Stefan Hougardy
Jens Vygen

Algorithmic Mathematics

Springer

Algorithmic Mathematics

Stefan Hougardy • Jens Vygen

Algorithmic Mathematics

Springer

Stefan Hougardy
Research Institute for Discrete Mathematics
University of Bonn
Bonn, Germany

Jens Vygen
Research Institute for Discrete Mathematics
University of Bonn
Bonn, Germany

Translated by Rabe von Randow, formerly University of Bonn, Germany

ISBN 978-3-319-39557-9 ISBN 978-3-319-39558-6 (eBook)
DOI 10.1007/978-3-319-39558-6

Library of Congress Control Number: 2016955863

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Since the advent of computers, the significance of algorithms has risen steadily in all fields of mathematics. At the University of Bonn this has led to the introduction of a new course of lectures for beginning students, alongside the two basic courses Analysis and Linear Algebra, namely Algorithmic Mathematics. This book comprises the content of this new course which the authors have taught several times and which consists of around 30 lectures of 90 min each, plus exercise tutorials. While the book assumes no more than high school knowledge, it is challenging for readers without mathematical experience.

In contrast to most other introductory texts on algorithms which are probably intended predominantly for computer science students, our emphasis is on a strict and rigorous mathematical presentation. Exact definitions, precise theorems, and carefully executed elegant proofs are in our view indispensable, particularly at the beginning of mathematical studies. Moreover, the book contains many worked examples, explanations, and references for further study.

Our choice of themes reflects our intention to present as wide a spectrum of algorithms and algorithmic problems as possible without a deeper knowledge of mathematics. We treat basic concepts (Chaps. 1, 2, and 3), numerical problems (Chaps. 4 and 5), graphs (Chaps. 6 and 7), sorting algorithms (Chap. 8), combinatorial optimization (Chaps. 9 and 10), and Gaussian elimination (Chap. 11). As themes are often interrelated, the order of the chapters cannot be changed without possibly having to refer to later sections. The reader will be introduced not only to the classic algorithms and their analysis but also to important theoretical foundations and will discover many cross-references and even unsolved research problems.

One cannot really understand algorithms and work with them without being able to implement them as well. Thus we have, alongside the mathematical themes, included an introduction to the programming language C++ in this book. In doing so we have, however, endeavored to restrict the technical details to the necessary minimum—this is not a programming course!—while still presenting the student lacking programming experience with an introductory text.

Our carefully designed programming examples have a twofold purpose: first to illustrate the main elements of C++ and also to motivate the student to delve further, and second to supplement the relevant theme. Clearly one cannot become a versatile programmer without actually doing it oneself, just as one cannot learn mathematics properly without doing exercises and solving problems oneself. One cannot emphasize this too strongly and we encourage all beginning students to do so.

It is our sincere wish that all our readers will enjoy studying Algorithmic Mathematics!

Bonn, Germany
March 2015

Stefan Hougardy
Jens Vygen

Remarks on the C++ Programs

This book contains a number of programming examples formulated in C++. The source code of all these programs can be downloaded from the websites of the authors. In this book we have used the C++ version specified in ISO/IEC 14882:2011 [5], which is also known as C++11. In order to compile these programming examples, one can use all the usual C++ compilers that support this C++ version. For example, the freely available GNU C++ Compiler g++ from version 4.8.1 onwards supports all the language elements of C++11 used in this book. Examples of good textbooks on C++11 are [25, 32]. Detailed information on C++11 is also available on the internet, see for example http://en.cppreference.com or http://www.cplusplus.com.

Acknowledgments

We would like to take the opportunity to thank all those who have helped us over the years with suggestions and improvements pertaining to this book. Along with the students taking our courses there are those that deserve special mention: Christoph Bartoschek, Ulrich Brenner, Helmut Harbrecht, Stephan Held, Dirk Müller, Philipp Ochsendorf, Jan Schneider, and Jannik Silvanus. To all of them we extend our warm thanks.

We will continue to be very grateful for being made aware of any remaining mistakes or of further suggestions for improvement.

Bonn, Germany

Stefan Hougardy
Jens Vygen

Contents

Index of Symbols

$:$	with the property that
$\exists$	there exists
$\forall$	for all
$\emptyset$	empty set
$\subseteq$	subset
$\subset$	proper subset
$\cup$	union of sets
$\cap$	intersection of sets
$\dot{\cup}$	disjoint union of sets
$\triangle$	symmetric difference of sets
$\times$	Cartesian product
$\wedge$	logical and
$\vee$	logical or
$\lceil . \rceil$	ceiling
$\lfloor . \rfloor$	floor
$\approx$	approximately equal
$\leftarrow$	assignment in pseudocode
$^\top$	transpose

1 Introduction

In this chapter we will define a number of basic concepts and present several simple algorithms which we will analyze and implement in C++. We will also see that not everything is computable.

1.1 Algorithms

The word algorithm is used to denote a finite prescription with the following specifications:

- what is accepted as input for the algorithm?
- which computational steps, possibly depending on the input and/or on intermediate results, will be performed and in which order?
- when does the algorithm terminate and what will then be the output?

In order to give a formal definition of an algorithm one requires a concrete computer model, e.g. the Turing machine [35], or a programming language, e.g. C++ [31]. We will not pursue this matter here at this point but will instead consider some examples of algorithms.

The origin of the word algorithm goes back to Muhammad ibn Musa Al-Khwarizmi (ca. 780–840), whose book on the Indian number system and reckoning methods (addition, subtraction, multiplication, division, working with fractions, computing roots) revolutionized occidental mathematics; see [12, 36].

There are, however, some much older algorithms; the Sieve of Eratosthenes, the Euclidean Algorithm and Gaussian Elimination, for example, were described in literature over 2000 years ago. We will discuss these algorithms in detail below.

Algorithms can be performed by people, or they can be implemented in hardware (formerly in mechanical calculating machines, today on computer chips) or in software, i.e. written in some programming language as a computer program which can then be translated and run on a universal microprocessor.

Algorithmic mathematics is that part of mathematics which deals with the design and analysis of algorithms. Since the advent of computers, the importance of algorithms has been steadily increasing in practically all areas of mathematics, not to mention the countless applications. In fact, the field of numerical mathematics and many areas of discrete mathematics arose in this way. Clearly, algorithms are also a vital part of computer science.

1.2 Computational Problems

As usual, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ will denote the sets of natural numbers (without zero), integers, rational numbers and real numbers, respectively. We will not define these formally here, nor will we define the basic concepts of set theory. We will, however, need to introduce several basic mathematical concepts.

Definition 1.1 Let A and B be two sets. Then their **Cartesian product** is given by the set $A \times B := \{(a, b) : a \in A,\ b \in B\}$. A **relation** on (A, B) is a subset of $A \times B$.

For a given set A and a relation $R \subseteq A \times A$ on A we will often write aRb instead of $(a, b) \in R$. Simple examples are the relation $\{(x, x) : x \in \mathbb{R}\}$, usually expressed by the symbol $=$, and the relation $\{(a, b) \in \mathbb{N} \times \mathbb{N} : \exists c \in \mathbb{N} \text{ with } b = ac\}$, usually expressed in words by “a divides b” .

Definition 1.2 Let A and B be two sets and $f \subseteq A \times B$ such that for each $a \in A$ there exists a uniquely defined $b \in B$ with $(a, b) \in f$. Then f is called a **function** (or **mapping**) from A to B. This is also written as $f: A \to B$. Instead of $(a, b) \in f$ we will write $f(a) = b$ or $f: a \mapsto b$. The set A is called the **domain** and the set B the **range** of f.

A function $f: A \to B$ is called **injective** if $a \neq a'$ implies $f(a) \neq f(a')$ for all $a, a' \in A$. A function $f: A \to B$ is called **surjective** if for every $b \in B$ there exists an $a \in A$ with $f(a) = b$. A function is called **bijective** if it is both injective and surjective.

Example 1.3 The function $f: \mathbb{N} \to \mathbb{N}$ given by $f(x) = 2 \cdot x$ is injective but not surjective. The function $g: \mathbb{Z} \to \mathbb{N}$ given by $g(x) = |x| + 1$ is surjective but not injective. The function $h: \mathbb{Z} \to \mathbb{N}$ given by

$$h(x) = \begin{cases} 2 \cdot x, & x > 0 \\ -2 \cdot x + 1, & x \leq 0 \end{cases}$$

is both injective and surjective and thus bijective.

Definition 1.4 For given $a, b \in \mathbb{Z}$ let $\{a, \ldots, b\}$ denote the set $\{x \in \mathbb{Z} : a \leq x \leq b\}$. For $b < a$ the set $\{a, \ldots, b\}$ is the **empty set** $\{\}$ which we will usually denote by the symbol $\emptyset$.

The set A is called **finite** if there exists an injective function $f: A \to \{1, \ldots, n\}$ for some $n \in \mathbb{N}$, otherwise it is called **infinite**. A set A is called **countable** if there exists an injective function $f: A \to \mathbb{N}$, otherwise it is called **uncountable**. The number of elements in a finite set A is denoted by $|A|$.

We can, by means of a function $f: A \to B$, say that a computer program turns a given input $a \in A$ into the output $f(a) \in B$. Practically all computers operate with zeros and ones internally. However, in order to make the input and output more readable, we normally also use other strings of characters:

Definition 1.5 Let A be a nonempty finite set. For a given $k \in \mathbb{N} \cup \{0\}$ let A^k denote the set of functions $f: \{1, \ldots, k\} \to A$. We often write an element $f \in A^k$ as a sequence $f(1) \ldots f(k)$ and call it a **word** (or **string**) of **length** k over the **alphabet** A. The single element $\emptyset$ of A^0 is called the **empty word** (it has length 0). Put $A^* := \bigcup_{k \in \mathbb{N} \cup \{0\}} A^k$. A **language** over the alphabet A is a subset of A^*.

