are as heavy as possible but not above $b$.

**Simple Knapsack Problem (SKP)**

Input: A positive integer $b$, and positive integers $w_1, w_2, \ldots, w_n$ for some $n \in \mathbb{N} - \{0\}$.

Constraints: $\mathcal{M}(b, w_1, w_2, \ldots, w_n) = \{T \subseteq \{1, \ldots, n\} \mid \sum_{i \in T} w_i \leq b\}$, i.e., a feasible solution for the problem instance $b, w_1, w_2, \ldots, w_n$ is every set of objects whose common weight does not exceed $b$.

Costs: For each $T \in \mathcal{M}(b, w_1, w_2, \ldots, w_n)$,

$$cost(T, b, w_1, w_2, \ldots, w_n) = \sum_{i \in T} w_i.$$

Goal: *maximum*.

For the problem instance $I = (b, w_1, \ldots, w_5)$, where $b = 29$, $w_1 = 3$, $w_2 = 6$, $w_3 = 8$, $w_4 = 7$, $w_5 = 12$, the only optimal solution is $T = \{1, 2, 3, 5\}$ with $cost(T, I) = 29$. If one considers the problem instance $I' = (b', w_1, \ldots, w_5)$ with $b' = 14$, then the optimal solution is $T' = \{2, 3\}$ with $cost(T', I') = 14$.

The instances of the general knapsack problem contain additionally a cost $c_i$ for every object $i$. The objective is to maximize the common cost of objects packed into the knapsack[32] by satisfying the constraint $b$ on the weight of the knapsack.

**Knapsack Problem (KP)**

Input: A positive integer $b$, and $2n$ positive integers $w_1, w_2, \ldots, w_n$, $c_1$, $c_2, \ldots, c_n$ for some $n \in \mathbb{N} - \{0\}$.

Constraints:
$\mathcal{M}(b, w_1, \ldots, w_n, c_1, \ldots, c_n) = \{T \subseteq \{1, \ldots, n\} \mid \sum_{i \in T} w_i \leq b\}$.

Costs: For each $T \in \mathcal{M}(b, w_1, \ldots, w_n, c_1, \ldots, c_n)$,

$$cost(T, b, w_1, \ldots, w_n, c_1, \ldots, c_n) = \sum_{i \in T} c_i.$$

Goal: *maximum*.

Consider the problem instance $I$ determined by $b = 59$, $w_1 = 12$, $c_1 = 9$, $w_2 = 5$, $c_2 = 4$, $w_3 = 13$, $c_3 = 5$, $w_4 = 18$, $c_4 = 9$, $w_5 = 15$, $c_5 = 9$, $w_6 = 29$, $c_6 = 22$. The optimal solution is $T = \{1, 5, 6\}$. Observe that $T$ satisfies the constraint because $w_1 + w_5 + w_6 = 12 + 15 + 29 = 56 < 59 = b$ and that $Opt_{KP}(I) = c_1 + c_5 + c_6 = 9 + 9 + 22 = 40$.

[32] Rather than their weights.

**BIN-PACKING PROBLEM.**

The bin-packing problem (Bin-P) is similar to the knapsack problem. One has $n$ objects of rational weights $w_1, \ldots, w_n \in [0,1]$. The goal is to distribute them among the knapsacks (bins) of unit size 1 in such a way that a minimal number of knapsacks (bins) is used.

**Bin-Packing Problem (BIN-P)**

Input: $n$ rational numbers $w_1, w_2, \ldots, w_n \in [0,1]$ for some positive integer $n$.

Constraints: $\mathcal{M}(w_1, w_2, \ldots, w_n) = \{S \subseteq \{0,1\}^n \mid$ for every $s \in S$, $s^{\mathsf{T}} \cdot (w_1, w_2, \ldots, w_n) \leq 1$, and $\sum_{s \in S} s = (1,1,\ldots,1)\}$.
{If $S = \{s_1, s_2, \ldots, s_m\}$, then $s_i = (s_{i1}, s_{i2}, \ldots, s_{in})$ determines the set of objects packed in the $i$th bin. The $j$th object is packed into the $i$th bin if and only if $s_{ij} = 1$. The constraint

$$s_i^{\mathsf{T}} \cdot (w_1, \ldots, w_n) \leq 1$$

assures that the $i$th bin is not overfilled. The constraint

$$\sum_{s \in S} s = (1,1,\ldots,1)$$

assures that every object is packed in exactly one bin.}

Cost: For every $S \in \mathcal{M}(w_1, w_2, \ldots, w_n)$,

$$cost(S, (w_1, \ldots, w_n)) = |S|.$$

Goal: *minimum*.

Observe that an alternative way to describe the constraints of BIN-P is to take

$$\begin{aligned}\mathcal{M}(w_1, \ldots, w_n) = \{(T_1, T_2, \ldots, T_m) \mid & m \in \mathbb{N} - \{0\}, T_i \subseteq \{1,2,\ldots,n\} \\ & \text{for } i = 1, \ldots, n, T_i \cap T_j = \emptyset \text{ for } i \neq j, \\ & \bigcup_{i=1}^{m} T_i = \{1,2,\ldots,n\}, \text{ and} \\ & \sum_{k \in T_j} w_k \leq 1 \text{ for } j = 1, \ldots, m\}.\end{aligned}$$

**MAXIMUM SATISFIABILITY PROBLEM.**

The general maximum satisfiability problem (MAX-SAT) is to find an assignment to the variables of a formula $\Phi$ such that the number of satisfied clauses is maximized.

**Maximum Satisfiability Problem (MAX-SAT)**

Input: A formula $\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ over $X = \{x_1, x_2, \ldots\}$ in CNF (an equivalent description of this instance of MAX-SAT is to consider the set of clauses $F_1, F_2, \ldots, F_m$).

Constraints: For every formula $\Phi$ over the set $\{x_1, \ldots, x_n\} \subseteq X$, $n \in \mathbb{N} - \{0\}$, $\mathcal{M}(\Phi) = \{0,1\}^n$.
{Every assignment of values to $\{x_1, \ldots, x_n\}$ is a feasible solution, i.e., $\mathcal{M}(\Phi)$ can also be written as $\{\alpha \mid \alpha : X \rightarrow \{0,1\}\}$.

Costs: For every $\Phi$ in CNF, and every $\alpha \in \mathcal{M}(\Phi)$, $cost(\alpha, \Phi)$ is the number of clauses satisfied by $\alpha$.

Goal: *maximum*.

Observe that, if $\Phi$ is a satisfiable formula, an optimal solution is any assignment $\alpha$ that satisfies $\Phi$ (i.e., $cost(\alpha, \Phi) = m$ if $\Phi$ consists of $m$ clauses).

We consider several subproblems of MAX-SAT here. For every integer $k \geq 2$, we define the **MAX-$k$SAT** problem as a subproblem of MAX-SAT, where the problem instances are formulae in $k$CNF[33]. For every integer $k \geq 2$, we define the **MAX-E$k$SAT** as a subproblem of MAX-$k$SAT, where the inputs are formulae consisting of clauses of the size $k$ only. Each clause $l_1 \vee_2 \vee \cdots \vee l_k$ of such a formula is a Boolean function over exactly $k$ variables, i.e., $l_i \neq l_j$ and $l_i \neq \bar{l}_j$ for all $i, j \in \{1, \ldots, k\}$, $i \neq j$.

**LINEAR PROGRAMMING.**

First, we define the general version of the linear programming problem and then we consider some special versions of it.

**Linear Programming (LP)**

Input: A matrix $A = [a_{ij}]_{i=1,\ldots,m, j=1,\ldots,n}$, a vector $b \in \mathbb{R}^m$, and a vector $c \in \mathbb{R}^n$, $n, m \in \mathbb{N} - \{0\}$.

Constraints: $\mathcal{M}(A, b, c) = \{X \in \mathbb{R}^n \mid A \cdot X = b$ and the elements of $X$ are non-negative reals only$\}$.

Costs: For every $X = (x_1, \ldots, x_n) \in \mathcal{M}(A, b, c)$, $c = (c_1, \ldots, c_n)^{\mathsf{T}}$, $cost(X, (A, b, c)) = c^{\mathsf{T}} \cdot X = \sum_{i=1}^{n} c_i x_i$.

Goal: *minimum*.

We observe that the set of constraints $A \cdot X = b$ and $x_j \geq 0$ for $i = 1, \ldots, n$ is a system of $m+n$ linear equations of $n$ unknowns. Since every linear equation determines an $(n-1)$-dimensional affine subspace of the vector space $\mathbb{R}^n$, $\mathcal{M}(A, b, c)$ can be considered as the intersection of the $m+n$ affine subspaces determined by the linear equations[34] of $A \cdot X = b$ and $X \in (\mathbb{R}^{\geq 0})^n$.

[33] That is, the size of each clause of $\Phi$ is at most $k$.

[34] Note that there exist several other forms of the linear programming problem. For instance, one can exchange the constraint $A \cdot X = b$ with $A \cdot X \geq b$. Or one can take maximization instead of minimization and consider the constraint $A \cdot X \leq b$ instead of $A \cdot X = b$.

In combinatorial optimization we often consider the problem of integer programming. It can be defined by exchanging reals with integers in the problem instances as well as in the feasible solutions.

**Integer Linear Programming (IP)**

Input: An $m \times n$ matrix $A = [a_{ij}]_{i=1,\ldots,m,j=1,\ldots,n}$, and two vectors $b = (b_1, \ldots, b_m)^T$, $c = (c_1, \ldots, c_n)^T$ for some $n, m \in \mathbb{N} - \{0\}$, $a_{ij}, b_i, c_j$ are integers for $i = 1, \ldots, m$, $j = 1, \ldots, n$.

Constraints: $\mathcal{M}(A, b, c) = \{X = (x_1, \ldots, x_n) \in \mathbb{Z}^n \mid AX = b \text{ and } x_i \geq 0 \text{ for } i = 1, \ldots, n\}$.

Costs: For every $X = (x_1, \ldots, x_n) \in \mathcal{M}(A, b, c)$, $cost(X, (A, b, c)) = \sum_{i=1}^{n} c_i x_i$.

Goal: *minimum*.

Note that IP is not a subproblem of LP, because we did not restrict the language of inputs only, but also the constraints.

The **0/1-Linear Programming (0/1-LP)** is the optimization problem with the language of input instances of IP and the additional constraints requiring that $X \in \{0,1\}^n$ (i.e., that $\mathcal{M}(A, b, c) \subseteq \{0,1\}^n$).

The last problems we consider are maximization problems on systems of linear equations. The objective is to find values for unknowns that satisfy the maximal possible number of linear equations of the given system. Let $k \geq 2$ be prime.

**Maximum Linear Equation Problem Mod $k$ (MAX-LINMOD$k$)**

Input: A set $S$ of $m$ linear equations over $n$ unknowns, $n, m \in \mathbb{N} - \{0\}$, with coefficients from $\mathbb{Z}_k$.
(An alternative description of an input is an $m \times n$ matrix over $\mathbb{Z}_k$ and a vector $b \in \mathbb{Z}_k^m$).

Constraints: $\mathcal{M}(S) = \mathbb{Z}_k^m$
{a feasible solution is any assignment of values from $\{0, 1, \ldots, k-1\}$ to the $n$ unknowns (variables)}.

Costs: For every $X \in \mathcal{M}(S)$, $cost(X, S)$ is the number of linear equations of $S$ satisfied by $X$.

Goal: *maximum*.

Consider the following example of an input instance over $\mathbb{Z}_2$:

$$\begin{aligned} x_1 + x_2 &= 1 \\ x_1 + x_3 &= 0 \\ x_2 + x_3 &= 0 \\ x_1 + x_2 + x_3 &= 1. \end{aligned}$$

Observe that this system of linear equations does not have any solution in $\mathbb{Z}_2$. The assignment $x_1 = x_2 = x_3 = 0$ satisfies the second equation and the third equation. The assignment $x_1 = x_2 = x_3 = 1$ satisfies the last

three equations and the assignment $x_1 = x_3 = 0$, $x_2 = 1$ satisfies the first two equations and the last equation. The last two assignments are optimal solutions.

For every prime $k$, and every positive integer $m$, we define the problem **MAX-E$m$LINMOD$k$** as the subproblem of MAX-LINMOD$k$, where the input instances are sets of linear equations such that every linear equation has at most $m$ nonzero coefficients (contains at most $m$ unknowns).

For instance, the first three linear equations of the system considered above form a problem instance of MAX-E2LINMOD2.

### Keywords introduced in Section 2.3.2

decision problem, primality testing, equivalence problem for polynomials, equivalence problem for one-time-only branching programs, satisfiability problems, clique problem, vertex cover problem, Hamiltonian problem, problems of existence of a solution of systems of linear equations, optimization problem, problem instance, feasible solution, optimal solution, maximization problem, minimization problem, traveling salesperson problem (TSP), metric TSP ($\triangle$-TSP), geometrical TSP, makespan problem (MS), minimum vertex cover problem (MIN-VCP), set cover problem (SCP), maximum clique problem (MAX-CL), minimum cut problem (MIN-CUT), maximum cut problem (MAX-CUT), knapsack problem (KP), simple knapsack problem (SKP), bin-packing problem (BIN-P), maximum satisfiability problem (MAX-SAT), linear programming (LP), integer programming (IP), 0/1-linear programming (0/1-LP), maximum linear equation problem modulo $k$ (MAX-LINMOD$k$)

Summary of Section 2.3.2

A decision problem is a problem of deciding whether a given input has a required property. Since the set of all inputs with the required property can be viewed as a language $L \subseteq \Sigma^*$, the decision problem for $L$ is to decide whether a given input $x \in \Sigma^*$ belongs to $L$ or $x \notin L$.

An optimization problem is specified by

- the set of problem instances,
- the constraints determining the set of feasible solutions to every problem instance (input),
- the cost function that assigns a cost to every feasible solution,
- the goal that may be maximization or minimization.

The objective is to find an optimal solution (one of the best feasible solutions according to the cost and the goal) for every input instance.

Currently, there are thousands of hard algorithmic problems considered in the literature on algorithmics and in numerous practical applications. To learn the fundamentals of algorithmics it is sufficient to consider some of them – so-called paradigmatic problems. Paradigmatic problems are some kind of pattern problems in the

sense that solving most of the hard problems can be reduced to solving some of the paradigmatic problems. Some of the most fundamental decision problems are satisfiability, primality testing, equivalence problem for polynomials, clique problem, and Hamiltonian problems. Some representatives of the paradigmatic optimization problems are maximum satisfiability, traveling salesperson problem, scheduling problems, set cover problem, maximum clique problem, knapsack problem, bin-packing problem, linear programming, integer programming, and maximum linear equation problems.

### 2.3.3 Complexity Theory

The aim of this section is to discuss the ways of measuring computational complexity of algorithms (computer programs) and to present the main framework for classifying algorithmic problems according to their computational hardness. The first part of this section is useful for all subsequent chapters. The second part, devoted to the complexity classes and to the concept of NP-completeness, provides (besides the general philosophy of algorithms and complexity) fundamentals for Section 4.4.3 on lower bounds for inapproximability (i.e., for the classification of optimization problems according to their polynomial-time approximability).

This book is devoted to the design of programs on the algorithmic level and so we will not deal with details of algorithm implementations in specific programming languages. To describe algorithms we use either an informal description of its parts such as "choose an edge from the graph $G$ and verify whether the rest of the graph is connected" or a Pascal-like language with instructions such as for, repeat, while, if ... then ... else, etc. Observe that this rough description can be sufficient for the analysis of complexity when the complexity of the implementation of the roughly described parts is well known.

The aim of the complexity analysis is to provide a robust analysis in the sense that the result does not depend on the structural and technological characteristics of concrete sequential computers and their system software. Here, we focus on the time complexity of computations and only sometimes consider the space complexity. We distinguish two basic ways of complexity measurement, namely the **uniform cost** measurement and the **logarithmic cost** measurement.

The approach based on uniform cost measure is the simplest one. The measurement of time complexity consists of determining the overall number of elementary[35] instructions executed in the considered computation, and the measurement of space complexity consists of determining the number of variables used in the computation. The advantage of this measurement is that it is simple. The drawback is that it is not always adequate because it considers

[35] Elementary instructions are arithmetic instructions over integers, comparison of two integers, reading, writing, loading integers and symbols, etc.

cost 1 for an arithmetic operation over two integers independently of their size. When the operands are integers whose binary representations consist of several hundreds of bits, none of them can be stored in one computer word (16 or 32 bits). Then the operands must be stored in several computer words (i.e., one needs several space units (variables) to save them) and the execution of the arithmetic operation over these two large integers corresponds to the execution of a special program performing an operation over large integers by several operations over integers of the computer word size. Thus, the uniform cost measurement may be applied in the cases where one can assume that during the whole computation all variables contain values whose size is bounded by a fixed constant (hypothetical computer word length). This is the case for computing over $\mathbb{Z}_p$ for a fixed prime $p$ or working with logical values only (in Boolean algebra).

A serious anomaly of the use of the uniform cost measurement appears in the following example. Let $k$ and $a \geq 2$ be two positive integers of sizes not exceeding the size of a hypothetical computer word. Consider the task of computing the number $a^{2^k}$. This can be done with the following strategy. Compute

$$a^2 = a \cdot a,\ a^4 = a^2 \cdot a^2,\ a^8 = a^4 \cdot a^4, \ldots,\ a^{2^k} = a^{2^{k-1}} \cdot a^{2^{k-1}}.$$

The uniform cost space complexity is 3 because one additional variable is sufficient to execute the following computation

$$\textbf{for } i = 1 \textbf{ to } k \textbf{ do } a := a * a.$$

The uniform cost time complexity is in $O(k)$ because exactly $k$ multiplications are executed. This contrasts with the fact that one needs at least $2^k$ bits to represent the result $a^{2^k}$ and to write $2^k$ bits any machine needs $\Omega(2^k)$ operations on its machine words. Since this consideration works for every positive integer $k$, we have an exponential gap between the uniform cost time complexity and any realistic time complexity, and an unbounded difference between the uniform cost space complexity and any realistic space complexity.

The solution to such a situation, where the values of variables grow unboundedly, is to use the **logarithmic cost** measurement. With respect to this measurement the cost of every elementary operation is the sum of the sizes of the binary representations of the operands.[36] Obviously, this approach to complexity measurement avoids anomalies of the kind mentioned above, and it is generally adopted in the complexity analysis of algorithms. Sometimes one distinguishes the time complexity of distinct operations. Since the best algorithm for multiplying two $n$-bit integers needs $\Omega(n \cdot \log n)$ binary operations, the complexity of multiplication and division is considered to be $O(n \cdot \log n)$ while addition, subtraction, and assignment have costs linear in the binary size of the arguments.

[36] If one wants to be very careful then the binary length of the addresses of the variables in memory (that correspond to the operands) can be added.

**Definition 2.3.3.1.** *Let $\Sigma_I$ and $\Sigma_O$ be alphabets. Let $A$ be an algorithm that realizes a mapping from $\Sigma_I^*$ to $\Sigma_O^*$. For every $x \in \Sigma_I^*$, $\boldsymbol{Time_A(x)}$ denotes the time complexity*[37] *(according to the logarithmic cost) of the computation of $A$ on the input $x$, and $\boldsymbol{Space_A(x)}$ denotes the space complexity (according to the logarithmic cost measurement) of the computation of $A$ on $x$.*

One never considers the time complexity $Time_A$ (the space complexity $Space_A$) as a function from $\Sigma_I^*$ to $\mathbb{N}$. This is because we usually have exponentially many inputs for every input length and to estimate $Time_A(x)$ for every $x \in \Sigma_I^*$ would often be an unrealistic job. Even if this would succeed, the description of $Time_A$ could be so complex that one would have trouble determining some fundamental characteristics of $Time_A$. The comparison of the complexities of two algorithms for the same algorithmic problem could be a difficult job, too. Thus, the complexity is always considered as a function of the **input size** and one observes the asymptotic growth of this function.

**Definition 2.3.3.2.** *Let $\Sigma_I$ and $\Sigma_O$ be two alphabets. Let $A$ be an algorithm that computes a mapping from $\Sigma_I^*$ to $\Sigma_O^*$. The* **(worst case) time complexity of $A$** *is a function $Time_A : (\mathbb{N} - \{0\}) \to \mathbb{N}$ defined by*

$$\boldsymbol{Time_A(n)} = \max\{Time_A(x) \mid x \in \Sigma_I^n\}$$

*for every positive integer $n$. The* **(worst case) space complexity of $A$** *is a function $Space_A : (\mathbb{N} - \{0\}) \to \mathbb{N}$ defined by*

$$\boldsymbol{Space_A(n)} = \max\{Space_A(x) \mid x \in \Sigma_I^n\}.$$

$Time_A$ is defined in such a way that we know that every input of size $n$ (i.e., every input from $\Sigma_I^n$) is solved by $A$ in time at most $Time_A(n)$ and that there is an input $x$ of size $n$ with $Time_A(x) = Time_A(n)$. This is the reason why one calls this kind of complexity analysis the **worst case analysis**. The drawback of the worst case analysis may occur when one uses it for an algorithm with very different complexities on inputs of the same length.[38] In that case one can consider the average case analysis that consists of determining the average complexity on all instances of size $n$. There are two problems with this approach. First, to determine the average complexity is usually a much harder problem than to determine $Time_A(n)$, and in many cases we are not able to perform the average complexity analysis. Secondly, the average complexity provides useful information only if the average is taken over a realistic probability distribution over the inputs of any fixed length. Such input distributions may essentially differ from application to application, i.e., an average cost analysis according to a specific input distribution does not

[37] Note that one assumes that an algorithm terminates for every input $x$, and so $Time_A(x)$ is always a non-negative integer.

[38] A famous example is the simplex algorithm for the problem of linear programming.

provide any robust answer according to all possible applications of this algorithm. Even in the case of one specific application, we are often not able to estimate these input distributions for every input size. In this book it will be sufficient to consider the worst case complexity only, because it will provide a good characterization of the behavior of almost all considered algorithms.

An important observation is that we have fixed the input length in Definition 2.3.3.2 in the logarithmic cost manner. There, the length of an input is considered to be the length of its code over $\Sigma_I$. Remember that our interpretation of an input alphabet $\Sigma_I$ is that it is the set of all computer words allowed. Note that sometimes it is sufficient to consider the unit cost of the input size. This means that the size is the number of items (for instance, integers) of the input. This is again acceptable if all the items of the input are of equal size. For instance, for a $n \times n$ matrix over $\mathbb{Z}_p$ for a fixed prime $p$, one can consider the size $n^2$ or even $n$, and measure the complexity according to this parameter. But usually one may not use this approach for problems from algorithmic number theory, where the input consists of one or a few numbers. In such cases we strictly consider the input size as the length of the binary representation of the input.

In this book we shall very rarely perform a very precise analysis of the designed algorithms, because of the following two reasons. First, to make a precise analysis one has to deal with the implementation details that are usually omitted here. Secondly, we consider hard problems only and we will be satisfied with establishing reasonable asymptotic upper bounds on the complexity of designed algorithms.

The main goal of complexity theory is to classify algorithmic problems according to their computational difficulty. Since the time complexity as the number of executed operations seems to be the central measure of algorithm complexity, one prefers to measure the computational difficulty of problems in terms of time complexity. Intuitively, the time complexity of a problem $U$ could be a function $T_U : \mathbb{N} \to \mathbb{N}$ such that $\Theta(T_U(n))$ operations are necessary and sufficient to solve $U$. But this is still not a consistent definition of the complexity of $U$ because we need to have an algorithmic solution, i.e., an algorithm solving $U$ in the time complexity $O(T_U(n))$. Thus, a natural way to define the time complexity of $U$ seems to be to say that the time complexity of $U$ is the time complexity for the "best" (optimal) algorithm for $U$. Unfortunately, the following fundamental result of complexity theory shows that this approach to define $T_U$ is not consistent.

**Theorem 2.3.3.3.** *There is a decision problem $(L, \Sigma_{bool})$ such that, for every algorithm $A$ deciding $L$, there exists another algorithm $B$ deciding $L$, such that*

$$Time_B(n) = \log_2(Time_A(n))$$

*for infinitely many positive integers $n$.*

Obviously, Theorem 2.3.3.3 implies that there is no best (optimal) algorithm for $L$ and so it is impossible to define complexity of $L$ as a function

from $\mathbb{N}$ to $\mathbb{N}$ in the way proposed above. This is the reason why one does not try to define the complexity of algorithmic problems but rather the lower and upper bounds on the problem complexity.

**Definition 2.3.3.4.** *Let $U$ be an algorithmic problem, and let $f$, $g$ be functions from $\mathbb{N}$ to $\mathbb{R}^+$. We say that $O(g(n))$ is an* **upper bound on the time complexity of $U$** *if there exists an algorithm $A$ solving $U$ with $Time_A(n) \in O(g(n))$.*

*We say that $\Omega(f(n))$ is a* **lower bound on the time complexity of $U$** *if every algorithm $B$ solving $U$ has $Time_B(n) \in \Omega(f(n))$.*

*An algorithm $C$ is* **optimal** *for the problem $U$ if $Time_C(n) \in O(g(n))$ and $\Omega(g(n))$ is a lower bound on the time complexity of $U$.*

To establish an upper bound on the complexity of a problem $U$ it is sufficient to find an algorithm solving $U$. Establishing a nontrivial lower bound on the complexity of $U$ is a very hard task because it requires proving that every of the infinitely many known and unknown algorithms solving $U$ must have its time complexity in $\Omega(f(n))$ for some $f$. This is in fact a nonexistence proof because one has to prove the nonexistence of any algorithm solving $U$ with the time complexity asymptotically smaller than $f(n)$. The best illustration of the hardness of proving lower bounds on problem complexity is the fact that we know thousands of algorithmic problems for which

(i) the time complexity of the best known algorithm is exponential in the input size, and
(ii) no superlinear lower bound such as $\Omega(n \log n)$ is known for any of them.

Thus, we conjecture, for many of these problems, that there does not exist any algorithm solving them in time polynomial in the input size, but we are unable to prove that one really needs more than $O(n)$ time to solve them.

To overcome our disability to prove lower bounds on problem complexity (i.e., to prove that some problems are hard), some concepts providing reasonable arguments for hardness of concrete problems instead of the evidence of their hardness were developed. These concepts are connected with some formal manipulation of algorithms and complexity in terms of Turing machines (TMs) and Turing machine complexity. We assume that the reader is familiar with the Turing machine model. Remember, that following the **Church-Turing thesis**, a Turing machine is a formalization of the intuitive notion of algorithm. This means that a problem $U$ can be solved by an algorithm (computer program in any programming language formalism) if and only if there exists a Turing machine solving $U$. Using the formalism of TMs it was proved that for every increasing function $f : \mathbb{N} \to \mathbb{R}^+$

(i) there exists a decision problem such that every TM solving it has the time complexity in $\Omega(f(n))$,
(ii) but there is a TM solving it in $O(f(n) \cdot \log f(n))$ time.