We can now define:

Definition 1.6 A **computational problem** is a relation $P \subseteq D \times E$ with the property that for every $d \in D$ there exists at least one $e \in E$ with $(d, e) \in P$. If $(d, e) \in P$, then e is a **correct output** for the problem P with input d. The elements of D are called the **instances** of the problem.

The computational problem P is said to be **unique** if it is a function. If D and E are languages over a finite alphabet A, then P is called a **discrete computational problem**. If D and E are subsets of $\mathbb{R}^m$ and $\mathbb{R}^n$ with $m, n \in \mathbb{N}$, respectively, then P is called a **numerical computational problem**. A unique computational problem $P: D \to E$ with $|E| = 2$ is called a **decision problem**.

In other words: a computational problem is unique if and only if it has exactly one correct output for every input. If, in addition, the output consists solely of a 0 or a 1 (no or yes), then we have a decision problem.

Computers can by their very nature solve only discrete computational problems. For numerical computational problems one must restrict oneself to inputs that can be written as finite strings of characters and furthermore, the output can only be in this form; thus one must accept the possibility of rounding errors. We will deal with this topic later and will next consider some discrete computational problems. However, before doing so we will, in the next section, make the notion of an algorithm more precise and explain how algorithms can be specified using so-called pseudocode or the programming language C++.

1.3 Algorithms, Pseudocode and C++

The purpose of algorithms is to solve computational problems. The first formal definitions of the concept of algorithm arose in the first half of the twentieth century, for example the definition of Alonzo Church in 1936 via the so-called lambda calculus [6] and of Alan Turing in 1937 by means of Turing machines [35]. Today there are numerous further definitions. All of them have, however, been shown to be equivalent, i.e. the set of computational problems that can be solved by an algorithm is always the same, whatever definition of algorithm we use.

From a practical viewpoint one is, of course, usually only interested in being able to run an algorithm on a computer. Today's computers need to receive their instructions in **machine code**. However, as machine code is practically unintelligible to the human mind, one uses so-called higher programming languages instead. These allow one to formulate relatively easily understood instructions for the computer, which are then translated into machine code by a so-called **compiler**.

In this book we will use the programming language C++ for specifying algorithms. A finite sequence of instructions satisfying the rules of a programming language is called a **program**. For C++ as well as for many other programming languages one can show that all algorithms can be written as programs in the respective programming language. One can therefore also define the concept of algorithm solely via C++ programs. In this book we will introduce only a certain subset of the language C++ and will not give a formal definition of semantics; see [31]. Instead, we will elucidate the most important elements of this programming language by means of examples. The process of formulating an algorithmic idea as a program is called **implementation**.

Another way of specifying algorithms, which will be used extensively in this book, is so-called **pseudocode**. This allows one to reformulate the colloquial description of an algorithm in a precise form which is nevertheless easy to read and can then be converted into the desired programming language formulation without much trouble. One should observe, however, that pseudocode allows one to use colloquial phrases, so that it is in particular impossible to convert pseudocode automatically into machine code or directly into C++. We will define neither syntax nor semantics of pseudocode here as these are largely self-evident and the sole purpose of using pseudocode here is to express the basic ideas of an algorithm.

We will now use a very simple algorithm in order to show how to obtain a pseudocode formulation and then a C++ program. Colloquially, the problem is this: compute the square of a natural number. We thus want to formulate an algorithm which computes the function $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(x) = x^2$. In pseudocode such an algorithm could read as follows:

Algorithm 1.7 (The Square of an Integer)

Input: $x \in \mathbb{N}$.
Output: the square of x.

```
result $\leftarrow$ $x \cdot x$
output result
```

As with every computer program, pseudocode can involve variables; here the variables are the input "x" and the output "result". The arrow "$\leftarrow$" in the above pseudocode specifies that the value of the expression to its right is to be assigned to the variable on its left. Thus, in our case the value of the expression $x \cdot x$ is assigned to the variable "result". Then the command **output** yields the value stored in "result".

In C++ this algorithm can be implemented as follows:

Program 1.8 (The Square of an Integer)

```
 1 // square.cpp (Compute the Square of an Integer)
 2
 3 #include <iostream>
 4
 5
 6 int main()
 7 {
 8     std::cout << "Enter an integer: ";
 9     int x;
10     std::cin >> x;
11     int result = x * x;
12     std::cout << "The square of " << x << " is " << result << ".\n";
13 }
```

We will now briefly explain the lines 8–12 of the program. A detailed explanation of all the lines of the program is contained in the boxes **C++ in Detail (1.1)**, **C++ in Detail (1.2)** and **C++ in Detail (1.3)**. Line 8 comprises a short textual message on the screen, requesting the input of an integer. Line 9 defines a variable "x" with integer values. In line 10 the integer input is read and stored in the variable "x". Line 11 defines another variable called "`result`" and assigns the value $x \cdot x$ to it. Finally, line 12 outputs the input integer and the computed result.

C++ in Detail (1.1): The Basic Structure of a Program

A C++ program is a string of characters satisfying the rules of the language C++. Every C++ program must contain a function called `main`. At the start of the program this function is run and the instructions enclosed between the curly braces { and } following the expression `int main()` are executed.

(continued)

The shortest possible C++ program is:

```
int main(){}
```

As the curly braces contain no characters, this program does nothing. A command in C++ always ends with a semicolon. One can write commands consecutively on a single line, but it improves readability if one writes just one command per line, as was done in Program 1.8. Blanks and empty lines can be inserted in many places of a C++ program without altering the function of the relevant program. One can thus indent blocks of related instructions in order to clarify the structure. Another important feature for increasing readability is the inclusion of comments in the programming code. Such a comment is preceded by `//`. The compiler then ignores all characters after `//` up to the end of the line.

It is also important to note that C++, in contrast to several other programming languages, differentiates between capital and small letters; thus, for example, `int Main(){}` is not a valid C++ program.

C++ in Detail (1.2): Input and Output

A program that computes a function requires as input an element in the domain of the function and yields the corresponding value of the function as output. In the simplest case, input and output of a program are realized with the help of a keyboard and monitor. The language C++ offers the necessary framework for this in the so-called C++ Standard Library. This library is a very extensive collection of useful commands. In order to be able to use these, one must instruct the compiler to make the relevant parts available to the program being written. The part of the library which deals with input and output is called `iostream`. In line 3 of Program 1.8 it is included by `#include <iostream>`.

In order to be able to use the commands from the C++ Standard Library, the prefix `std::` is necessary. The screen output is implemented by entering the command `std::cout` and writing what is to appear on the screen after the output operator `<<`. A line of text will be shown if one includes it between quotation marks, as was done in line 8 of Program 1.8. Values of variables are shown by entering the name of the relevant variable. One need not begin every single output with `std::cout` but can obtain several consecutive outputs by writing them consecutively, separated by `<<` operators, as can be seen in line 12 of Program 1.8. Here, in addition, a new line is produced by means of the characters `\n`.

(continued)

The counterpart to `cout` is `cin`. The command `std::cin >>` enables one to enter values with the keyboard. Here, too, one can perform several input operations consecutively. For example, the command

```
std::cin >> x >> y >> num_iterations;
```

enters values for the three variables `x`, `y` and `num_iterations`.

C++ in Detail (1.3): Elementary Data Types and Operators
A main feature of every programming language is the provision for defining new variables. Every variable in C++ is of a fixed type. For us, the most important types are `int`, `bool` and `double`. The data type `int` is used for storing integers. A variable of type `bool` has only two possible values, called `true` and `false`. Variables of type `double` are used for storing real numbers. In order to define a variable, i.e. to make it known to the compiler, one just enters the relevant data type and the name of the variable; see for example line 9 of Program 1.8. One can also give the variable a certain value at the same time, as was done in line 11 of Program 1.8. Furthermore, one can define several variables of the same type in one instruction by separating them with commas as in the following example:

```
int i, j = 77, number_of_iterations = j + 2, first_result;
```

For the names of variables one can use all letters, digits and the symbol "_". The name of a variable cannot, however, begin with a digit. Moreover, one should always choose the name of a variable so as to enhance the readability of the program.

An expression consists of either a single variable or several variables linked by operators. For the data types `int` and `double` there are not only the elementary arithmetical operators `+`, `-`, `*` and `/`, i.e. addition, subtraction, multiplication and division, but also the relational operators `==`, `<`, `>`, `<=`, `>=` and `!=`. The relational operators correspond to the well-known relations $=$, $<$, $>$, $\leq$, $\geq$ and $\neq$ and their outputs are of type `bool`. Note that the single equal-sign `=` is not a relational operator in C++; it is used for assigning a value to a variable. Expressions of type `bool` can be logically linked by the operators `and` and `or` and can be negated by the operator `not`. The sequential order in which the expressions are to be evaluated can be fixed by means of parentheses `()`.

1.4 A Simple Primality Test

As our first example of a decision problem we will present a test for the primality of a given natural number.

Definition 1.9 A natural number n is called **prime**, if $n \geq 2$ and there are no natural numbers a and b such that $a > 1, b > 1$ and $n = a \cdot b$.

We are thus considering the decision problem $\{(n, e) \in \mathbb{N} \times \{0,1\} : e = 1 \Leftrightarrow n \text{ prime}\}$. Note that in this context $\mathbb{N}$ is taken to be a language over an alphabet, e.g. the usual alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. We usually write such problems as follows:

Decision Problem 1.10 (Primality)

Input: $n \in \mathbb{N}$.
Question: is n prime?