This means that there is an infinite hierarchy of the hardness of decision problems. In what follows we shall use the terms algorithm and computer program instead of the term Turing machine whenever possible, and so we omit unnecessary technicalities.

One can say that the main objective of the complexity theory is

*to find a formal specification of the class of practically solvable problems*

and

*to develop methods enabling the classification of algorithmic problems according to their membership in this class.*

The first efforts in searching for a reasonable formalization of the intuitive notion of practically solvable problems result in the following definition. Let, for every TM (algorithm) $M$, $\boldsymbol{L(M)}$ denote the language decided by $M$.

**Definition 2.3.3.5.** *We define the complexity class* P *of languages decidable in polynomial-time by*

$$\mathbf{P} = \{L = L(M) \mid M \text{ is a TM (an algorithm) with } Time_M(n) \in O(n^c) \text{ for some positive integer } c\}.$$

*A language (decision problem) $L$ is called* **tractable (practically solvable)** *if $L \in$ P. A language $L$ is called* **intractable** *if $L \notin$ P.*

Definition 2.3.3.5 introduces the class P of decision problems decidable by polynomial-time computations and says that exactly the set P is the specification of the class of tractable (practically solvable) problems. Let us discuss the advantages and the disadvantages of this formal definition of tractability. The two main reasons to connect polynomial-time computations with the intuitive notion of practical solvability are the following:

(1) The definition of the class P is robust in the sense that P is invariant for all reasonable models of computation. The class P remains the same independent of whether it is defined in terms of polynomial-time Turing machines, in terms of polynomial-time computer programs over any programming language, or in terms of polynomial-time algorithms of any reasonable formalization of computation. This is the consequence of another fundamental result of complexity theory saying that all computation models (formalizations of the intuitive notion of algorithm) that are realistic in the complexity measurement are polynomially equivalent. **Polynomially equivalent** means that if there is a polynomial-time algorithm for an algorithmic problem $U$ in one formalism, then there is a polynomial-time algorithm for $U$ in the other formalism, and vice versa. Turing machines and all programming languages used are in this class of polynomially equivalent computing models. Thus, if one designs a polynomial-time algorithm for $U$ in C++, then there is a polynomial-time algorithm for $U$ in any

reasonable computing formalism. On the other hand, if one proves that there is no polynomial-time TM deciding a language $L$, then one can be sure that there is no polynomial-time computer program deciding $L$. Note that this kind of robustness is very important and it must be required for any reasonable specification of the class of tractable problems.

(2) While the first reason for choosing P is a theoretical one, the second reason is more connected with intuition about practical solvability and experience in algorithm design. Consider the table in Figure 2.14 that illustrates the growth of complexity functions $10n$, $2n^2$, $n^3$, $2^n$, and $n!$ for input sizes 10, 50, 100, and 300.

| $f(n)$ \ $n$ | 10 | 50 | 100 | 300 |
|---|---|---|---|---|
| $10n$ | 100 | 500 | 1000 | 3000 |
| $2n^2$ | 200 | 5000 | 20000 | 180000 |
| $n^3$ | 1000 | 125000 | 1000000 | 27000000 |
| $2^n$ | 1024 | (16 digits) | (31 digits) | (91 digits) |
| $n!$ | $\approx 3.6 \cdot 10^6$ | (65 digits) | (161 digits) | (623 digits) |

**Fig. 2.14.**

Observe that if the values of $f(n)$ are too large we write only the number of digits of the decimal representation of $f(n)$. Assuming that one has a computer that executes $1000000 = 10^6$ operations per second, an algorithm $A$ with $Time_A(n) = n^3$ runs in 27 seconds for $n = 300$. But if $Time_A(n) = 2^n$, then the execution of $A$ for $n = 50$ would take more than 30 years, and for $n = 100$ more than $3 \cdot 10^{16}$ years. If one compares the values of $2^n$ and $n!$ for a realistic input size between 100 and 300 with the suggested number of seconds since the "Big Bang" that has 21 digits[39], then everybody sees that the execution of algorithms of exponential complexity on realistic inputs is beyond the borders of physical reality. Moreover, observe the following properties of the functions $n^3$ and $2^n$. If $M$ is the time you can wait for the results, then developing a computer that executes twice as many instructions in a time unit as the previous computer, helps you

(i) to increase the size of tractable input instances from $M^{1/3}$ to $\sqrt[3]{2} \cdot M^{1/3}$ for an $n^3$-algorithm (i.e., one can compute on $\sqrt[3]{2}$ times larger sizes of input instances than before), but

(ii) to increase the size of tractable input instances by 1 bit for a $2^n$-algorithm.

Thus, algorithms of exponential complexity cannot be considered practical, and the algorithms of the polynomial-time complexity $O(n^c)$ for small $c$'s can be considered practical. Of course, a running time $n^{1000}$ is unlikely

[39] Note that the number of protons in the known universe has 79 digits.

to be of any practical use because $n^{1000} > 2^n$ for all reasonable sizes $n$ of inputs. Nevertheless, experience has proved the reasonability of considering polynomial-time computations to be tractable. In almost all cases, once a polynomial-time algorithm has been found for an algorithmic problem that formerly appeared to be hard, some key insight into the problem has been gained and new polynomial-time algorithms of a low[40] degree of the polynomial have been designed for the problem. There are only a few known exceptions of nontrivial problems where the best polynomial-time algorithm is not of practical utility.

Above we argued that P is a good specification of the class of practically solvable problems. Nevertheless, this whole book is devoted to solving problems that are probably not in P. But this does not destroy the idea of taking polynomial-time as the threshold of practical solvability. Our approaches to solving problems outside P usually change our requirements such as

- using randomized algorithms (providing the right solution with some probability) instead of deterministic ones (providing the right solution with certainty) or
- searching for an approximation of an optimal solution instead of searching for an optimal solution.

Thus, the current view on the specification of the class of tractable problems is more or less connected with randomized polynomial-time (approximation[41]) algorithms.

Having the class P, one would like to have methods of classifying problems according to their membership in P. To prove the membership of a decision problem $L$ to P, it is sufficient to design a polynomial-time algorithm for $L$. As already mentioned above, we do not have any method that would be able to prove for most of the practical problems of interest that they are not in P, i.e., that they are intractable (hard). To overcome this unpleasant situation the concept of NP-completeness was introduced. This concept provides at least a good reason to believe that a specific problem is hard, when one is unable to prove the evidence of this fact.

To introduce the concept of NP-completeness we have to consider **nondeterministic computation**. Nondeterminism is nothing natural from the computational point of view because we do not know any way it can be efficiently implemented on real computers.[42] For the reader not familiar with nondeterministic Turing machines, one can introduce nondeterminism to every programming language by adding an operation $choice(a, b)$ with the meaning *goto* $a$ or *goto* $b$. Thus, the computation may branch into two computations.

[40] At most 6, but often 3

[41] In the case of optimization problems

[42] In fact we do not believe that there exists an efficient simulation of nondeterministic computations by deterministic ones.

This means that a nondeterministic TM (algorithm) may have a lot of computations on an input $x$, while any deterministic TM (algorithm) has exactly one computation for every input. One usually represents all computations of a nondeterministic algorithm $A$ on an input $x$ by the so-called **computation tree of $A$ on $x$**. Such a computation tree is depicted in Figure 2.15 for a nondeterministic algorithm $A$ that accepts SAT (solves the decision problem $(\text{SAT}, \Sigma_{logic})$).

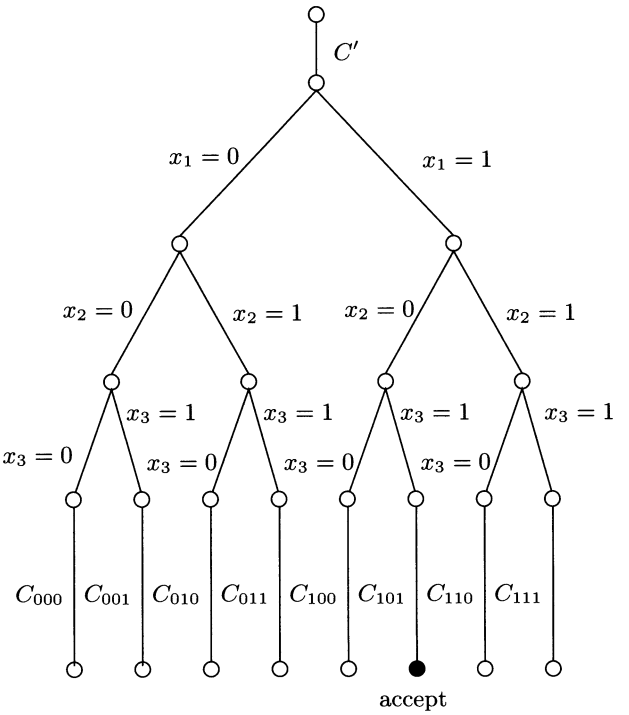



**Fig. 2.15.**

The tree $T_A(x)$ in Figure 2.15 contains all computations of $A$ on the input $x$ corresponding to the formula

$$\Phi_x = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3) \wedge \overline{x}_2$$

over three variables $x_1, x_2$, and $x_3$. $A$ proceeds as follows. First $A$ deterministically (i.e., without computation branching) verifies whether $x$ is a code of a formula $\Phi_x$ over $\Sigma_{logic}$ in the computation part $C'$ (Figure 2.15). If not, $A$ rejects the input. Assume that $x$ codes the above formula $\Phi_x$. The general strategy of $A$ is to nondeterministically guess an assignment that satisfies $\Phi_x$. This is realized in as many branching steps as there are variables in the formula. In our example (Figure 2.15) $A$ first branches into two computations.

The left one corresponds to the guess that $x_1 = 0$ and the right computation corresponds to the guess that $x_1 = 1$. Each of these two computations immediately branches into two computations according to the choice of the Boolean value for $x_2$. Thus, we obtain 4 computations and each of them immediately branches according to the choice of $x_3$. Finally, we have 8 computations, each of them corresponding to the choice of one of the 8 assignments for $\{x_1, x_2, x_3\}$. For every assignment $\alpha$, the corresponding computation $C_\alpha$ deterministically verifies whether $\alpha$ satisfies $\Phi_x$. If $\alpha$ satisfies $\Phi_x$, $A$ accepts $x$; otherwise, $A$ rejects $x$. We observe in Figure 2.15 that $A$ accepts $x$ in the computation $C_{101}$ because 101 is the only assignment satisfying $\Phi_x$. All other computations $C_\alpha$, $\alpha \neq 101$, reject $x$.

The acceptance and the complexity of nondeterministic algorithms are defined as follows.

**Definition 2.3.3.6.** *Let $M$ be a nondeterministic TM (algorithm). We say that* **$M$ accepts a language $L$, $L = L(M)$,** *if*

*(i) for every $x \in L$, there exists at least one computation of $M$ that accepts $x$, and*
*(ii) for every $y \notin L$, all computations of $M$ reject $y$.*

*For every input $w \in L$, the* **time complexity $Time_M(w)$ of $M$ on $w$** *is the time complexity of the shortest accepting computation of $M$ on $w$. The* **time complexity of $M$** *is the function $Time_M$ from $\mathbb{N}$ to $\mathbb{N}$ defined by*

$$Time_M(n) = \max\{ Time_M(x) \mid x \in L(M) \cap \Sigma^n \}.$$

*We define the class*

$$\mathbf{NP} = \{L(M) \mid M \text{ is a polynomial-time nondeterministic TM}\}$$

*as the class of decision problems decided nondeterministically in polynomial time.*

We observe that, for a nondeterministic algorithm, it is sufficient that one of the choices is the right way providing the solution (correct answer). For a decision problem $(L, \Sigma)$, any (deterministic) algorithm $B$ has to decide whether $x \in L$ or $x \notin L$ for every input $x$. The accepting (rejecting) computation of $B$ on $x$ can be viewed as the proof of the fact $x \in L$ ($x \notin L$). So, the complexity of (deterministic) algorithms for decision problems may be considered as the complexity of producing a proof of the correctness of the output. Following the example of the nondeterministic algorithm for SAT (Figure 2.15), the essential part of the computation from the complexity point of view is the verification whether a guessed assignment satisfies the given formula or not. Thus, the complexity of the nondeterministic algorithm $A$ is in fact the complexity of the verification whether "a given assignment $\alpha$ proves the satisfiability of $\Phi_x$". This leads to the following hypothesis:

*The complexity of deterministic computations is the complexity of proving the correctness of the produced output, while the complexity of nondeterministic computation is equivalent to the complexity of deterministic verification of a given proof (certificate) of the fact $x \in L$.*

In what follows we show that this hypothesis is true for polynomial-time computations. The fact that deterministic computations can be viewed as proofs of the correctness of the produced outputs is obvious. To prove that the nondeterministic complexity is the complexity of the verification we need the following formal concept.