We want to design an algorithm that will solve this problem, i.e. that will decide for an input $n \in \mathbb{N}$ whether or not n is prime. Such an algorithm is usually called a primality test.

The definition of primality suggests an obvious (but not very good) algorithm. One first tests whether $n \geq 2$. If so, then one tests for all natural numbers a and b with $2 \leq a, b \leq \frac{n}{2}$ whether $n = a \cdot b$.

This requires up to $(\frac{n}{2} - 1)^2$ multiplications. Even though today's computers can perform several billion operations per second (including multiplications), the runtime for an eight-digit number, for example, would amount to several hours.

Of course one can do much better. We will first present our primality test in pseudocode:

Algorithm 1.11 (Simple Primality Test)

Input: $n \in \mathbb{N}$.
Output: the answer to the question whether or not n is prime.

```
if $n = 1$ then result $\leftarrow$ "no" else result $\leftarrow$ "yes"
for $i \leftarrow 2$ to $\lfloor\sqrt{n}\rfloor$ do
    if $i$ divides $n$ then result $\leftarrow$ "no"
output result
```

Here we meet two new elements of pseudocode. First there is the **if** command, which is used when wanting to make some instruction depend on the truth of a certain statement (in our case the statement "$n = 1$" or "i divides n"). If the statement is true, then the instructions contained in the **then**-part are executed, and if it is false, then the instructions in the **else**-part are executed. One can also omit the **else**-part, which then means an empty **else**-part.

The second new command appearing in the above example is the **for** command, which is used for consecutively assigning different values to a variable. In our case it is the variable i to which we consecutively assign the values 2 to $\lfloor\sqrt{n}\rfloor$. For each of these values the indented block of instructions below the **for**-part is executed.

There are another two places in Algorithm 1.11 which need a comment. The following definitions are very useful:

Definition 1.12 For $x \in \mathbb{R}$ we define the **floor** and **ceiling** respectively as follows:

$$\lfloor x \rfloor := \max\{k \in \mathbb{Z} : k \leq x\}\,,$$
$$\lceil x \rceil := \min\{k \in \mathbb{Z} : k \geq x\}\,.$$

For two natural numbers a and b we define $a \bmod b := a - b \cdot \lfloor \frac{a}{b} \rfloor$.

Thus, $a \bmod b$ denotes the remainder in the Euclidean division of a by b. Clearly, b divides a if and only if $a \bmod b = 0$. The operations $a \bmod b$ and $\lfloor \frac{a}{b} \rfloor$ are done by any modern computer in just a few clock cycles, so they are considered to be elementary operations. In C++ they are denoted by `a%b` and `a/b`, respectively. Note, however, that if either `a` or `b` is negative, then `a/b` is always rounded to 0 in C++.

It is also not immediately clear whether computing the square root can be accomplished with an elementary operation; this can, however, easily be avoided by increasing i stepwise by 1 until $i \cdot i > n$.

As required, every step of the algorithm is well-defined for any admissible input. It is immediately clear that the algorithm always terminates (i.e. the algorithm stops after a finite number of steps) and computes the correct result. We also say that it **computes the function** $f: \mathbb{N} \to \{\text{yes,no}\}$, where $f(n) = \text{yes}$ if and only if n is prime. Instead of $\{\text{yes,no}\}$ we often write $\{$`true`, `false`$\}$ or $\{1,0\}$.

The number of steps (elementary operations) is here essentially proportional to $\sqrt{n}$. In order to express this precisely, one needs to introduce the Landau symbols:

Definition 1.13 Let $g: \mathbb{N} \to \mathbb{R}_{\geq 0}$. Then we define:

$$O(g) := \{f: \mathbb{N} \to \mathbb{R}_{\geq 0} : \exists \alpha \in \mathbb{R}_{>0}\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0: f(n) \leq \alpha \cdot g(n)\}\,;$$
$$\Omega(g) := \{f: \mathbb{N} \to \mathbb{R}_{\geq 0} : \exists \alpha \in \mathbb{R}_{>0}\, \exists n_0 \in \mathbb{N}\, \forall n \geq n_0: f(n) \geq \alpha \cdot g(n)\}\,;$$
$$\Theta(g) := O(g) \cap \Omega(g)\,.$$

Instead of writing $f \in O(g)$ one also often says: f is $O(g)$ or writes: $f = O(g)$. Note that the equal-sign as it is used here is not symmetric as usual; even so, this notation has become the generally accepted one.

If g denotes the function $n \mapsto \sqrt{n}$ and $f(n)$ the number of elementary operations of Algorithm 1.11 when the input is n, then $f \in O(g)$. We also say that the algorithm has (asymptotic) **running time** $O(\sqrt{n})$.

One can even say in this case that the algorithm has running time $\Theta(\sqrt{n})$, because the number of steps performed by the algorithm is also never less than $\sqrt{n}$. But one can alter Algorithm 1.11 slightly by stopping the **for** loop when the first divisor has been found (this is in fact how we will implement it). This version of the algorithm often terminates much sooner, for example after a constant number of steps for all even numbers n.

Constant factors play no role in the O-notation. We do not in any case know exactly how many elementary operations a computer program will really perform. This number can also depend on the compiler and the hardware. However, the statement that Algorithm 1.11 with running time $O(\sqrt{n})$ is faster than the naive version with running time $\Theta(n^2)$, definitely holds for sufficiently large n and independently of any constants hidden in the O-notation.

One can do better. It was, however, not until 2002 that an algorithm with running time $O((\log n)^k)$ for some constant k was found [1]. Here **log** or $\log_2$ denotes the logarithm to the base 2. The natural logarithm, i.e. the logarithm to the base e, is denoted by **ln**. For the O-notation it is immaterial which base is chosen for logarithms, as $\log n = \Theta(\ln n)$.

Program 1.14 (Simple Primality Test)

```cpp
// prime.cpp (Simple Primality Test)

#include <iostream>


bool is_prime(int n)
{
    // numbers less than 2 are not prime:
    if (n < 2) {
        return false;
    }

    // check all possible divisors up to the square root of n:
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
           return false;
        }
    }
    return true;
}


int get_input()
{
    int n;
    std::cout << "This program checks whether a given integer is prime.\n"
              << "Enter an integer: ";
    std::cin >> n;
    return n;
}

```

```

void write_output(int n, bool answer)
{
    if (answer) {
        std::cout << n << " is prime.\n";
    }
    else {
        std::cout << n << " is not prime.\n";
    }
}


int main()
{
    int n = get_input();
    write_output(n, is_prime(n));
}
```

Program 1.14 comprises a C++ implementation of Algorithm 1.11. More precisely, the function `is_prime` implements the algorithm while the two other functions `get_input` and `write_output` only deal with the input and output, respectively. How one passes parameters to a function in C++ and how functions return their result is explained in Box **C++ in Detail (1.4)**. As in the pseudocode version of Algorithm 1.11, we have used an `if` command and a `for` command in the function `is_prime`. Both of these commands are explained in Box **C++ in Detail (1.5)**.

C++ in Detail (1.4): Functions

Functions in C++ are in some ways similar to mathematical functions. A function in C++ is assigned a certain number of arguments called **function parameters** and yields a value as result. Functions contribute in an essential way to the structure and readability of programs. Suitably chosen names of functions as for example `write_output` and `is_prime` in Program 1.14 immediately tell the reader what an instruction like `write_output (n, is_prime(n))` will do, without having to understand in detail how the function `is_prime` is implemented.

The definition of a function begins with a type specifier fixing the type of the result of the function. If the function does not produce a result, e.g. the function `write_output` in Program 1.14, then its type specifier is `void`. Next comes the name of the function to be defined, and then, in parentheses, the list of assigned parameters separated by commas. Each parameter is entered by giving its type and name. The final entry, in curly braces, is the instruction part of the function. In order to return a result (a value of the function) one uses the `return` command. The function then terminates and the expression following the `return` command is evaluated and appears as the result.

The expression `++i` in line 14 of Program 1.14 has the effect of increasing the variable `i` by 1; alternatively one could also write `i = i + 1`.

C++ in Detail (1.5): `if` and `for`
An `if` command in C++ has the following general form: `if` (*condition*) {*instructionblock1*} `else` {*instructionblock2*} The expression *condition* yields a result of type `bool`. The parentheses containing it are essential. If the result of this expression is `true`, then the instructions in *instructionblock1* are executed, and if the result is `false`, then the instructions in *instructionblock2* are executed. An example of an `if` command is found on lines 35–40 of Program 1.14. The `else`-part of an `if` command can also be omitted. This corresponds to an `else`-part whose *instructionblock2* is empty. We see an example of this on lines 15–17 of Program 1.14. If either of the instruction blocks in an `if` command contains just one instruction, then one can dispense with the curly braces containing it.

A `for` command in C++ has the following general form: `for` `(`*initialization*`;` *condition*`;` *expression*`)` {*instructionblock*} The `for` command begins with the *initialization*. Here one usually defines a variable which is to run through a sequence of values. Then one tests whether the *condition* yields the result `true`. If so, then the *instructionblock* is executed and after that the *expression* is evaluated. The *expression* usually serves the purpose of changing the values of the variables defined in the initialization. The last three steps are repeated until the *condition* yields the value `false`. In lines 14–18 of Program 1.14 we have defined a `for` loop with initialization `int i = 2`, condition `i * i <= n` and expression `++i`. In this example we also see that, due to the `return` command in the instruction block of the loop, a `for` loop can be terminated early, i.e. even when the *condition* still holds.