**Definition 2.3.3.7.** *Let $L \subseteq \Sigma^*$ be a language. An algorithm $A$ working on inputs from $\Sigma^* \times \{0,1\}^*$ is called a* **verifier for $L$**, *denoted* $\boldsymbol{L = V(A)}$, *if*

$$L = \{w \in \Sigma^* \mid A \text{ accepts } (w,c) \text{ for some } c \in \{0,1\}^*\}.$$

*If $A$ accepts $(w,c) \in \Sigma^* \times \{0,1\}^*$, we say that $c$ is a* **proof (certificate**[43]**)** *of the fact $w \in L$.*

*A verifier $A$ for $L$ is called a* **polynomial-time verifier** *if there exists a positive integer $d$ such that, for every $w \in L$, $Time_A(w,c) \in O(|w|^d)$ for a proof $c$ of $w \in L$.*

*We define the* **class of polynomially verifiable languages** *as*

$$\mathbf{VP} = \{V(A) \mid A \text{ is a polynomial-time verifier}\}.$$

We illustrate Definition 2.3.3.7 with the following example. A verifier for SAT is an algorithm that, for each input from $(x,c) \in \Sigma^*_{logic} \times \Sigma^*_{bool}$, interprets $x$ as a representation of a formula $\Phi_x$ and $c$ as an assignment of Boolean values to the variables of $\Phi_x$. If this interpretation is possible (i.e., $x$ is a correct code of a formula $\Phi_x$ and the length of $c$ is equal to the number of variables in $\Phi_x$), then the verifier checks whether $c$ satisfies $\Phi_x$. Obviously, the verifier accepts $(x,c)$ if and only if $c$ is an assignment satisfying $\Phi_x$. We observe that the verifier is a polynomial-time algorithm because a certificate $c$ for $x$ is always shorter than $x$ and one can efficiently evaluate a formula for a given assignment to its variables.

**Exercise 2.3.3.8.** Describe a polynomial-time verifier for

(i) HC,
(ii) VC, and
(iii) CLIQUE. □

The following theorem proves our hypothesis in the framework of polynomial-time.

[43] Note that a certificate of "$w \in L$" needs not necessarily be a mathematical proof of the fact "$w \in L$". More or less, a certificate should be considered as additional information that essentially simplifies proving the fact "$w \in L$".

**Theorem 2.3.3.9.**

$$\text{NP} = \text{VP}.$$

*Proof.* We prove NP = VP by proving NP $\subseteq$ VP and VP $\subseteq$ NP.

(i) We prove $NP \subseteq$ VP. Let $L \in$ NP, $L \subseteq \Sigma^*$. Then there exists a polynomial-time nondeterministic algorithm (TM) $M$ such that $L = L(M)$. One can construct a polynomial-time verifier $A$ that works as follows:

$A$: **Input:** $(x, c) \in \Sigma^* \times \Sigma^*_{bool}$.
(1) $A$ interprets $c$ as a navigator for the simulation of the nondeterministic choices of $M$. $A$ simulates the work of $M$ (step by step) on $w$. If $M$ has a choice of two possibilities, then $A$ takes the first one if the next bit of $c$ is 0, and $A$ takes the second possibility if the next bit of $c$ is 1. In this way $A$ simulates exactly one of the computations of $M$ on $x$.
(2) If $M$ still has a choice and $A$ used already all bits of $c$, then $A$ halts and rejects.
(3) If $A$ succeeds to simulate a complete computation of $M$ on $x$, then $A$ accepts $(x, c)$ iff $M$ accepts $x$ in this computation.

Obviously, $V(A) = L(M)$ because if $M$ accepts $x$, then there exists a certificate $c$ that corresponds to a sequence of nondeterministic choices unambiguously determining an accepting computation of $A$ on $x$. Since $A$ does nothing else than simulating $M$ step by step and, for every $x \in L(M)$, $A$ simulates the shortest accepting computation of $M$ on $x$, too, $V$ is a polynomial-time verifier for $L$.

(ii) We prove VP $\subseteq$ NP. Let $L \subseteq \Sigma^*$, for an alphabet $\Sigma$, be a language from VP. Thus, there exists a polynomial-time verifier $A$ such that $V(A) = L$. One can design a polynomial-time nondeterministic algorithm $M$ that simulates $A$ as follows.

$M$: **Input:** an $x \in \Sigma^*$.
(1) $M$ nondeterministically generates a word $c \in \{0, 1\}^*$.
(2) $M$ simulates step by step the work of $A$ on $(x, c)$.
(3) $M$ accepts $x$, if $A$ accepts $(x, c)$, and $M$ rejects $x$, if $A$ rejects $(x, c)$.

Obviously, $L(M) = V(A)$ and $M$ runs in polynomial-time. □

Now we have the following situation. We have defined two language classes P and NP. Almost all interesting decision problems appearing in practice are in NP. So, NP is an interesting class from the practical point of view, too. Almost everybody conjectures, if not directly believes, that P $\subset$ NP. The two main reasons for this conjecture follow.

(i) *A theoretical reason*
People do not believe that finding a proof is as easy as verifying the correctness of a given proof. This mathematical intuition supports the hypothesis P $\subset$ NP = VP.

(ii) *A practical reason (experience)*
We know more than 3000 problems in NP, many of them have been investigated for 40 years, for which no deterministic polynomial-time algorithm is known. It is not very probable that this is only the consequence of our disability to find efficient algorithms for them. Even if this would be the case, for the current practice the classes P and NP are different because we do not have polynomial-time algorithms for numerous problems from NP.

This provides a new idea for how to "prove" the hardness of some problems, even if we do not have direct mathematical methods for this purpose. Let us try to prove $L \notin P$ for a $L \in \mathrm{NP}$ by the additional assumption $\mathrm{P} \subset \mathrm{NP}$. The idea is to say that a decision problem $L$ is one of the hardest problems in NP if $L \in \mathrm{P}$ would immediately imply $\mathrm{P} = \mathrm{NP}$. Since we do not believe $\mathrm{P} = \mathrm{NP}$, this is a reasonable argument to believe that $L \notin \mathrm{P}$, i.e., that $L$ is hard. Since we want to avoid hard proofs of the nonexistence of efficient algorithms, we define the hardest problems of NP as such problems, so that any hypothetical efficient algorithm for them can be transformed into an efficient algorithm for any other decision problem in NP. The following definition formalizes this idea.

**Definition 2.3.3.10.** *Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ be two languages. We say that $L_1$ is* **polynomial-time reducible**[44] *to $L_2$, $\boldsymbol{L_1 \leq_p L_2}$, if there exists a polynomial-time algorithm $A$ that computes a mapping from $\Sigma_1^*$ to $\Sigma_2^*$ such that, for every $x \in \Sigma_1^*$,*

$$x \in L_1 \iff A(x) \in L_2.$$

*$A$ is called the* **polynomial-time reduction** *from $L_1$ to $L_2$.*
*A language $L$ is called* **NP-hard** *if, for every $U \in \mathrm{NP}$, $U \leq_p L$.*
*A language $L$ is called* **NP-complete** *if*

*(i) $L \in \mathrm{NP}$, and*
*(ii) $L$ is NP-hard.*

First, observe that $L_1 \leq_p L_2$ means that $L_2$ is at least as hard as $L_1$ because if a polynomial-time algorithm $M$ decides $L_2$ then the "concatenation" of $A$ reducing $L_1$ to $L_2$ and $M$ provides a polynomial-time algorithm for $L_1$. The following claim shows that NP-hardness is exactly the term that we have searched for.

**Lemma 2.3.3.11.** *If $L$ is NP-hard and $L \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.*

*Proof.* Let $L \subseteq \Sigma^*$ be an NP-hard language and let $L \in \mathrm{P}$. Then, there is a polynomial-time algorithm $M$ with $L = L(M)$. We prove that for every

[44] Note that polynomial-time reducibility of Definition 2.3.3.10 is also called Karp-reducibility or polynomial-time many-to-one reducibility in the literature.

$U \in$ NP, $U \subseteq \Sigma_1^*$, there is a polynomial-time algorithm $A_U$ with $L(A_U) = U$, i.e., that $U \in$ P. Since $U \leq_p L$, there exists a polynomial-time algorithm $B$ such that $x \in U$ iff $B(x) \in L$. An algorithm $A_U$ with $L(A_U) = U$ can work as follows.

$A_U$ : **Input:** an $x \in \Sigma_1^*$.
Step 1: $A_U$ simulates the work of $B$ on $x$ and computes $B(x)$.
Step 2: $A_U$ simulates the work of $M$ on $B(x) \in \Sigma^*$. $A_U$ accepts $x$ iff $M$ accepts $B(x)$.

Since $x \in U$ iff $B(x) \in L$, $L(A_U) = U$. Since $Time_{A_U}(x) = Time_B(x) + Time_M(B(x))$, $B$ and $M$ work in polynomial time, and $|B(x)|$ is polynomial in $|x|$, we see that $A_U$ is a polynomial-time algorithm. □

The remaining task is to prove, for a specific language $L$, that all languages from NP are reducible to $L$. This so-called master reduction was proved for SAT. We do not present the proof here because we want to omit technical considerations based on the Turing machine formalism.

**Theorem 2.3.3.12 (Cook's Theorem).** SAT *is* NP*-complete.*[45]

From the practical point of view one is interested in a simple method for establishing that a problem $U$ of interest is NP-hard. One does not need any variation of the master reduction to do it. As claimed in the following observation it is sufficient to take a known NP-hard problem $L$ and to find a polynomial-time reduction from $U$ to $L$.

**Observation 2.3.3.13.** Let $L_1$ and $L_2$ be two languages. If $L_1 \leq_p L_2$ and $L_1$ is NP-hard, then $L_2$ is NP-hard.

**Exercise 2.3.3.14.** Prove Observation 2.3.3.13.
(*Hint:* The proof of Observation 2.3.3.13 is very similar to the proof of Lemma 2.3.3.11.) □

Historically, by using the claim of Observation 2.3.3.13, SAT has been used to prove NP-completeness for more than 3000 decision problems. An interesting point is that an NP-complete problem can be viewed as a problem that somehow codes any other problem from NP. For instance, the NP-completeness of SAT means that every decision problem $(L, \Sigma)$ from NP can be expressed in the language of Boolean formulae. This is right because, for each input $x \in \Sigma^*$, we can efficiently construct a Boolean formula $\Phi_x$ that is satisfiable iff $x \in L$. Similarly, proving NP-hardness of a problem from graph theory shows that every problem from NP can be expressed in a graph-theoretical language. In what follows we present some examples of reductions between different formal representations (languages).

[45] A very detailed, transparent proof of this theorem is given in [Hro03].

**Lemma 2.3.3.15.** SAT $\leq_p$ CLIQUE.

*Proof.* Let $\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ be a formula in CNF, where $F_i = (l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik_i})$, $k_i \in \mathbb{N} - \{0\}$, for $i = 1, 2, \ldots, m$. We construct an input instance $(G, k)$ of the clique problem, such that $G$ contains a $k$-clique iff $\Phi$ is satisfied, as follows.

$k := m$;
$G = (V, E)$, where

$$V := \{[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq k_i\},$$

i.e., we take one vertex for every occurrence of a literal in $\Phi$;

$$E := \{\{[i, j], [r, s]\} \mid \text{ for all } [i, j], [r, s] \in V \text{ such that } i \neq r \text{ and } l_{ij} \neq \bar{l}_{rs}\},$$

i.e., the edges connect vertices corresponding to literals from different clauses only, and additionally if $\{u, v\} \in E$, then the literal corresponding to $u$ is not the negation of the literal corresponding to $v$.

Observe that the above construction of $(G, k)$ from $\Phi$ can be computed efficiently in a straightforward way.

Figure 2.16 shows the graph $G$ corresponding to the formula

$$\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3) \wedge \overline{x}_2.$$

It remains to be shown that

$$\Phi \text{ is satisfiable} \iff G \text{ contains a clique of size } k = m. \tag{2.27}$$

The idea of the proof is that two literals (vertices) $l_{ij}$ and $l_{rs}$ are connected by an edge if and only if they are from different clauses and there exists an assignment for which both values of $l_{ij}$ and $l_{rs}$ are 1's. Thus, a clique corresponds to the possibility of finding an input assignment that evaluates all the literals corresponding to the vertices of the clique to 1's.

We prove the equivalence (2.27) by subsequently proving both implications.

(i) Let $\Phi$ be satisfiable. Thus, there exists an assignment $\varphi$ such that $\varphi(\Phi) = 1$. Obviously, $\varphi(F_i) = 1$ for all $i \in \{1, \ldots, m\}$. This implies that, for every $i \in \{1, \ldots, m\}$, there exists a $d_i \in \{1, \ldots, k_i\}$ such that $\varphi(l_{id_i}) = 1$. We claim that the set of vertices $\{[i, d_i] \mid 1 \leq i \leq m\}$ defines an $m$-clique in $G$. Obviously, $[1, d_1], [2, d_2], \ldots, [m, d_m]$ are from different clauses. The equality $l_{id_i} = \bar{l}_{jd_j}$ for some $i \neq j$ would imply $\omega(l_{id_i}) \neq \omega(l_{jd_j})$ for every input assignment $\omega$ and so $\varphi(l_{id_i}) = \varphi(l_{jd_j})$ would be impossible. Thus, $l_{id_i} \neq \bar{l}_{jd_j}$ for all $i, j \in \{1, \ldots, m\}$, $i \neq j$, and $\{[i, d_i], [j, d_j]\} \in E$ for all $i, j = 1, \ldots, m, i \neq j$.