Here we will not be concerned with how Program 1.14 will react if the input is not a natural number (in practice one should, however, write programs in such a way that unexpected results are prevented from occurring). But it is important to point out already here that Program 1.14 will only function correctly if the input number is not too large. This problem and how it can be solved will be discussed in the next chapter.

1.5 The Sieve of Eratosthenes

In the last section we considered a decision problem. We will now consider an example of a more general discrete computational problem:

Computational Problem 1.15 (List of Prime Numbers)

Input: $n \in \mathbb{N}$.
Task: compute all prime numbers p with $p \leq n$.

If we want to write a program for this problem, we can, of course, reuse the function `is_prime` (as well as `get_input`): this leads directly to an algorithm with running time $O(n\sqrt{n})$. However, Algorithm 1.16, called the Sieve of Eratosthenes, is better. It is given in pseudocode and uses the variable p which is a vector of length n. Accessing the i-th component of this vector is done with $p[i]$.

Algorithm 1.16 (The Sieve of Eratosthenes)

Input: $n \in \mathbb{N}$.
Output: all prime numbers less than or equal to n.

```
for $i \leftarrow 2$ to $n$ do $p[i] \leftarrow$ "yes"
for $i \leftarrow 2$ to $n$ do
    if $p[i] =$ "yes" then
        output $i$
        for $j \leftarrow i$ to $\lfloor \frac{n}{i} \rfloor$ do $p[i \cdot j] \leftarrow$ "no".
```

Theorem 1.17 *Algorithm 1.16 works correctly and has running time $O(n \log n)$.*

Proof For the correctness we show for all $k \in \{2, \dots, n\}$: k is part of the result if and only if k is prime.

Let $k \in \{2, \dots, n\}$. Whenever we get $p[k] \leftarrow$ "no" in the algorithm, it follows that $k = i \cdot j$ with $j \geq i \geq 2$ and so k is not prime. Thus, $p[k] =$ "yes" holds for all prime numbers $k \in \{2, \dots, n\}$ and therefore they are listed correctly.

Suppose that k is not prime. Let i be the least divisor of k with $i \geq 2$. Put $j := \frac{k}{i}$. Then clearly $2 \leq i \leq j = \frac{k}{i} \leq \lfloor \frac{n}{i} \rfloor$ and i is prime. Hence we always get $p[i] =$ "yes" and so we will reach $p[i \cdot j] \leftarrow$ "no". From this point on we will have $p[k] =$ "no". Later, when i gets to the value k, the number k will therefore not appear in the result.

The first line requires $O(n)$ time. The running time of the remainder is at the most proportional to $\sum_{i=2}^{n} \frac{n}{i} = n \sum_{i=2}^{n} \frac{1}{i} \leq n \int_1^n \frac{1}{x} dx = n \ln n$, so Algorithm 1.16 has running time $O(n \log n)$. □

The running time is in fact less because the inner loop is only performed for prime numbers i (note that one first tests whether $p[i] =$ "yes"). Because $\sum_{p \le n:\, p \text{ prime}} \frac{1}{p} = O(\log\log n)$ (an elementary proof can be found in [34]), the Sieve of Eratosthenes in fact only requires $O(n \log\log n)$ time.

It is worth mentioning that this algorithm was probably known already before the time of Eratosthenes (ca. 276–194 BC). Program 1.18 comprises an implementation in C++. The algorithm is actually implemented by the function `sieve` in which we define a variable `is_prime` of type `vector` for the purpose of noting for each number whether or not it is prime. The data type `vector` is explained in Box **C++ in Detail (1.6)**.

Program 1.18 (The Sieve of Eratosthenes)

```cpp
// sieve.cpp (Eratosthenes' Sieve)

#include <iostream>
#include <vector>


void write_number(int n)
{
    std::cout << " " << n;
}


void sieve(int n)
{
    std::vector<bool> is_prime(n + 1, true);  // Initializes variables

    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            write_number(i);
            for (int j = i; j <= n / i; ++j) {
                is_prime[i * j] = false;
            }
        }
    }
}


int get_input()
{
    int n;
    std::cout << "This program lists all primes up to a given integer.\n"
              << "Enter an integer: ";
    std::cin >> n;
    return n;
}


int main()
{
    int n = get_input();
    if (n < 2) {
        std::cout << "There are no primes less than 2.\n";
    }
    else {
        std::cout << "The primes up to " << n << " are:";
        sieve(n);
        std::cout << ".\n";
    }
}
```

C++ in Detail (1.6): The Abstract Data Type `vector`
A `vector` is used for storing a set of objects of the same type. This data type is defined in the C++ Standard Library and can be used after inserting `#include <vector>`. A `vector` is defined by means of the command `std::vector<`*`datatype`*`>` *`vectorname`*`;`, here *`datatype`* fixes the type of objects to be stored in the `vector` and *`vectorname`* gives a name to the variable. In the definition of `vector` one is free to define one or two parameters enclosed between parentheses. The first of these stipulates how many objects the `vector` must have to begin with, and the second stipulates the value with which these are to be initialized. Thus the following definitions

```
std::vector<bool> is_prime;
std::vector<int> prime_number(100);
std::vector<int> v(1000,7);
```

yield a `vector` called `is_prime` with 0 components of type `bool`, a `vector` called `prime_number` with 100 components of type `int`, and a `vector` called `v` with 1000 components of type `int`, all with the value `7`. The components of a `vector` are numbered consecutively, starting with 0. So, for example, the `vector v` defined above contains 1000 components with the indices 0 to 999. One can access the value of the ith component of a `vector` with *`vectorname`*`[i]`; e.g. `v[99]` yields the 100th component of the `vector v` defined above.

A related problem is to find all the prime factors of a given number $n \in \mathbb{N}$, i.e. to find its prime factorization. The question of how fast this problem can be solved is of cardinal importance in practice because cryptographic methods like RSA are based on the assumption that the factors of a given number $n = p \cdot q$ with p and q both large primes cannot be found efficiently.

1.6 Not Everything Is Computable

With algorithms (in particular C++ programs) one can compute a great deal, but not everything. For a start we will show that every language, including C++ programs of course, is countable. We will need the following lemma:

Lemma 1.19 *Let $k \in \mathbb{N}$ and $l \in \mathbb{N}$. Define $f^l \colon \{0, \ldots, k-1\}^l \to \{0, \ldots, k^l - 1\}$ by $f^l(w) := \sum_{i=1}^{l} a_i k^{l-i}$ for all $w = a_1 \ldots a_l \in \{0, \ldots, k-1\}^l$. Then f^l is well-defined and bijective.*

Proof For $w \in \{0, \ldots, k-1\}^l$ we have $0 \le f^l(w) \le \sum_{i=1}^{l}(k-1)k^{l-i} = k^l - 1$, hence f^l is well-defined.

We will now show that f^l is injective. Let $w, w' \in \{0, \dots, k-1\}^l$ with $w = a_1 \dots a_l$, $w' = a'_1 \dots a'_l$ and $w \neq w'$. Then there exists a minimum index j with $1 \leq j \leq l$ such that $a_j \neq a'_j$. We can assume without loss of generality that $a_j > a'_j$. Then the following holds: $f^l(w') = \sum_{i=1}^{l} a'_i k^{l-i} = \sum_{i=1}^{j-1} a'_i k^{l-i} + a'_j k^{l-j} + \sum_{i=j+1}^{l} a'_i k^{l-i}$. Because $\sum_{i=1}^{j-1} a'_i k^{l-i} = \sum_{i=1}^{j-1} a_i k^{l-i}$, $a'_j k^{l-j} \leq (a_j - 1)k^{l-j}$ and $\sum_{i=j+1}^{l} a'_i k^{l-i} \leq \sum_{i=j+1}^{l} (k-1)k^{l-i} = k^{l-j} - 1$, it follows that $f^l(w') \leq \sum_{i=1}^{j-1} a_i k^{l-i} + (a_j - 1)k^{l-j} + k^{l-j} - 1 = \sum_{i=1}^{j-1} a_i k^{l-i} + a_j k^{l-j} - 1 < f^l(w)$. Hence $f^l(w') \neq f^l(w)$.

As $|\{0, \dots, k-1\}^l| = k^l = |\{0, \dots, k^l - 1\}|$, the function f^l is also surjective. □

Theorem 1.20 *Let A be a nonempty finite set. Then the set A^* is countable.*

Proof Let $f: A \to \{0, \dots, |A| - 1\}$ be a bijection and define the function $g: A^* \to \mathbb{N}$ by $g(a_1 \dots a_l) := 1 + \sum_{i=0}^{l-1} |A|^i + \sum_{i=1}^{l} f(a_i)|A|^{l-i}$. We assert that g is injective. This is, however, a direct consequence of Lemma 1.19 because g maps the set of words of length l bijectively onto the set $\{1 + \sum_{i=0}^{l-1} |A|^i, \dots, \sum_{i=0}^{l} |A|^i\}$. □

Corollary 1.21 *The set of all C++ programs is countable.*

Proof C++ programs are words over a finite alphabet. As a subset of a countable set is again countable, the statement follows from Theorem 1.20. □

With this we can now show:

Theorem 1.22 *There exist functions $f: \mathbb{N} \to \{0,1\}$ that cannot be computed by any C++ program.*

Proof Let $\mathcal{P}$ be the (countable by Corollary 1.21) set of those C++ programs that compute a function $f: \mathbb{N} \to \{0,1\}$. Let $g: \mathcal{P} \to \mathbb{N}$ be an injective function. For $P \in \mathcal{P}$ let f^P be the function computed by P.