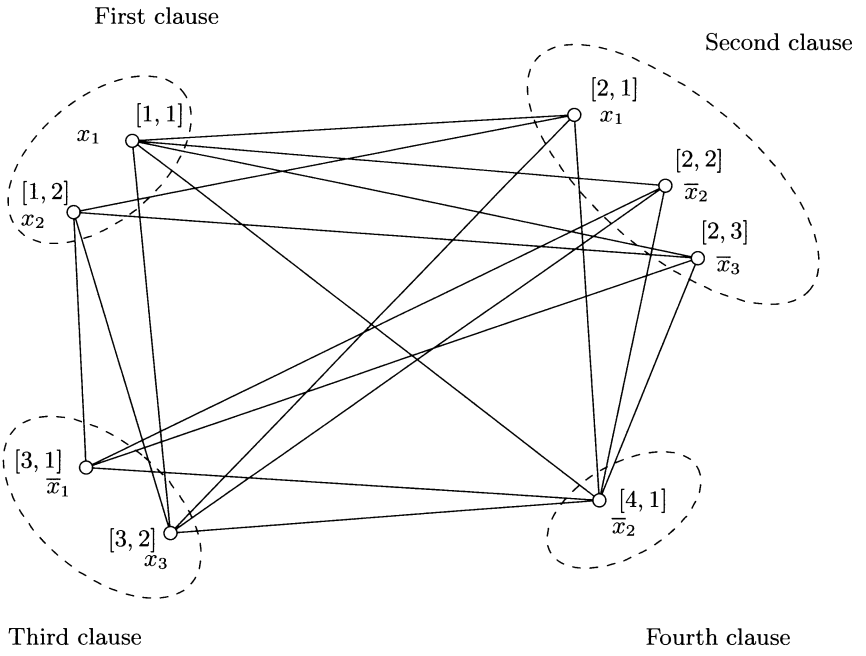



**Fig. 2.16.**

(ii) Let $Q$ be a clique of $G$ with $k = m$ vertices. Since two vertices are connected in $G$ only if they correspond to two literals from different clauses, there exists $d_1, d_2, \ldots, d_m$, $d_p \in \{1, 2, \ldots, k_p\}$ for $p = 1, \ldots, m$, such that $Q = \{[1, d_1], [2, d_2], \ldots, [m, d_m]\}$. Following the construction there exists an assignment $\varphi$ such that $\varphi(l_{1d_1}) = \varphi(l_{2d_2}) = \cdots = \varphi(l_{md_m}) = 1$. This directly implies $\varphi(F_1) = \varphi(F_2) = \cdots = \varphi(F_m) = 1$ and so $\varphi$ satisfies $\Phi$. □

**Lemma 2.3.3.16.**

$$\text{CLIQUE} \leq_p \text{VC}.$$

*Proof.* Let $G = (V, E), k$ be an input of the clique problem. We construct an input $(\overline{G}, m)$ of the vertex cover problem as follows:

$m := |V| - k$,
$\overline{G} = (V, \overline{E})$, where $\overline{E} = \{\{v, u\} \mid v, u \in V, u \neq v, \text{ and } \{u, v\} \notin E\}$.
Obviously, this construction can be executed in linear time.

Figure 2.17 illustrates the construction of the graph $\overline{G}$ from $G$. The idea of the construction is the following one. If $Q$ is a clique in $G$, then there is no edge between any pair of vertices from $Q$ in $\overline{G}$. Thus, $V - Q$ must be a vertex cover in $\overline{G}$. So, the clique $\{v_1, v_4, v_5\}$ of $G$ in Figure 2.17 corresponds to the vertex cover $\{v_2, v_3\}$ in $\overline{G}$. The clique $\{v_1, v_2, v_5\}$ of $G$ corresponds to

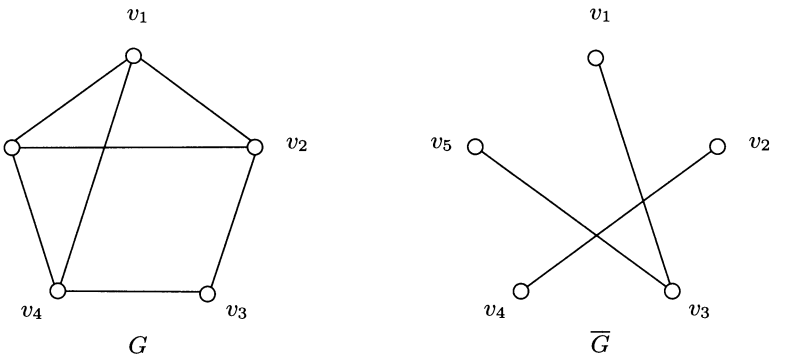



**Fig. 2.17.**

the vertex cover $\{v_3, v_4\}$ in $\overline{G}$ and the clique $\{v_1, v_2\}$ in $G$ corresponds to the vertex cover $\{v_3, v_4, v_5\}$ in $\overline{G}$.

Obviously, for proving

$$(G, k) \in \text{CLIQUE} \iff (\overline{G}, |V| - k) \in VC$$

it is sufficient to prove

"$S \subseteq V$ is a clique in $G \iff V - S$ is a vertex cover in $\overline{G}$".

We prove this equivalence by proving subsequently the corresponding implications.

(i) Let $S$ be a clique in $G$. This implies that there is no edge between the vertices of $S$ in $\overline{G}$, i.e., every edge of $\overline{G}$ is incident to at least one vertex in $V - S$. Thus, $V - S$ is a vertex cover in $\overline{G}$.

(ii) Let $C \subseteq V$ be a vertex cover in $\overline{G}$. Following the definition of a vertex cover, every edge of $\overline{G}$ is incident to at least one vertex in $C$, i.e., there is no edge $\{u, v\} \in \overline{E}$ such that both $u$ and $v$ belong to $V - C$. So, $\{u, v\} \in E$ for all $u, v \in V - C$, $u \neq v$, i.e., $V - C$ is a clique in $G$. □

**Exercise 2.3.3.17.** Prove VC $\leq_p$ CLIQUE. □

**Exercise 2.3.3.18.** Prove 3SAT $\leq_p$ VC. □

The next reduction transforms the language of Boolean formulae into the language of linear equations.

**Lemma 2.3.3.19.** 3SAT $\leq_p$ SOL-0/1-LP.

*Proof.* Let $\Phi = F_1 \wedge F_2 \wedge \cdots \wedge F_m$ be a formula in 3CNF over the set of variables $X = \{x_1, \ldots, x_n\}$. Let $F_i = l_{i1} \vee l_{i2} \vee l_{i3}$ for $i = 1, \ldots, m$. First, we construct a system of linear inequalities over $X$ as follows. For every $F_i$, $i = 1, \ldots, m$, we take the linear inequality $LI_i$

$$z_{i1} + z_{i2} + z_{i3} \geq 1,$$

where $z_{ir} = x_k$ if $l_{ir} = x_k$ for some $k \in \{1, \ldots, n\}$, and $z_{ir} = (1 - x_q)$ if $l_{ir} = \overline{x}_q$ for some $q \in \{1, \ldots, n\}$. Obviously, $\varphi(F_i) = 1$ for an assignment $\varphi : X \to \{0,1\}$ if and only if $\varphi$ is a solution for the linear inequality $LI_i$. Thus, every $\varphi$ satisfying $\Phi$ is a solution of the system of linear inequalities $LI_1, LI_2, \ldots, LI_m$, and vice versa.

To get a system of linear equations we take $2m$ new Boolean variables (unknowns) $y_1, \ldots, y_m$, $w_1, \ldots, w_m$, and transform every $LI_i$ into the linear equation

$$z_{i1} + z_{i2} + z_{i3} - y_i - w_i = 1.$$

Clearly, the constructed system of linear equations has a solution if and only if the system of linear inequalities $LI_1, LI_2, \ldots, LI_m$ has a solution. □

**Exercise 2.3.3.20.** (*) Prove the following reducibilities.

(i) SAT $\leq_p$ SOL-0/1-LP,
(ii) 3SAT $\leq_p$ SOL-IP$_k$ for every prime $k$,
(iii) SOL-0/1-LP $\leq_p$ SAT,
(iv) SAT $\leq_p$ SOL-IP,
(v) CLIQUE $\leq_p$ SOL-0/1-LP. □

Above, a successful machinery for proving the hardness of decision problems under the assumption P $\subset$ NP has been introduced. We would like to have a method for proving a similar kind of hardness for optimization problems, too. To develop it, we first introduce the classes PO and NPO of optimization problems that are counterparts of the classes P and NP for decision problems.

**Definition 2.3.3.21. NPO** *is the class of optimization problems, where* $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal) \in$ NPO *if the following conditions hold:*

*(i)* $L_I \in$ P,
*(ii) there exists a polynomial* $p_U$ *such that*
  *a) for every* $x \in L_I$, *and every* $y \in \mathcal{M}(x)$, $|y| \leq p_U(|x|)$, *and*
  *b) there exists a polynomial-time algorithm that, for every* $y \in \Sigma_O^*$ *and every* $x \in L_I$ *such that* $|y| \leq p_U(|x|)$, *decides whether* $y \in \mathcal{M}(x)$, *and*
*(iii) the function cost is computable in polynomial time.*

Informally, we see that an optimization problem $U$ is in NPO if

(i) one can efficiently verify whether a string is an instance of $U$,
(ii) the size of the solutions is polynomial in the size of the problem instances and one can verify in polynomial time whether a string $y$ is a solution to any given input instance $x$, and
(iii) the cost of any solution can be efficiently determined.

Following the condition (ii), we see the relation to NP that can be seen as the class of languages accepted by polynomial-time verifiers. The conditions (i) and (iii) are natural because we are interested in problems whose kernel is the optimization and not the tractability of deciding whether a given input is a consistent problem instance or the tractability of the cost function evaluation.

We observe that the MAX-SAT problem is in NPO because

(i) one can check in polynomial time whether a word $x \in \Sigma^*_{logic}$ represents a Boolean formula $\Phi_x$ in CNF,
(ii) for every $x$, any assignment $\alpha \in \{0,1\}^*$ to the variables of $\Phi_x$ has the property $|\alpha| < |x|$ and one can verify whether $|\alpha|$ is equal to the number of variables of $\Phi_x$ even in linear time, and
(iii) for any given assignment $\alpha$ to the variables of $\Phi_x$, one can count the number of satisfied clauses of $\Phi_x$ in linear time.

**Exercise 2.3.3.22.** Prove that the following optimization problems are in NPO:

(i) MAX-CUT,
(ii) MAX-CL, and
(iii) MIN-VCP. □

The following definition corresponds to the natural idea of what tractable optimization problems are.

**Definition 2.3.3.23. PO** *is the class of optimization problems* $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ *such that*

*(i)* $U \in$ NPO, *and*
*(ii) there is a polynomial-time algorithm that, for every* $x \in L_I$, *computes an optimal solution for* $x$.

In what follows we present a simple method for introducing NP-hardness of optimization problems in the sense that if an NP-hard optimization problem would be in PO, than P would be equal to NP.

**Definition 2.3.3.24.** *Let* $U = (\Sigma_I, \Sigma_O, L, L_I, \mathcal{M}, cost, goal)$ *be an optimization problem from* NPO. *We define the* **threshold language of** $U$ *as*

$$\boldsymbol{Lang_U} = \{(x,a) \in L_I \times \Sigma^*_{bool} \mid Opt_U(x) \leq Number(a)\}$$

*if* $goal = minimum$, *and as*

$$\boldsymbol{Lang_U} = \{(x,a) \in L_I \times \Sigma^*_{bool} \mid Opt_U(x) \geq Number(a)\}$$

*if* $goal = maximum$.

*We say that* $U$ **is NP-hard** *if* $Lang_U$ *is* NP*-hard.*

The following lemma shows that to prove the NP-hardness of $Lang_U$ is really a way of showing that $U$ is hard for polynomial-time computations.

**Lemma 2.3.3.25.** *If an optimization problem $U \in$ PO, then $Lang_U \in$ P.*

*Proof.* If $U \in$ PO, then there is a polynomial-time algorithm $A$ that, for every input instance $x$ of $U$, computes an optimal solution for $x$ and so the value $Opt_U(x)$. Then, $A$ can be used to decide $Lang_U$. □

**Theorem 2.3.3.26.** *Let $U$ be an optimization problem. If $Lang_U$ is* NP*-hard and* P $\neq$ NP*, then $U \notin$* PO.

*Proof.* Assume the opposite, i.e., $U \in$ PO. Following Lemma 2.3.3.25 $Lang_U \in$ P. Since $Lang_U$ is NP-hard, $Lang_U \in$ P directly implies P = NP, a contradiction. □

To illustrate the simplicity of this method for proving the NP-hardness of optimization problems, we present the following examples.

**Lemma 2.3.3.27.** MAX-SAT *is* NP*-hard.*

*Proof.* Following Definition 2.3.3.23, we have to show that $Lang_{\text{MAX-SAT}}$ is NP-hard. Since we know that SAT is NP-hard, it is sufficient to prove SAT $\leq_p$ $Lang_{\text{MAX-SAT}}$. This reduction is straightforward. Let $x$ code a formula $\Phi_x$ of $m$ clauses. Thus, one takes $(x, m)$ as the input for a polynomial-time algorithm for $Lang_{\text{MAX-SAT}}$. Obviously, $(x, m) \in Lang_{\text{MAX-SAT}}$ iff $\Phi_x$ is satisfiable. □

**Lemma 2.3.3.28.** Max-CL *is* NP*-hard.*

*Proof.* Observe that CLIQUE = $Lang_{\text{MAX-CL}}$. Since we have already proved that CLIQUE is NP-hard, the proof is completed.

**Exercise 2.3.3.29.** Prove that the following optimization problems are NP-hard.

(i) MAX-3SAT,
(ii)(*) MAX-2SAT[46],
(iii) MIN-VCP,
(iv) SCP,
(v) SKP,
(vi) MAX-CUT,
(vii) TSP, and
(viii) MAX-E3LINMOD2. □

**Exercise 2.3.3.30.** Prove that P $\neq$ NP implies PO $\neq$ NPO. □

Keywords introduced in Section 2.3.3

uniform cost measurement, logarithmic cost measurement, worst case complexity, time complexity, space complexity, lower and upper bounds on problem complexity, complexity classes P, NP, PO, and NPO, verifiers, polynomial-time reduction, NP-hardness, NP-completeness, NP-hardness of optimization problems

[46] Observe that 2SAT $\in$ P

Summary of Section 2.3.3

Time complexity and space complexity are the fundamental complexity measures. Time complexity is measured as the number of elementary operations executed over computer words (operands of constant size). If one executes operations over unboundedly large operands, then one should consider the logarithmic cost measurement, where the cost of one operation is proportional to the length of the representation of the operands.

The (worst case) time complexity of an algorithm $A$ is a function $Time_A(n)$ of the input size. In $Time_A(n)$ complexity $A$ computes the output to each of the inputs of size $n$ and there is an input of size $n$ on which $A$ runs exactly with $Time_A(n)$ complexity.

The class P is the class of all languages that can be decided in polynomial time. Every decision problem in P is considered to be tractable (practically solvable). The class P is an invariant of the choice of any reasonable model of computation.

The class NP is the class of all languages that are accepted by polynomial-time nondeterministic algorithms, or, equivalently, by (deterministic) polynomial-time verifiers. One conjectures that $P \subset NP$, because if P would be equal to NP, then to find a solution would be as hard as to verify the correctness of a given solution for many mathematical problems. To prove or to disprove $P \subset NP$ is one of the most challenging open problems currently in computer science and mathematics.

The concept of NP-completeness provides a method for proving intractability (hardness) of specific decision problems, assuming $P \neq NP$. In fact, an NP-complete problem encodes any other problem from NP in some way, and, for any decision problem from NP, this encoding can be computed efficiently. NP-complete problems are considered to be hard, because if an NP-complete problem would be in P, then P would be equal to NP. This concept can be extended to optimization problems, too.

### 2.3.4 Algorithm Design Techniques

Over the years people identified several general techniques (concepts) that often yield efficient algorithms for large classes of problems. This chapter provides a short overview on the most important algorithm design techniques; namely,

divide-and-conquer,
dynamic programming,
backtracking,
local search, and
greedy algorithms.

We assume that the reader is familiar with all these techniques and knows many specific algorithms designed by them. Thus, we reduce our effort in this section to a short description of these techniques and their basic properties.

Despite the fact that these techniques are so successful that when getting a new algorithmic problem the most reasonable approach is to look whether one of these techniques alone can provide an efficient solution, none of these techniques alone can solve NP-hard problems. Later, we shall see that these methods combined with some new ideas and approaches can be helpful in designing practical algorithms for hard problems. But this is the topic of subsequent chapters.

In what follows we present the above mentioned methods for the design of efficient algorithms in a uniform way. For every technique, we start with its general description and continue with a simple illustration of its application.

### DIVIDE-AND-CONQUER.

Informally, the divide-and-conquer technique is based on breaking the given input instance into several smaller input instances in such a way that from the solutions to the smaller input instances one can easily compute a solution to the original input instance. Since the solutions to the smaller problem instances are computed in the same way, the application of the divide-and-conquer principle is naturally expressed by a recursive procedure. In what follows, we give a more detailed description of this technique. Let $U$ be any algorithmic[47] problem.

**Divide-and-Conquer Algorithm for $U$**

Input: An input instance $I$ of $U$, with a $size(I) = n$, $n \geq 1$.
Step 1: **if** $n = 1$ **then** compute the output to $I$ by any method
**else** continue with Step 2.
Step 2: Using $I$, derive problem instances $I_1, I_2, \ldots, I_k$, $k \in \mathbb{N} - \{0\}$ of $U$ such that $size(I_j) = n_j < n$ for $j = 1, 2, \ldots, k$.
{Usually, Step 2 is executed by partitioning $I$ into $I_1, I_2, \ldots, I_k$. $I_1, I_2, \ldots, I_k$ are also called **subinstances** of $I$.}
Step 3: Compute the output $U(I_1), \ldots, U(I_k)$ to the inputs (subinstances) $I_1, \ldots, I_k$, respectively, by recursively using the same procedure.
Step 4: Compute the output to $I$ from the outputs $U(I_1), \ldots, U(I_k)$.

In the standard case the complexity analysis results in solving a recurrence. Following our general schema, the time complexity of a divide-and-conquer algorithm $A$ may be computed as follows:

$$Time_A(1) \leq b \text{ if Step 1 can be done in time complexity } b \text{ for every input of } U \text{ of size 1,}$$

$$Time_A(n) = \sum_{i=1}^{k} Time_A(n_i) + g(n) + f(n)$$

[47] Decision problem, optimization problem, or any other problem

where $g(n)$ is the time complexity of partitioning of $I$ of size $n$ into subinstances $I_1, \ldots, I_k$ (Step 2), and $f(n)$ is the time complexity of computing $U(I)$ from $U(I_1), \ldots, U(I_k)$ (Step 4).

In almost all cases Step 2 partitions $I$ into $k$ input subinstances of the same size $\lceil \frac{n}{m} \rceil$. Thus, the typical recurrence has the form

$$Time_A(n) = k \cdot Time_A\left(\left\lceil \frac{n}{m} \right\rceil\right) + h(n)$$

for a nondecreasing function $h$ and some constants $k$ and $m$. How to solve such recurrences was discussed in Section 2.2.

Some famous examples of the divide-and-conquer technique are binary search, Mergesort, and Quicksort for sorting, and Strassen's matrix multiplication algorithm. There are numerous applications of this technique, and it is one of the most widely applicable algorithm design technique. Here, we illustrate it on the problem of long integer multiplication.

**Example (The problem of multiplying large integers)** Let

$$a = a_n a_{n-1} \ldots a_1 \text{ and } b = b_n b_{n-1} \ldots b_1$$

be binary representations of two integers $Number(a)$ and $Number(b)$, $n = 2^k$ for some positive integer $k$. The aim is to compute the binary representation of $Number(a) \cdot Number(b)$. Recall that the elementary school algorithm involves computing $n$ partial products of $a_n a_{n-1} \ldots a_1$ by $b$, for $i = 1, \ldots, n$, and so its complexity is in $O(n^2)$.

A naive divide-and-conquer approach can work as follows. One breaks each of $a$ and $b$ into two integers of $n/2$ bits each:

$$A = Number(a) = \underbrace{Number(a_n \ldots a_{n/2+1})}_{A_1} \cdot 2^{n/2} + \underbrace{Number(a_{n/2} \ldots a_1)}_{A_2}$$

$$B = Number(b) = \underbrace{Number(b_n \ldots b_{n/2+1})}_{B_1} \cdot 2^{n/2} + \underbrace{Number(b_{n/2} \ldots b_1)}_{B_2}.$$

The product of $Number(a)$ and $Number(b)$ can be written as

$$A \cdot B = A_1 \cdot B_1 \cdot 2^n + (A_1 \cdot B_2 + B_1 \cdot A_2) \cdot 2^{n/2} + A_2 \cdot B_2. \tag{2.28}$$

Designing a divide-and-conquer algorithm based on the equality (2.28) we see that the multiplication of two $n$-bit integers was reduced to

- four multiplications of $(\frac{n}{2})$-bit integers $(A_1 \cdot B_1, A_1 \cdot B_2, B_1 \cdot A_2, A_2 \cdot B_2)$,
- three additions of integers with at most $2n$ bits, and
- two shifts (multiplications by $2^n$ and $2^{n/2}$).

Since these additions and shifts can be done in $cn$ steps for some suitable constant $c$, the complexity of this algorithm is given by the following recurrence:

$$Time(1) = 1$$
$$Time(n) = 4 \cdot Time\left(\frac{n}{2}\right) + cn. \tag{2.29}$$

Following the Master Theorem, the solution of (2.29) is $Time(n) = O(n^2)$. This is no improvement of the classical school method from the asymptotic point of view. To get an improvement one needs to decrease the number of subproblems, i.e., the number of multiplications of $(n/2)$-bit integers. This can be done with the following formula

$$A \cdot B = A_1B_1 \cdot 2^n + [A_1B_1 + A_2B_2 + (A_1 - A_2) \cdot (B_2 - B_1)] \cdot 2^{n/2} + A_2B_2 \tag{2.30}$$

because

$$\begin{aligned}&(A_1 - A_2) \cdot (B_2 - B_1) + A_1B_1 + A_2B_2\\ &= A_1B_2 - A_1B_1 - A_2B_2 + A_2B_1 + A_1B_1 + A_2B_2\\ &= A_1B_2 + A_2B_1.\end{aligned}$$

Although (2.30) looks more complicated than (2.28), it requires only

- three multiplications of $\left(\frac{n}{2}\right)$-bit integers ($A_1 \cdot B_1$, $(A_1 - A_2) \cdot (B_2 - B_1)$, and $A_2 \cdot B_2$),
- four additions, and two subtractions of integers of at most $2n$ bits, and
- two shifts (multiplications by $2^n$ and $2^{n/2}$).

Thus, the divide-and-conquer algorithm $C$ based on (2.30) has the time complexity given by the recurrence

$$Time_C(1) = 1$$
$$Time_C(n) = 3 \cdot Time_C\left(\frac{n}{2}\right) + dn \tag{2.31}$$

for a suitable constant $d$. According to the Master Theorem (Section 2.2.2) the solution of (2.31) is $Time_C(n) \in O\left(n^{\log_2 3}\right)$, where $\log_2 3 \approx 1.59$. So, $C$ is asymptotically faster than the school method.[48] □

## DYNAMIC PROGRAMMING.

The similarity between divide-and-conquer and dynamic programming is that both these approaches solve problems by combining the solutions to problem subinstances. The difference is that divide-and-conquer does it recursively by dividing a problem instance into subinstances and calling itself on these problem subinstances,[49] while dynamic programming works in a bottom-up

[48] Note that the school method is superior to $C$ for integers with fewer than 500 bits because the constant $d$ is too large.

[49] Thus, divide-and-conquer is a top-down method.

fashion as follows. It starts by computing solutions to the smallest (simplest) subinstances, and continues to larger and larger subinstances until the original problem instance has been solved. During its work, any algorithm based on dynamic programming stores all solutions to problem subinstances in a table. Thus, a dynamic-programming algorithm solves every problem subinstance just once because the solutions are saved and reused every time the subproblem is encountered. This is the main advantage of dynamic programming over divide-and-conquer. The divide-and-conquer method can result in solving exponentially many problem subinstances despite the fact that there is only a polynomial number of different problem subinstances. This means that it may happen that a divide-and-conquer algorithm solves some subinstance several times.[50]

Perhaps the most transparent example for the difference between dynamic programming and divide-and-conquer can be seen when computing the $n$th Fibonacci number $F(n)$. Recall that

$$F(1) = F(2) = 1 \text{ and } F(n) = F(n-1) + F(n-2) \text{ for } n \geq 3.$$

The dynamic-programming algorithm $A$ subsequently computes

$$\text{“}F(1),\ F(2),\ F(3) = F(1) + F(2),\ \ldots,\ F(n) = F(n-1) + F(n-2)\text{”}.$$

Obviously, $Time_A$ is linear in the value of $n$. A divide-and-conquer algorithm DCF on the input $n$ will recursively call DCF$(n-1)$ and DCF$(n-2)$. DCF$(n-1)$ will again recursively call DCF$(n-2)$ and DCF$(n-3)$, etc. The tree in Figure 2.18 depicts a part of the recursive calls of DCF. One easily observes that the number of subproblems calls is exponential in $n$ and so $Time_{\text{DCF}}$ is exponential in $n$.

**Exercise 2.3.4.1.** Estimate, for every $i \in \{1, 2, \ldots, n-1\}$, how many times the subproblem $F(n-i)$ is solved by DCF. □

**Exercise 2.3.4.2.** Consider the problem of computing $\binom{n}{k}$ with the formula

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}. \tag{2.32}$$

What is the difference between the time complexities of divide-and-conquer and dynamic programming when both methods use the formula (2.32) to compute the result? (Note that the complexity has to be measured in both input parameters $n$ and $k$.) □

Some of the well-known applications of dynamic programming are the Floyd algorithm for the shortest path problem, the algorithm for the minimum

[50] Note that it sometimes happens that the natural way of dividing an input instance leads to overlapping subinstances and so the number of different subinstances may even be exponential.

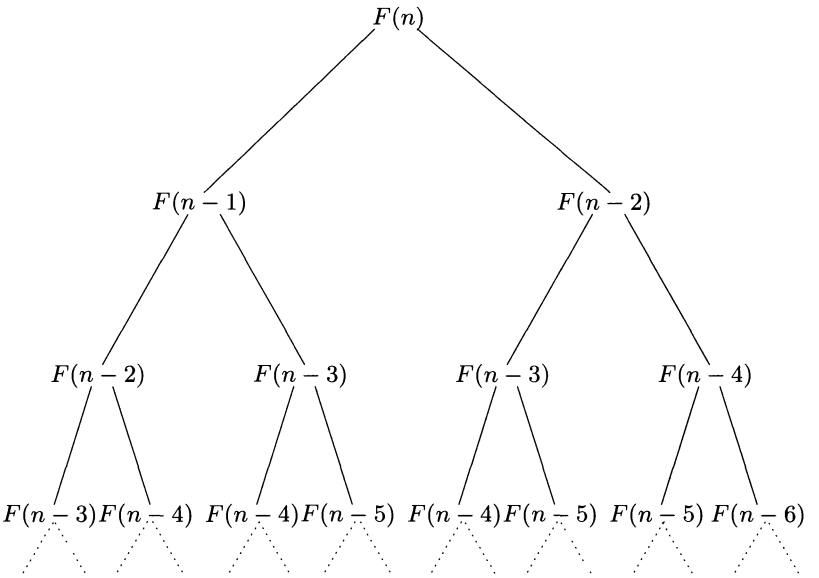



**Fig. 2.18.**

triangulation problem, the optimal merge pattern algorithm, and the pseudo-polynomial algorithm for the knapsack problem (presented in Chapter 3). We illustrate the method of dynamic programming with the Floyd algorithm.