Consider a function $f: \mathbb{N} \to \{0,1\}$ with $f(g(P)) := 1 - f^P(g(P))$ for all $P \in \mathcal{P}$. Such an f exists because g is injective. Clearly $f \neq f^P$ for all $P \in \mathcal{P}$, hence there is no C++ program that computes f. □

This is essentially Cantor's diagonal proof which also proves that $\mathbb{R}$ is uncountable. One sees in fact that computable functions are "sparse": there are only countably many of them in an uncountable set.

Among the non-computable functions there are certainly some very interesting ones which can be given concretely. We will present the most famous one here.

Let $\mathcal{Q}$ be the set of all C++ programs that accept a natural number as input. Let $g\colon \mathbb{N} \to \mathcal{Q}$ be a surjective function. As $\mathcal{Q}$ is countable by Corollary 1.21, there exists an injective function $f\colon \mathcal{Q} \to \mathbb{N}$ and we can define $g(f(Q)) := Q$ for $Q \in \mathcal{Q}$ and $g(n) := Q_0$ for some C++ program Q_0 if $n \notin \{f(Q) : Q \in \mathcal{Q}\}$.

We now define a function $h\colon \mathbb{N} \times \mathbb{N} \to \{0,1\}$ by

$$h(x,y) = \begin{cases} 1 & \text{if the program } g(x) \text{ with input } y \text{ terminates after finitely many steps;} \\ 0 & \text{otherwise.} \end{cases}$$

This decision problem is called the **halting problem** and the function h is thus called halting function.

Theorem 1.23 *The halting function h cannot be computed by any C++ program.*

Proof Suppose that there exists a program P which computes h. Then there also exists a program Q which, given $x \in \mathbb{N}$ as input, first computes the function $h(x,x)$ and then terminates if $h(x,x) = 0$ and enters an infinite loop if $h(x,x) = 1$.

Let $q \in \mathbb{N}$ with $g(q) = Q$. By the definition of h it follows that $h(q,q) = 1$ if and only if $g(q)$ terminates with input q. By the construction of Q, however, $h(q,q) = 0$ if and only if Q terminates with input q. As this is a contradiction, no such P exists. □

Clearly, this theorem is equally valid for any other programming language and also for algorithms in general. One also says that the halting problem is not **decidable**.

Also in practice it is often no easy matter to determine whether a program will always terminate. A good example of this is the following algorithm:

Algorithm 1.24 (Collatz Sequence)

Input: $n \in \mathbb{N}$.
Output: the answer to the question whether or not the Collatz sequence for n attains the value 1.

```
while $n > 1$
    if $n \bmod 2 = 0$
        then $n \leftarrow n/2$
        else $n \leftarrow 3 \cdot n + 1$
output "the Collatz sequence attains the value 1"
```

To the present day one does not know whether this simple algorithm always terminates. This question is known as the Collatz problem. An implementation of Algorithm 1.24 in C++ is given in Program 1.25:

Program 1.25 (Collatz Sequence)

```
 1 // collatz.cpp (Collatz Sequence)
 2
 3 #include <iostream>
 4
 5 using myint = long long;
 6
 7 myint get_input()
 8 {
 9     myint n;
10     std::cout << "This program computes the Collatz sequence for an "
11               << "integer.\n" << "Enter an integer: ";
12     std::cin >> n;
13     return n;
14 }
15
16
17 int main()
18 {
19     myint n = get_input();
20     while (n > 1) {
21         std::cout << n << "\n";
22         if (n % 2 == 0) {
23             n = n / 2;
24         }
25         else {
26             n = 3 * n + 1;
27         }
28     }
29     std::cout << n << "\n";
30 }
```

In Program 1.25 we have used the data type `long long` instead of `int` (and have shortened it to `myint` with the instruction `using`), as this enables us to store large integers as well. The largest representable value can be found with the help of the function `std::numeric_limits<myint>::max()`, which is defined in `<limits>`. The result depends on the compiler used, but its value is usually $2^{63} - 1$.

In Program 1.25, using the data type `long long` does not, however, necessarily prevent one from exceeding the range of representable numbers and thus causing a so-called overflow. In the following chapter we will see how even larger numbers can be represented.

In lines 20–28 of Program 1.25 we have made use of another new element of C++, namely the `while` command. It is explained in Box **C++ in Detail (1.7)**.

C++ in Detail (1.7): The `while` Command
The `while` command has the following general form: `while (`*`condition`*`) {`*`instructionblock`*`}`. The *`condition`* is evaluated and if the result is `true`, then the *`instructionblock`* is executed; if the result is `false`, then one continues with the programming code that follows *`instructionblock`*. This step is repeated as long as the *`condition`* has the value `true` and stops when it has the value `false`. Alternatively, there is the `do while` command: `do {`*`instructionblock`*`} while (`*`condition`*`);` This code works similarly, the only difference being that the *`instructionblock`* is executed before the *`condition`* is tested.

Experimental results have shown that Collatz sequences are finite for numbers up to $5 \cdot 2^{60}$. Thus the above program terminates for such inputs [27].

2 Representations of the Integers

A computer stores all its information by means of bits valued 0 or 1. A byte consists of eight bits. We have up to this point assumed that we could store all occurring natural numbers in the data type `int`. This, however, is in fact not the case. A variable of type `int` usually corresponds to a sequence of 4 bytes. This clearly enables us to represent no more than 2^{32} different numbers. In this chapter we will learn how integers are stored.

2.1 The b-Adic Representation of the Natural Numbers

The natural numbers are stored using the binary representation (also called the 2-adic representation). This is exactly analogous to the usual decimal representation: for example

$$\begin{aligned}106 &= 1\cdot 10^2 + 0\cdot 10^1 + 6\cdot 10^0\\ &= 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0\end{aligned}$$

with the binary representation 1101010. Instead of 10 or 2 one can choose any other natural number $b \geq 2$ as a base:

Theorem 2.1 *Let $b \in \mathbb{N}$, $b \geq 2$, and $n \in \mathbb{N}$. Then there exist uniquely defined numbers $l \in \mathbb{N}$ and $z_i \in \{0, \dots, b-1\}$ for $i = 0, \dots, l-1$ with $z_{l-1} \neq 0$, such that*

$$n = \sum_{i=0}^{l-1} z_i b^i .$$

The word $z_{l-1} \dots z_0$ is called the **b-adic representation** *of n; sometimes also written as $n = (z_{l-1} \dots z_0)_b$. The following always holds: $l - 1 = \lfloor \log_b n \rfloor$.*

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_2

Mathematical Induction

In order to prove a statement $A(n)$ for all $n \in \mathbb{N}$, it clearly suffices to show that $A(1)$ holds (initial step) and that for all $i \in \mathbb{N}$ the statement $A(i)$ implies the statement $A(i+1)$ (inductive step). In the proof of the inductive step the statement $A(i)$ is called the induction hypothesis. This method of proof is known as (mathematical) induction.

More generally, one can prove a statement $A(m)$ for all $m \in M$, where M is any set, by giving a function $f: M \to \mathbb{N}$ and showing that for every $m \in M$ the statement "$A(m)$ is false" implies that there exists an $m' \in M$ with $f(m') < f(m)$ such that $A(m')$ is also false. This is referred to as induction over f.

Proof The final statement follows immediately: if $n = \sum_{i=0}^{l-1} z_i b^i$ with $z_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l-1$ and $z_{l-1} \neq 0$, then $b^{l-1} \leq n \leq \sum_{i=0}^{l-1}(b-1)b^i = b^l - 1$ and thus $\lfloor \log_b n \rfloor = l - 1$.

The uniqueness of the b-adic representation now follows directly with Lemma 1.19.

The existence is proved by induction over $l(n) := 1 + \lfloor \log_b n \rfloor$; see the box **Mathematical Induction**. If $l(n) = 1$, i.e. $n \in \{1, \ldots, b-1\}$, then n has the representation $n = \sum_{i=0}^{0} z_i b^i$ with $z_0 = n$.

It remains to prove the case $n \in \mathbb{N}$ with $l(n) \geq 2$. Define $n' := \lfloor n/b \rfloor$. Then $l' := l(n') = l(n) - 1$. By the induction hypothesis n' has a representation $n' = \sum_{i=0}^{l'-1} z'_i b^i$ with $z'_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l'-1$ and $z'_{l'-1} \neq 0$. Define $z_i := z'_{i-1}$ for $i = 1, \ldots, l'$ and $z_0 := n \bmod b \in \{0, \ldots, b-1\}$. Then we obtain:

$$n = b\lfloor n/b \rfloor + (n \bmod b) = bn' + z_0 = b \cdot \sum_{i=0}^{l'-1} z'_i b^i + z_0 = \sum_{i=1}^{l'} z'_{i-1} b^i + z_0 = \sum_{i=0}^{l-1} z_i b^i.$$

□

If a number is given in b-adic notation as $(z_{l-1} \ldots z_0)_b$, one can find its decimal representation by using the so-called Horner scheme:

$$\sum_{i=0}^{l-1} z_i \cdot b^i \;=\; z_0 + b \cdot (z_1 + b \cdot (z_2 + \ldots + b \cdot (z_{l-2} + b \cdot z_{l-1}) \cdots))$$

saving several multiplication steps.

Conversely, the proof of Theorem 2.1 immediately yields an algorithm for obtaining the b-adic representation of a number $z \in \mathbb{N}$ for any b. Program 2.2 is a C++ implementation of this algorithm which works for $2 \leq b \leq 16$. Apart from the binary representation ($b = 2$), the octal representation ($b = 8$) and the hexadecimal representation ($b = 16$) traditionally also play a certain role. When representing a number with base $b > 10$, one uses the letters A,B,C,. . . for the digits greater than 9.

Program 2.2 (Base Converter)

```cpp
// baseconv.cpp (Integer Base Converter)

#include <iostream>
#include <string>
#include <limits>

const std::string hexdigits = "0123456789ABCDEF";

std::string b_ary_representation(int base, int number)
// returns the representation of "number" with base "base", assuming 2<=base<=16.
{
    if (number > 0) {
        return b_ary_representation(base, number / base) + hexdigits[number % base];
    }
    else {
        return "";
    }
}


bool get_input(int & base, int & number)           // call by reference
{
    std::cout << "This program computes the representation of a natural number"
              << " with respect to a given base.\n"
              << "Enter a base among 2,...," << hexdigits.size() << " : ";
    std::cin >> base;
    std::cout << "Enter a natural number among 1,...,"
              << std::numeric_limits<int>::max() << " : ";
    std::cin >> number;
    return (base > 1) and (base <= hexdigits.size()) and (number > 0);
}


int main()
{
    int b, n;
    if (get_input(b, n)) {
        std::cout << "The " << b << "-ary representation of " << n
                  << " is " << b_ary_representation(b, n) << ".\n";
    }
    else std::cout << "Sorry, wrong input.\n";
}
```

Program 2.2 uses the data type `string` from the C++ Standard Library. A little more background information on using this data type is given in the box **C++ in Detail** (2.1).

C++ in Detail (2.1): Strings

In order to be able to use the data type `string` from the C++ Standard Library, one first has to include the relevant part with `#include <string>`. A variable `s` of type `string` can then be defined by means of `std::string s;`. As in line 7 of Program 2.2, one can assign a value directly to a `string`-variable by entering the explicit value when defining the variable. Another useful feature is, that one can fill a `string`-variable with a given number of the same symbol. For example,

```
std::string s(10, 'A');
```

defines a `string`-variable `s` with the value `"AAAAAAAAAA"`. One can access the symbol in position `i` of a `string`-variable `s` via `s[i]`, noting, however, that the first symbol in a `string`-variable is in position 0. The expression `s1 + s2` generates a `string` which is formed by joining the `string` `s2` to the back of the `string` `s1`. The function `s.size()` returns the number of symbols contained in the `string` `s`.

In Program 2.2 we have our first example of a function which calls itself. Such functions are said to be recursive. By using them one can often make the programming code more elegant; however, one has to take care that the recursion depth, i.e. the maximum number of nested function calls, is bounded (here it is not more than 32 if the largest representable number of the data type `int` is $2^{31} - 1$). An alternative nonrecursive implementation of the function `b_ary_representation` is shown in the following excerpt of programming code:

```
std::string b_ary_representation(int base, int number)
// returns the representation of "number" with base "base", assuming 2<=base<=16.
{
    std::string result = "";
    while (number > 0) {
        result = hexdigits[number % base] + result;
        number = number / base;
    }
    return result;
}
```

In addition, one should note that variables are not allotted to the function `get_input` as was the case with earlier functions where values were assigned to new local variables ("call by value"), but that references are assigned to variables defined in the function `main` by means of the prefix `&` ("call by reference"). The function `get_input` thus has no variables of its own. Even more holds: the two variables `b` and `n` defined in the function `main` continue to be used in the function `get_input` with new names, `base` and `number`, respectively.

Note also, that we have defined the `string`-variable `hexdigits` in line 7 of Program 2.2 outside of every function. Thus, this variable is a **global variable** and is recognized by each of the three functions appearing in the program. In order to prevent a value of this variable being accidentally changed, the variable has been given the prefix `const` which defines it to be a **constant**.

The representation of numbers to the base 8 or 16 can, moreover, be achieved very easily in C++ by using the stream manipulators `std::oct` and `std::hex`. Thus, for example, the instruction `std::cout << std::hex;` has the effect that all numbers in the output are given in hexadecimal notation. With `std::dec` one returns to the decimal representation of numbers.

2.2 Digression: Organization of the Main Memory

Program 2.2 serves well as an example for elucidating the organization of that part of the main computer memory which the operating system has made available for running the program. Every byte in this area has an address which is usually given in hexadecimal notation.

When a program is run, the available part of the main memory comprises four parts, often called "code", "static", "stack" and "heap" (see Fig. 2.1; further details depend on the processor and the operating system).

The part denoted code contains the program itself, given in machine code which the compiler has generated from the source code. It is partitioned into the programming code of the various functions.

In the part denoted stack, every call of a function reserves a place "on top" of the stack for the result (if it is not `void`) and for the local variables of the function (including its arguments). For any variable allotted by "call by reference", however, no new variable is formed but just the memory address of its allotted variable is stored. When running the program, the processor always keeps track of two addresses, one for the currently executed point in the code and another for the place in the stack where the range of the currently executed function begins. If a function is run recursively, then every call is assigned a new part in the stack.

In addition, two return addresses are stored for every function (except for `main`), one in the code and one in the stack, so that the program knows where and with which variables it must continue after finishing with the function, whose place in the stack is then vacated and can be reused.

There is, furthermore, some "free memory", the so-called heap, which one needs mainly when one does not know how much memory is required until running the program. For example, the data saved in a `vector` or a `string` are stored in a section of the heap. There is, however, always a corresponding normal variable in the stack, containing the storage address of the place where the section of the heap with the actual data begins. Such a variable is called a pointer.

Finally, there is a part denoted static, containing any global variables (which one should avoid if possible).

The contents of the memory are not initialized at the beginning; thus, for example, one does not know the value of a non-initialized variable. One also has no influence on the choice of storage addresses in the stack or heap. The storage

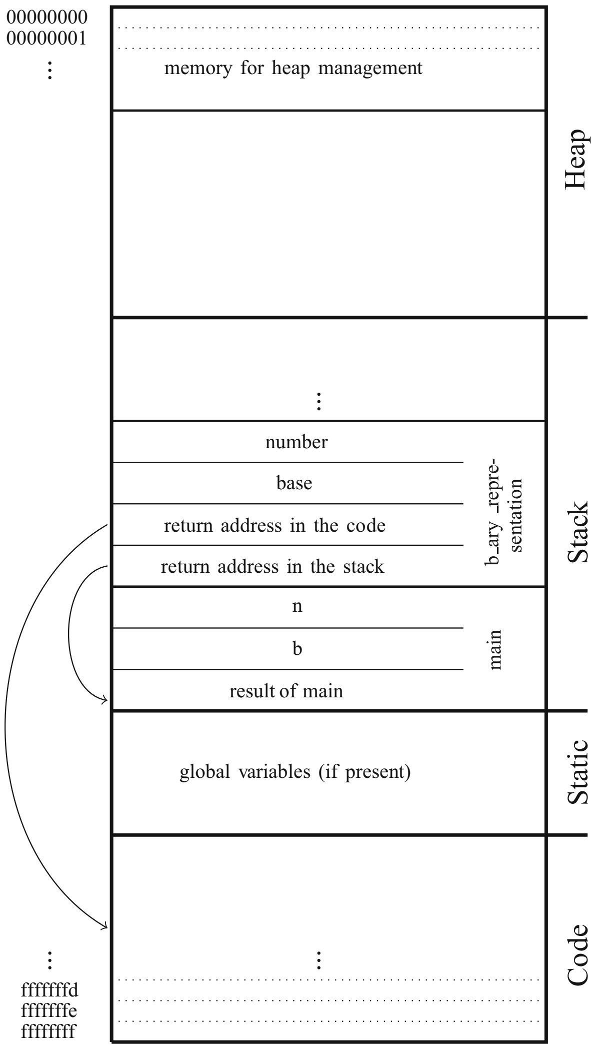


Fig. 2.1 An example of the organization of the main memory

address of a variable `x` is denoted by `&x`. Conversely, if `p` is a pointer, then `*p` denotes the content of the variable saved at the storage address `p`.

The value of a variable handed to a function via call by reference can be changed by this function (but clearly not its storage address). If one does not want to change the value, it is best to use the prefix `const` as a note for the reader of the source code and in order to prevent an inadvertent change.

2.3 The *b*'s Complement Representation of the Integers

We are accustomed to denoting negative numbers by prefixing them with a "–" sign. In a computer this could, for example, be achieved by reserving the first bit for the sign, assigning the value 0 to the "+" sign and the value 1 to the "–" sign. This is called the sign representation.

Example 2.3 Consider the binary representation with four bits, where the first bit is reserved for the sign. The number $+5$ is given by 0101 and the number -5 as 1101. Further examples of the sign representation are:

$$\begin{array}{lcl} 0 \text{ is given by } 0000 & \text{and} & -0 \text{ is given by } 1000 \\ 1 \text{ is given by } 0001 & & -1 \text{ is given by } 1001 \\ \vdots & & \vdots \\ 7 \text{ is given by } 0111 & & -7 \text{ is given by } 1111 \end{array}$$

One disadvantage of this representation is that 0 is represented in two different ways. A much more serious disadvantage is, however, that addition requires a case distinction: one cannot simply add the binary representations, as the following counterexample shows: the addition of 0010 and 1001 should yield the value 0001.

For this reason the above representation is not used in standard data types. Instead, one identifies a negative number $z \in \{-2^{l-1}, \dots, -1\}$ with $z + 2^l$, where l is the number of bits used. This is called the 2's complement representation.

Example 2.4 With this representation using four bits we now have:

$$\begin{array}{lcl} 0 \text{ is given by } 0000 & \text{and} & -8 \text{ is given by } 1000 \\ 1 \text{ is given by } 0001 & & -7 \text{ is given by } 1001 \\ \vdots & & \vdots \\ 7 \text{ is given by } 0111 & & -1 \text{ is given by } 1111 \end{array}$$

The addition of 0010 (representing the number 2) and 1001 (representing the number -7) yields 1011, which represents the number -5.