**Example** The all-pairs shortest path problem is to find the costs of the shortest path between any pair of vertices of a given weighted graph $(G, c)$, where $G = (V, E)$ and $c : E \to \mathbb{N} - \{0\}$. Let $V = \{v_1, v_2, \ldots, v_n\}$.

The idea is to consecutively compute the values

$$Cost_k(i,j) = \text{the cost of the shortest path between } v_i \text{ and } v_j \text{ whose internal vertices are from } \{v_1, v_2, \ldots, v_k\}$$

for $k = 0, 1, \ldots, n$. At the beginning one sets

$$Cost_0(i,j) = \begin{cases} c(\{v_i, v_j\}) & \text{if } \{v_i, v_j\} \in E, \\ \infty & \text{if } \{v_i, v_j\} \notin E \text{ and } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

To compute $Cost_k(i,j)$ from $Cost_{k-1}(r,s)$'s for $r, s \in \{1, \ldots, n\}$ one can use the formula

$$Cost_k(i,j) = \min\{Cost_{k-1}(i,j), Cost_{k-1}(i,k) + Cost_{k-1}(k,j)\}$$

for all $i, j \in \{1, 2, \ldots, n\}$. This is because the shortest path between $v_i$ and $v_j$ going via the vertices from $\{v_1, \ldots, v_k\}$ only, either does not go via the vertex $v_k$ or if it goes via $v_k$ then it visits $v_k$ exactly once.[51]

[51] The shortest path does not contain cycles because all costs on edges are positive.

The following algorithm is the straightforward implementation of this strategy.

**FLOYD'S ALGORITHM**

Input: A graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, $n \in \mathbb{N} - \{0\}$, and a cost function $c : E \to \mathbb{N} - \{0\}$.

Step 1: **for** $i = 1$ **to** $n$ **do**
**do begin** $Cost[i, i] := 0$;
**for** $j := 1$ **to** $n$ **do**
**if** $\{v_i, v_j\} \in E$ **then** $Cost[i, j] := c(\{v_i, v_j\})$
**else if** $i \neq j$ **then** $Cost[i, j] := \infty$
**end**

Step 2: **for** $k := 1$ **to** $n$ **do**
**for** $i := 1$ **to** $n$ **do**
**for** $j := 1$ **to** $n$ **do**
$Cost[i, j] := \min\{Cost[i, j], Cost[i, k] + Cost[k, j]\}$.

Obviously, the complexity of the FLOYD'S ALGORITHM is in $O(n^3)$. □

**BACKTRACKING.**

Backtracking is a method for solving optimization problems by a possibly exhaustive search of the set of all feasible solutions or for determining an optimal strategy in a finite game by a search in the set of all configurations of the game. Here, we are interested only in the application of backtracking for optimization problems.

In order to be able to apply the backtrack method for an optimization problem one needs to introduce some structure in the set of all feasible solutions. If the specification of every feasible solution can be viewed as an $n$-tuple $(p_1, p_2, \ldots, p_n)$, where every $p_i$ can be chosen from a finite set $P_i$, then the following way brings a structure into the set of all feasible solutions.

Let $\mathcal{M}(x)$ be the set of all feasible solutions to the input instance $x$ of an optimization problem.

We define $\boldsymbol{T_{\mathcal{M}(x)}}$ as a labeled rooted tree with the following properties:

(i) Every vertex $v$ of $T_{\mathcal{M}(x)}$ is labeled by a set $S_v \subseteq \mathcal{M}(x)$.
(ii) The root of $T_{\mathcal{M}(x)}$ is labeled by $\mathcal{M}(x)$.
(iii) If $v_1, \ldots, v_m$ are all sons of a father $v$ in $T_{\mathcal{M}(x)}$, then $S_v = \bigcup_{i=1}^{m} S_{v_i}$ and $S_{v_i} \cap S_{v_j} = \emptyset$ for $i \neq j$.
{The sets corresponding to the sons define a partition of the set of their father.}
(iv) For every leaf $u$ of $T_{\mathcal{M}(x)}$, $|S_u| \leq 1$.
{The leaves correspond to the feasible solutions of $\mathcal{M}(x)$.}

If every feasible solution can be specified as described above, one can start to build $T_{\mathcal{M}(x)}$ by setting $p_1 = a$. Then the left son[52] of the root corresponds to the set of feasible solutions with $p_1 = a$ and the right son of the root corresponds to the set of feasible solutions with $p_1 \neq a$. Continuing with this strategy the tree $T_{\mathcal{M}(x)}$ can be constructed in the straightforward way.[53] Having the tree $T_{\mathcal{M}(x)}$ the backtrack method is nothing else than a search (the depth-first-search or the breadth-first-search) in $T_{\mathcal{M}(x)}$. In fact, one does not implement backtracking as a two-phase algorithm, where $T_{\mathcal{M}(x)}$ is created in the first phase, and the second phase is the depth-first-search in $T_{\mathcal{M}(x)}$. This approach would require a too large memory. The tree $T_{\mathcal{M}(x)}$ is considered only hypothetically and one starts the depth-first-search in it directly. Thus, it is sufficient to save the path from the root to the actual vertex only.

In the following example we illustrate the backtrack method on the TSP problem.

**Example 2.3.4.3 (Backtracking for TSP).** Let $x = (G, c)$, $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$, $E = \{e_{ij} \mid i, j \in \{1, \ldots, n\}, i \neq j\}$ be an input instance of TSP. Any feasible solution (a Hamiltonian tour) to $(G, c)$ can be unambiguously specified by an $(n-1)$-tuple $(\{v_1, v_{i_1}\}, \{v_{i_1}, v_{i_2}\}, \ldots, \{v_{i_{n-2}}, v_{i_{n-1}}\}) \in E^{n-1}$, where $\{1, i_1, i_2, \ldots, i_{n-1}\} = \{1, 2, \ldots, n\}$ (i.e., $v_1, v_{i_1}, v_{i_2}, \ldots, v_{i_{n-1}}, v_1$ is the Hamiltonian cycle that consists of edges $e_{1i_1}, e_{i_1 i_2}, \ldots, e_{i_{n-2} i_{n-1}}, e_{i_{n-1}1}$. Let, in what follows, $S_x(h_1, \ldots, h_r, \overline{k}_1, \ldots, \overline{k}_s)$ denote the subset of feasible solutions containing the edges $h_1, \ldots, h_r$ and not containing any of the edges $k_1, \ldots, k_s$, $r, s \in \mathbb{N}$. $T_{\mathcal{M}(x)}$ can be created by dividing $\mathcal{M}(x)$ into $S_x\{e_{12}\} \subseteq \mathcal{M}(x)$ that consists of all feasible solutions that contain the edge $e_{12}$ and $S_x(\overline{e}_{12})$ consisting of all Hamiltonian tours that do not contain the edge $e_{12}$, etc. Next, we construct $T_{\mathcal{M}(x)}$ for the input instance $x = (G, c)$. $T_{\mathcal{M}(x)}$ is depicted in Figure 2.20. We set $S_x(\emptyset) = \mathcal{M}(x)$.

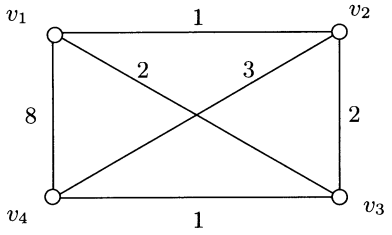



**Fig. 2.19.**

Let us show that all leaves of $T_{\mathcal{M}(x)}$ are one-element subsets of $\mathcal{M}(x)$. $S_x(e_{12}, e_{23}) = \{(e_{12}, e_{23}, e_{34}, e_{41})\}$ because the only possibility of closing a

[52] $T_{\mathcal{M}(x)}$ does not necessarily need to be a binary tree, i.e., one may use another strategy to create it.

[53] A specific construction is done in the following example.

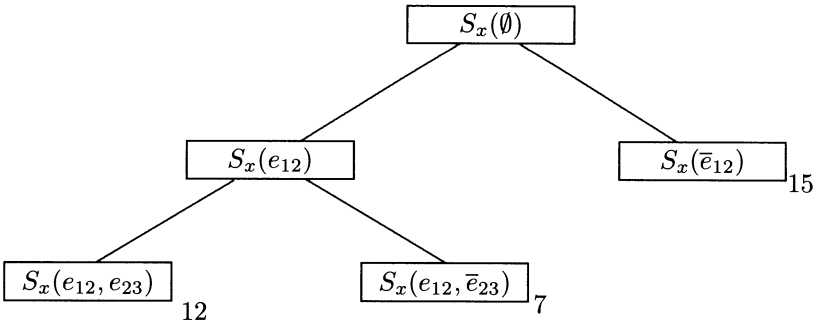



**Fig. 2.20.**

Hamiltonian tour starting with $v_1, v_2, v_3$ is to continue with $v_4$ and then finally with $v_1$. The cost of this tour is 12. Every Hamiltonian tour that contains $e_{12}$ and does not contain $e_{23}$ must contain $e_{24}$. So, $S_x(e_{12}, \bar{e}_{23}) = \{e_{12}, e_{24}, e_{43}, e_{31}\}$, and the corresponding Hamiltonian tour $v_1, v_2, v_4, v_3, v_1$ has cost 7. $S_x(\bar{e}_{12}) = \{(e_{13}, e_{23}, e_{24}, e_{14})\} = S_x\{e_{13}, e_{14}\}$ because if $e_{12}$ is not in a Hamiltonian tour, then $e_{13}$ and $e_{14}$ must be in this tour.[54] Thus, $v_4, v_1, v_3$ (or $v_3, v_1, v_4$) must be part of the Hamiltonian tour. Because only the vertex $v_2$ is missing, one has to take edge $e_{24}$ and $e_{23}$ and the resulting Hamiltonian tour is $v_4, v_1, v_3, v_2, v_4 = v_1, v_3, v_2, v_4, v_1$. Its cost is 15. □

We shall present a more complex example in Section 3.4, where the use of the backtrack method for solving hard problems will be discussed.

One could ask why we do not simply use the brute force approach by enumerating all $|P_1| \cdot |P_2| \cdot \cdots \cdot |P_{n-1}|$ [$(n-1)^{n-1}$ many in the case of TSP] tuples $(p_1, \ldots, p_{n-1})$ and to look at the costs of them that specify a Hamiltonian tour. There are at least two possible advantages of backtracking over the brute force method. First of all we generate only feasible solutions by backtracking and so their number may be essentially smaller than $|P_1| \cdot \cdots \cdot |P_{n-1}|$. For instance, we have at most $(n-1)!/2$ Hamiltonian tours in a graph of $n$ vertices but the brute force method would generate $(n-1)^{(n-1)}$ tuples candidating for a feasible solution. The main advantage is that we do not necessarily need to realize a complete search in $T_{\mathcal{M}(x)}$. If we find a feasible solution $\alpha$ with a cost $m$, and if we can calculate in some vertex $v$ of $T_{\mathcal{M}(x)}$ that all solutions corresponding to the subtree $T_v$ rooted by $v$ cannot be better than $\alpha$ (their costs are larger than $m$ if one considers a minimization problem), then we can omit the search in $T_v$. If one is lucky, the search in many subtrees may be omitted and the backtrack method becomes more efficient. But we stop the discussion on this topic here, because making backtracking more efficient in the applica-

[54] Note that, for every vertex $v$ of $G$, there are two edges adjacent to $v$ in any Hamiltonian tour of $G$.

tions for hard optimization problems is the main topic of Section 3.4 in the next chapter.

**LOCAL SEARCH.**

Local search is an algorithm design technique for optimization problems. In contrast to backtracking we do not try to execute an exhaustive search of the set of feasible solutions, but rather a restricted search in the set of feasible solutions. If one wants to design a local search algorithm it is necessary to start by defining a structure on the set of all feasible solutions. The usual way to do it is to say that two feasible solutions $\alpha$ and $\beta$ are neighbors if $\alpha$ can be obtained from $\beta$ and vice versa by some local (small) change of their specification. Such local changes are usually called **local transformations**. For instance, two Hamiltonian tours $H_1$ and $H_2$ are neighbors if one can obtain $H_2$ from $H_1$ by exchanging two edges of $H_1$ for another two edges. Obviously, any neighborhood on $\mathcal{M}(x)$ is a relation on $\mathcal{M}(x)$. One can view $\mathcal{M}(x)$ as a graph $G(\mathcal{M}(x))$ whose vertices are feasible solutions, and two solutions $\alpha$ and $\beta$ are connected by the edge $\{\alpha, \beta\}$ if and only if $\alpha$ and $\beta$ are neighbors. The local search is nothing else but a search in $G(\mathcal{M}(x))$ where one moves via an edge $\{\alpha, \beta\}$ from $\alpha$ to $\beta$ only if $cost(\beta) > cost(\alpha)$ for maximization problems and $cost(\beta) < cost(\alpha)$ for minimization problems. Thus, local search can be viewed as an iterative improvement that halts with a feasible solution that cannot be improved by the movement to any of its neighbors. In this sense the produced outputs of local search are local optima of $\mathcal{M}(x)$.