One can generalize this and define the *b*'s complement representation of a number for any base $b \geq 2$. For this we first need to define the *b*'s complement:

Definition 2.5 Let l and $b \geq 2$ be natural numbers and $n \in \{0, \dots, b^l - 1\}$. Then $K_b^l(n) := (b^l - n) \bmod b^l$ is the l-digit ***b*'s complement** of n.

K_2 is also called the 2's complement and K_{10} the 10's complement.

Lemma 2.6 *Let $b, l \in \mathbb{N}$ with $b \geq 2$, and let $n = \sum_{i=0}^{l-1} z_i b^i$ with $z_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l-1$. Then the following statements hold:*

(i) *$K_b^l(n+1) = \sum_{i=0}^{l-1}(b-1-z_i)b^i$ if $n \neq b^l - 1$; furthermore, $K_b^l(0) = 0$;*
(ii) *$K_b^l(K_b^l(n)) = n$.*

Proof $K_b^l(0) = 0$ follows immediately from the definition of the b's complement. Furthermore, for $n \in \{0, \ldots, b^l - 2\}$ we have: $K_b^l(n+1) = b^l - 1 - \sum_{i=0}^{l-1} z_i b^i = \sum_{i=0}^{l-1}(b-1)b^i - \sum_{i=0}^{l-1} z_i b^i = \sum_{i=0}^{l-1}(b-1-z_i)b^i$. This completes the proof of (i).

$K_b^l(K_b^l(0)) = 0$ follows immediately from (i). For $n > 0$ we have that $K_b^l(n) = b^l - n > 0$, hence $K_b^l(K_b^l(n)) = b^l - (b^l - n) = n$. This completes the proof of (ii). □

Example 2.7 The b's complement of a positive number can be computed by means of Lemma 2.6(i): subtract 1 and then compute the difference with $b-1$ place by place, for example:

$$K_2^4((0110)_2) = (1010)_2 \qquad K_{10}^4((4809)_{10}) = (5191)_{10}$$
$$K_2^4((0001)_2) = (1111)_2 \qquad K_{10}^4((0000)_{10}) = (0000)_{10}$$

Definition 2.8 Let l and $b \geq 2$ be natural numbers and $n \in \{-\lfloor b^l/2 \rfloor, \ldots, \lceil b^l/2 \rceil - 1\}$. Then one obtains the ***l*-digit *b*'s complement representation** of n by preceding the b-adic representation of n (for $n \geq 0$) or the b-adic representation of $K_b^l(-n)$ (for $n < 0$) with as many zeros as necessary to yield an l-digit number.

The main advantage of the b's complement representation is that it enables easy computing without having to deal with different cases, as is shown in the following theorem.

Theorem 2.9 *Let l and $b \geq 2$ be natural numbers and $Z := \{-\lfloor b^l/2 \rfloor, \ldots, \lceil b^l/2 \rceil - 1\}$. Consider the function $f: Z \to \{0, \ldots, b^l - 1\}$ defined by $z \mapsto \begin{cases} z & \text{for } z \geq 0 \\ K_b^l(-z) & \text{for } z < 0 \end{cases}$ Then f is bijective and for $x, y \in Z$ the following two statements hold:*

(a) *If $x + y \in Z$, then $f(x+y) = (f(x) + f(y)) \bmod b^l$;*
(b) *If $x \cdot y \in Z$, then $f(x \cdot y) = (f(x) \cdot f(y)) \bmod b^l$.*

Proof For $z \in Z$ we have $f(z) = (z + b^l) \bmod b^l$. As $|Z| = b^l$, it follows that f is bijective. Let $x, y \in Z$ and $p, q \in \{0, 1\}$ with $f(x) = x + pb^l$ and $f(y) = y + qb^l$. Then the following two statements hold:

(a) $f(x+y) = (x + y + b^l) \bmod b^l = ((x + pb^l) + (y + qb^l)) \bmod b^l = (f(x) + f(y)) \bmod b^l$.

(b) $f(x \cdot y) = (x \cdot y + b^l) \bmod b^l = ((x+pb^l)\cdot((y+qb^l)) \bmod b^l = (f(x)\cdot f(y)) \bmod b^l$. □

The 2's complement representation is used for storing integers in the computer. In this representation the first bit equals 1 if and only if the represented number is negative. The number l of bits used is nearly always a power of 2 and a multiple of 8. For the data type `int`, for example, usually $l = 32$ (see below), permitting the representation of all numbers in the range $\{-2^{31}, \ldots, 2^{31} - 1\}$.

Partitions and Equivalence Relations
Let S be a set. A **partition** of S is a set of nonempty, pairwise disjoint subsets of S whose union is S. The set $\{1, 2, 3\}$, for example, has five different partitions.
A relation $R \subseteq S \times S$ is called an **equivalence relation** (on S) if the following three statements hold for all $a, b, c \in S$:

- $(a, a) \in R$ (reflexivity);
- $(a, b) \in R \Rightarrow (b, a) \in R$ (symmetry);
- $((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$ (transitivity).

For every equivalence relation R on S the set $\{\{s \in S : (a, s) \in R\} : a \in S\}$ is a partition of S; its elements are called the **equivalence classes** of R. Conversely, every partition $\mathcal{P}$ of S induces an equivalence relation on S defined by the relation $R := \{(a, b) \in S \times S : \exists P \in \mathcal{P} \text{ with } a, b \in P\}$.
For example, $=$ is an equivalence relation on $\mathbb{R}$; each of its equivalence classes contains just one element. On $\mathbb{Z}$ the relation $R_k := \{(x, y) \in \mathbb{Z} \times \mathbb{Z} : k \text{ divides } |x-y|\}$ defines a different equivalence relation for every $k \in \mathbb{N}$, with exactly k (infinite) equivalence classes. The set of equivalence classes of R_k is often denoted by $\mathbb{Z}/k\mathbb{Z}$ and is then called the ring of residue classes of $\mathbb{Z}$ mod k.

If one ignores numbers lying outside the stipulated range Z (or always calculates mod b^l), one can thus use the b's complement representation just like the b-adic representation. In both cases one in fact uses the elements of the residue class ring $\mathbb{Z}/b^l\mathbb{Z}$; see the box **Partitions and Equivalence Relations**. Just the representatives of some of the equivalence classes differ in the two representations.

Subtraction can be reduced to addition, and comparisons by prior subtraction to testing the first bit.

The b's complement representation was already used in the seventeenth century by designers of mechanical calculating machines. One of the largest collections worldwide of such machines is housed and can be viewed in the Arithmeum in Bonn.

Numbers lying outside the stipulated range are not normally intercepted automatically. One therefore has to watch this when programming and, if necessary, check by prior testing whether a (usually unwanted) overflow can occur.

The following table lists the most important C++ data types for integers. The numbers are represented in the 2's complement representation, so the least or greatest representable number is given by the number of bits used.

Data type	Number of bits with `gcc 6.1.0` Windows 10	Number of bits with `gcc 6.1.0` CentOS 7
`short`	16	16
`int`	32	32
`long`	32	64
`long long`	64	64

The number of bytes required by a data type or a variable can be ascertained by means of `sizeof (`*`Datatype`*`)` or `sizeof` *`Variable`* respectively. The result is of type `size_t` which can store nonnegative integers of sufficient size. So there is no reason to assume blindly that, for example, an `int` really has 4 bytes.

We have seen that the standard data types only allow us to store integers within a certain interval. This is often problematic; in a Collatz sequence, for example, numbers larger than 2^{64} can appear, even when the input is much smaller (cf. Program 1.25). If one does not intercept this, the computation usually continues mod 2^l, where l is the number of bits of the data type.

2.4 Rational Numbers

One can, of course, store rational numbers by allotting separate variables to numerator and denominator. Here it makes sense to define a new data type; called a **class** in C++. Classes constitute the most important construct in C++, so we will discuss them in some detail here.

A class enables one to store data in variables and provides functions that can operate on this data. Furthermore, types and constants can be defined. Of decisive importance is the clear distinction between the internal data management and the interface with the exterior. Consider Program 2.10, which consists of two files. In the file `fraction.h` a class called `Fraction` has been defined, which is suitable for storing rational numbers. Program `harmonic.cpp` is an example of how this class can be used: in fact very much like a standard data type.

The members of a class are either `public` or `private`. The part called `public` comprises the interface with the exterior. The part called `private` should contain the data (here the numerators and denominators); possibly also functions that are only needed as subroutines of the `public` functions and should not themselves be visible from the outside. Everything that is `private` is not directly accessible from the outside. This helps to prevent programming errors.

A class can contain four kinds of functions:

- Constructors for generating an object in the class, very much like the way a variable of a standard data type is declared (and possibly also initialized). A constructor always has the same name as the class itself. In `fraction.h` the (in this case single) constructor is defined in lines 9–12. If no constructor is defined, the compiler defines a standard constructor which initializes the objects belonging to the class.
- A destructor, which is called when the lifetime of an object in the class terminates, just like variables of a function are vacated after terminating the function. As no destructor has been defined explicitly here (it would be called `~Fraction`), this role is taken on by a standard destructor generated by the compiler.
- Further functions which always operate on an existing object in the class. They can also only be called in connection with an object. They can either change this object (as for example `reciprocal()`) or not (as for example `numerator()`); in the second case one should write `const` behind the declaration. Otherwise they behave like normal functions.
- Operators, predefined for certain standard data types, can also be defined for a new class (as shown here for < and +). Operators behave just like functions; except for their special name and their more comfortable call (see for example line 16 of `harmonic.cpp`) there are no differences.