If the neighborhood is defined, one can briefly describe local search algorithms by the following scheme.

Input: An input instance $x$.
Step 1: Start with a (randomly chosen) feasible solution $\alpha$ from $\mathcal{M}(x)$.
Step 2: Replace $\alpha$ with a neighbor of $\alpha$ whose cost is an improvement in the comparison with the cost of $\alpha$.
Step 3: Repeat Step 2 until a solution $\alpha$ is reached such that no neighbor of $\alpha$ can be viewed as a better solution than $\alpha$.
Output: $\alpha$.

The local search technique is usually very efficient and it is one of the most popular methods for attacking hard optimization problems. Besides this, some successful heuristics like simulated annealing are based on this technique. Because of this we postpone deeper discussion and analysis of the local search technique to Chapters 3 and 6, which focus on the topic of this book. We finish the presentation of this method here with a simple application example.

**Example 2.3.4.4. A local search algorithm for the minimum spanning tree problem**

The minimum spanning tree problem is to find, for a given weighted graph $G = (V, E, c)$, a tree $T = (V, E')$, $E' \subseteq E$, with minimal cost. Let

$T_1 = (V, E_1)$ and $T_2 = (V, E_2)$ be two spanning trees of $G$. We say that $T_1$ and $T_2$ are neighbors if $T_1$ can be obtained from $T_2$ by exchanging one edge (i.e., $|E_1 - E_2| = |E_2 - E_1| = 1$). Observe that in looking for neighbors one cannot exchange arbitrary edges because this can destroy the connectivity. The simplest way to do a consistent transformation is to add a new edge $e$ to $T_1 = (V, E_1)$. After this we obtain the graph

$$(V,\ E_1 \cup \{e\})$$

that contains exactly one cycle since $T_1$ is a spanning tree. If one removes an edge $h$ of the cycle with $c(h) > c(e)$, then one obtains a better spanning tree

$$(V, (E_1 \cup \{e\}) - \{h\})$$

of $G$ which is a neighbor of $T_1$.

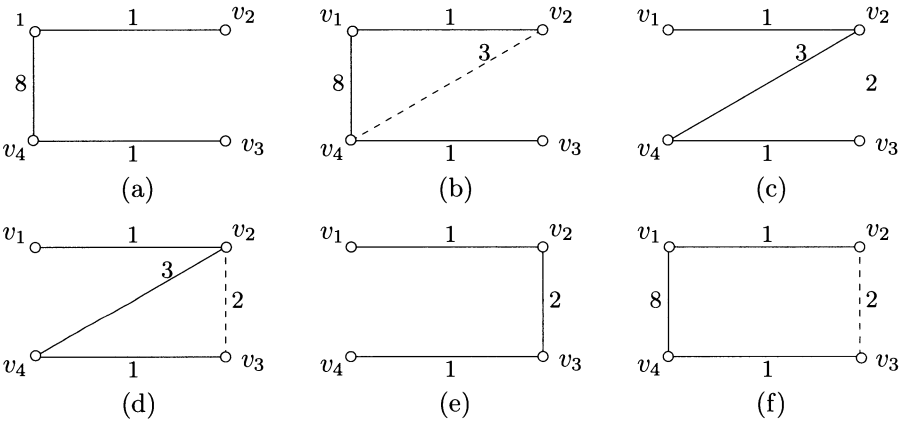



**Fig. 2.21.**

Consider the graph $G = (V, E, c)$ depicted in Figure 2.19. We start the local search with the tree depicted in Figure 2.21a. Its cost is 10. Let us add the edge $\{v_2, v_4\}$ first. Then, we get the cycle $v_1, v_2, v_4, v_1$ (Figure 2.21b). Removing the most expensive edge $\{v_1, v_4\}$ of this cycle we get the spanning tree in Figure 2.21c with cost 5. Now, one adds the edge $\{v_2, v_3\}$ (Figure 2.21d). Removing $\{v_2, v_4\}$ from the cycle $v_2, v_3, v_4, v_2$ we obtain an optimal spanning tree (Figure 2.21e) whose cost is 4. The solution can be also obtained in one iterative step if one would add the edge $\{v_2, v_3\}$ in the first iterative step and then remove the edge $\{v_1, v_4\}$ (Figure 2.21f). □

**Exercise 2.3.4.5.** Prove that the local search algorithm for the minimum spanning tree always computes an optimal solution in time $O\left(|E|^2\right)$. □

**GREEDY ALGORITHMS.**

The greedy method is perhaps the most straightforward algorithm design technique for optimization problems. A similarity to backtracking and to local algorithms is in that one needs a specification of a feasible solution by a tuple $(p_1, p_2, \ldots, p_n)$, $p_i \in P_i$ for $i = 1, \ldots, n$, and that any greedy algorithm can be viewed as a sequence of local steps. But the greedy algorithms do not move from one feasible solution to another feasible solution. They start with an empty specification and fix one local parameter of the specification (for instance, $p_2$) forever. In the second step a local algorithm fixes a second parameter (for instance, $p_1$) of the specification and so on until a complete specification of a feasible solution is reached. The name greedy comes from the way in which the decisions about local specifications are done. A greedy algorithm chooses the parameter that seems to be most promising from all possibilities to make the next local specification. It never reconsiders its decision, whatever situation may arise later. For instance, a greedy algorithm for TSP starts by deciding that the cheapest edge must be in the solution. This is locally the best choice if one has to specify only one edge of a feasible solution. In the next steps it always adds the cheapest new edge that can, together with the already fixed edges, form a Hamiltonian tour to the specification.

Another point about greedy algorithms is that they realize exactly one path from the root to a leaf in the tree $T_{\mathcal{M}(x)}$ created by backtracking. In fact an empty specification means that one considers the set of all feasible solutions $\mathcal{M}(x)$ and $\mathcal{M}(x)$ is the label of the root of the tree $T_{\mathcal{M}(x)}$. Specifying the first parameter $p_1$ corresponds to restricting $\mathcal{M}$ to a set $S(p_1) = \{\alpha \in \mathcal{M}(x) \mid$ the first parameter of the specification of $\alpha$ is $p_1\}$. Continuing this procedure we obtain the sequence of sets of feasible solutions

$$\mathcal{M}(x) \supseteq S(p_1) \supseteq S(p_1, p_2) \supseteq S(p_1, p_2, p_3) \supseteq \cdots \supseteq S(p_1, p_2, \ldots, p_n),$$

where $|S(p_1, p_2, \ldots, p_n)| = 1$.

Following the above consideration we see that greedy algorithms are usually very efficient, especially in the comparison with backtracking. Another advantage of the greedy method is that it is easy to invent and easy to implement. The drawback of the greedy method is that too many optimization problems are too complex to be solved by such a naive strategy. On the other hand even this simple approach can be helpful in attacking hard problems. Examples documenting this fact will be presented in Chapter 4. The following two examples show that the greedy method can efficiently solve the minimum spanning tree problem and that it can be very weak for TSP.

**Example 2.3.4.6 (Greedy for the minimum spanning tree problem).** The greedy algorithm for the minimum spanning tree problem can be simply described as follows.

**GREEDY-MST**

Input: A weighted connected graph $G = (V, E, c)$, $c : E \to \mathbb{N} - \{0\}$.
Step 1: Sort the edges according to their costs. Let $e_1, e_2, \ldots, e_m$ be the sequence of all edges of $E$ such that $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$.
Step 2: Set $E' := \{e_1, e_2\}$; $I := 3$;
Step 3: **while** $|E'| < |V| - 1$ **do**
**begin** add $e_I$ to $E'$ if $(V, E' \cup \{e_I\})$ does not contain any cycle;
$I := I + 1$
**end**
Output: $(V, E')$.

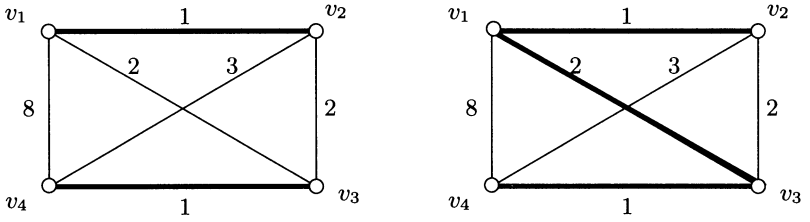



**Fig. 2.22.**

Since $|E'| = |V| - 1$ and $(V, E')$ does not contain any cycle, it is obvious that $(V, E')$ is a spanning tree of $G$. To prove that $(V, E')$ is an optimal spanning tree is left to the reader.

We illustrate the work of GREEDY-MST on the input instance $G$ depicted in Figure 2.19. Let $\{v_1, v_2\}, \{v_3, v_4\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_1, v_4\}$ be the sequence of the edges after sorting in Step 1. Then Figure 2.22 depicts the steps of the specification of the optimal solution $(V, \{\{v_1, v_2\}, \{v_3, v_4\}, \{v_1, v_3\}\})$.

□

**Exercise 2.3.4.7.** Prove that GREEDY-MST always computes an optimal solution. □

**Example 2.3.4.8 (Greedy for TSP).** The greedy algorithm for TSP can be described as follows.

**GREEDY-TSP**

Input: A weighted complete graph $G = (V, E, c)$ with $c : E \to \mathbb{N} - \{0\}$, $|V| = n$ for some positive integer $n$.
Step 1: Sort the costs of the edges. Let $e_1, e_2, \ldots, e_{\binom{n}{2}}$ be the sequence of all edges of $G$ such that $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_{\binom{n}{2}})$.
Step 2: $E' = \{e_1, e_2\}$, $I := 3$;

Step 3: **while** $|E'| < n$ **do**
**begin** add $\{e_I\}$ to $E'$ if $(V, E' \cup \{e_I\})$ does not contain any vertex of degree greater than 2 and any cycle of length shorter than $n$;
$I := I + 1$;
**end**
Output: $(V, E')$.

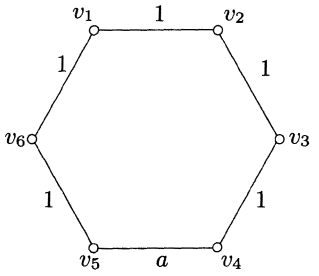



**Fig. 2.23.**

Since $|E'| = n$ and no vertex has degree greater than 2, $(V, E')$ is a Hamiltonian tour. Considering the graph partially depicted in Figure 2.23 we see that GREEDY-TSP may produce solutions whose costs are arbitrarily far from the optimal cost.

Let the edges $\{v_i, v_{i+1}\}$ for $i = 1, 2, \ldots, 5$ and $\{v_1, v_6\}$ have the costs as depicted in Figure 2.23 and let all missing edges have the cost 2. Assume that $a$ is a very large number. Obviously, GREEDY-TSP takes first all edges of cost 1. Then the Hamiltonian tour $v_1, v_2, v_3, v_4, v_5, v_6, v_1$ is unambiguously determined and its cost is $a + 5$. Figure 2.24 presents an optimal solution $v_1, v_4, v_3, v_2, v_5, v_6, v_1$ whose cost is 8.

Since one can choose an arbitrarily large $a$, the difference between $a + 5$ and 8 can be arbitrarily large. □

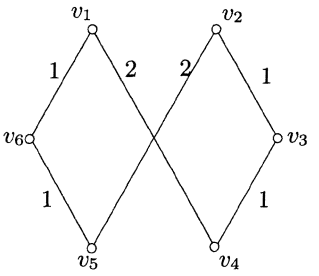



**Fig. 2.24.**

Keywords introduced in Section 2.3.4

divide-and-conquer, dynamic programming, backtracking, local search, greedy algorithms

Summary of Section 2.3.4

Divide-and-conquer, dynamic programming, backtracking, local search, and greedy algorithms are fundamental algorithm design techniques. These techniques are robust and paradigmatic in the sense that, when getting a new algorithm problem, the most reasonable approach is to look whether one of these techniques alone can provide an efficient solution.

Divide-and-conquer is a recursive technique based on breaking the given problem instance into several problem subinstances in such a way that from the solution to the smaller problem instances one can easily compute a solution to the original problem instance.

Dynamic programming is similar to the divide-and-conquer method in the sense that both techniques solve problems by combining the solutions to subproblems. The difference is that divide-and-conquer does it recursively by dividing problem instances into subinstances and calling itself on these subinstances, while dynamic programming works in a bottom-up fashion by starting with computing solutions to smallest subinstances and continuing to larger and larger subinstances until the original problem instance is solved. The main advantage of dynamic programming is that it solves every subinstance exactly once, while divide-and-conquer may compute a solution to the same subinstance many times.

Backtracking is a technique for solving optimization problems by a possibly exhaustive search of the set of all feasible solutions, in such a systematic way that one never looks twice at the same feasible solution.

Local search is an algorithm design technique for optimization problems. The idea is to define a neighborhood in the set of all feasible solutions $\mathcal{M}(x)$ and then to search in $\mathcal{M}(x)$ going from a feasible solution to a neighboring feasible solution if the cost of the neighboring solution is better than the cost of the original solution. A local search algorithm stops with a feasible solution that is a local optimum according to the defined neighborhood.

Greedy method is based on a sequence of steps, where in every step the algorithm specifies one parameter of a feasible solution. The name greedy comes from the way it chooses of the parameters. Always, the most promising choice from all possibilities is taken to specify the next parameter, and no decision is reconsidered later.