As with other functions, operators can, apart from the object in the class with which they are associated, also be assigned another object in the same class as a parameter and can, as a result, also generate a new object in the class. Their data and functions can also be accessed with "`.`". The addition shown here is an example of this.

Constructors and destructors can be called explicitly or implicitly in different ways. Lines 14 and 16 of `harmonic.cpp` show two explicit calls of the constructor, another one takes place in lines 44–46 of `fraction.h`.

Program 2.10 (Rational and Harmonic Numbers)

```
1  // fraction.h (Class Fraction)
2
3  #include <stdexcept>
4
5  class Fraction {
6  public:
7      using inttype = long long;
8
9      Fraction(inttype n, inttype d): _numerator(n), _denominator(d)
10     {   // constructor
11         if (d < 1) error_zero();
12     }
13
14     inttype numerator() const                   // return numerator
```

```
    {
        return _numerator;
    }

    inttype denominator() const                  // return denominator
    {
        return _denominator;
    }

    void reciprocal()                            // replaces a/b by b/a
    {
        if (numerator() == 0) {
            error_zero();
        } else {
            std::swap(_numerator, _denominator);
            if (denominator() < 0) {
                _numerator   = -_numerator;
                _denominator = -_denominator;
            }
        }
    }

    bool operator<(const Fraction & x) const     // comparison
    {
        return numerator() * x.denominator() < x.numerator() * denominator();
    }

    Fraction operator+(const Fraction & x) const // addition
    {
        return Fraction(numerator() * x.denominator() +
                        x.numerator() * denominator(),
                        denominator() * x.denominator());
    }

    // further operations may be added here

private:
    inttype _numerator;
    inttype _denominator;

    void error_zero()
    {
        throw std::runtime_error("Denominator < 1 not allowed in Fraction.");
    }
};
```

```
// harmonic.cpp (Harmonic Numbers)

#include <iostream>
#include <stdexcept>
#include "fraction.h"

int main()
{
try {
    std::cout << "This program computes H(n)=1/1+1/2+1/3+...+1/n.\n"
              << "Enter an integer n: ";
    Fraction::inttype n;
    std::cin >> n;
    Fraction sum(0,1);
    for (Fraction::inttype i = 1; i <= n; ++i) {
        sum = sum + Fraction(1, i);
    }
    std::cout << "H(" << n << ") = "
              << sum.numerator() << "/" << sum.denominator() << " = "
              << static_cast<double>(sum.numerator()) / sum.denominator() << "\n";
}
catch(std::runtime_error e) {
    std::cout << "RUNTIME ERROR: " << e.what() << "\n";
    return 1;
}
}
```

The output operator << can also be redefined. We could then replace the output in line 19 of `harmonic.cpp` by `<< sum`. Note, however, that in this case the first operand of << is of type `ostream`. So the new operator cannot be defined within the class `Fraction`, it has to be defined as a normal function outside `Fraction`. One could, for example, write the following code:

```
// output operator for class Fraction

std::ostream & operator<<(std::ostream & os, const Fraction & f)
{
    os << f.numerator() << "/" << f.denominator();
    return os;
}
```

Program 2.10 also yields our first example of how to deal with errors occurring while a program is running. Here one uses so-called `exceptions` which are explained in more detail in the box **C++ in Detail (2.2)**. In line 29 of `fraction.h` we have used the `swap` function which interchanges the contents of the two allotted variables.

C++ in Detail (2.2): Handling Errors with `try-catch`
During the running time of a program numerous errors can occur, for which it is not immediately clear how to deal with them at the place in the code where the error was found. Typical errors occurring while the program is running are, for example, division by 0 or the range overflow of an `int` type. The idea behind error handling in C++ is, that the part of the programming code where the error was identified reports it to the place in the code where one knows how to deal with it. The various methods of handling errors are defined in the C++ Standard Library under the header `stdexcept`. This header must therefore be included with `#include <stdexcept>`. If a runtime error is noticed, it can be referred to the calling part of the programming code with

`throw std::runtime_error (`*`error_message`*`);`

Here *`error_message`* is a `string` describing the relevant error. Line 57 of `fraction.h` provides an example. A runtime error referenced by `throw` can be suitably treated with a `try-catch` construct. This takes the general form:

`try {`*`subprogram`*`} catch (std::runtime_error e) {`*`error handling`*`}`

(continued)

An example is found in lines 9–25 of Program `harmonic.cpp`. If an error which is referenced by `throw` occurs in *`subprogram`*, then *`subprogram`* is terminated and a test is run as to whether there is a `catch` instruction matching the error type and if so, the part *`error handling`* is executed. Alongside `runtime_error` several other error types are defined in `stdexcept`. They all have one property in common, namely that one can call up the error message text stored in the `throw` instruction by means of `.what()`.

As soon as a `throw` instruction has been executed, the program continues with the error handling part of a matching `catch` instruction. If no matching `catch` instruction is available, the program returns recursively to the calling functions until a matching `catch` instruction is found. If none of the calling functions contains a matching `catch` instruction, the program terminates.

Finally, line 20 of `harmonic.cpp` shows the conversion of an integer into the data type `double`. One can in general convert an expression into another data type by means of `static_cast<`*`Datatype`*`> (`*`Expression`*`)`.

The class `Fraction` has not yet been properly implemented. At least, zero denominators are intercepted. But until now, nothing happened in the case of range overflows of `inttype`. It is, however, exactly these that can quickly occur; Program `harmonic.cpp`, for example, only works correctly for $n \leq 20$.

Of course one could, and should, always reduce all occurring fractions. This will be dealt with in the next chapter. But even this does not always prevent overflows. One could intercept these with `throw`, but it would of course be better if numerators and denominators can become arbitrarily large.

2.5 Arbitrarily Large Integers

We will now investigate the structure of a class that is able to represent arbitrarily large integers. Strictly speaking, we would have to make available all the standard operations of arithmetic as well as the comparison operators, as with standard data types like `int`. For simplicity we will consider only the operators +, +=, <, as well as one output function. The += operator enables one to write expressions of the form `a = a + b;` more briefly as `a += b;`.

For the sake of clarity Program 2.11 is divided into three parts. The files `largeint.h` and `largeint.cpp` define a new type (class) called `LargeInt`.

The file `largeint.h` contains the declaration of the class (consisting of the parts `public` and `private`). The file `largeint.cpp` contains the implementation of all the functions belonging to the class. The division into a header file and an implementation file is quite usual and also practical when the classes are large.

The class LargeInt stores an arbitrarily large natural number in a vector called _v, where every place of the vector corresponds to a decimal place of the number. In addition, there is a constant string called digits, which is required for the numerical representation change. It is, however, not necessary for every object of type LargeInt to contain a copy of this string. This is the reason for the prefix static, which ensures that digits is stored only once for all objects in the class. The initialization of a static variable in a class must take place outside the class. In our case this happens in line 5 of largeint.cpp. Users of the class need not, however, know all these details, as they are all in the private part. Only the following functions are visible from the outside:

- One can form a new variable of type LargeInt by means of the constructor. Its argument is the number to be stored. A constructor automatically calls the constructors for the variables contained in the class; in this instance the constructor of vector.
- The function decimal outputs the decimal representation of the number in the form of a string.
- The comparison operator < is implemented and it tests whether or not the stored value is less than the one in the argument.
- The operator += is implemented and is used for implementing the operator +. This enables one to add numbers of type LargeInt.

Program 2.11 (Arbitrarily Large Integers)

```
1  // largeint.h (Declaration of Class LargeInt)
2
3  #include <vector>
4  #include <string>
5
6
7  class LargeInt {
8  public:
9      using inputtype = long long;
10
11     LargeInt(inputtype);                          // constructor
12     std::string decimal() const;                  // decimal representation
13     bool operator<(const LargeInt &) const;       // comparison
14     LargeInt operator+(const LargeInt &) const;   // addition
15     const LargeInt & operator+=(const LargeInt &); // addition
16
17 private:
18     std::vector<short> _v;  // store single digits, last digit in _v[0]
19     static const std::string digits;
20 };
```

```
1  // largeint.cpp (Implementation of Class LargeInt)
2
3  #include "largeint.h"
4
5  const std::string LargeInt::digits = "0123456789";
6
7  LargeInt::LargeInt(inputtype i)   // constructor, calls constructor of vector
8  {
9     do {
10        _v.push_back(i % 10);
11        i /= 10;
```

```
    } while (i > 0);
}


std::string LargeInt::decimal() const   // returns decimal representation
{
    std::string s("");
    for (auto i : _v) {         // range for statement: i runs over all
        s = digits[i] + s;      // elements of _v
    }
    return s;
}


bool LargeInt::operator<(const LargeInt & arg) const   // checks if < arg
{
    if (_v.size() == arg._v.size()) {
      auto it2 = arg._v.rbegin();
      for (auto it1 = _v.rbegin(); it1 != _v.rend(); ++it1, ++it2) {
        if (*it1 < *it2) return true;
        if (*it1 > *it2) return false;
      }
      return false;
    }
    return _v.size() < arg._v.size();
}


LargeInt LargeInt::operator+(const LargeInt & arg) const  // addition
{
    LargeInt result(*this);
    result += arg;
    return result;
}


const LargeInt & LargeInt::operator+=(const LargeInt & arg)   // addition
{
    if (arg._v.size() > _v.size()) {
        _v.resize(arg._v.size(), 0);
    }
    auto it1 = _v.begin();
    for (auto it2 = arg._v.begin(); it2 != arg._v.end(); ++it2, ++it1) {
        *it1 += *it2;
    }
    short carry = 0;
    for (auto & i : _v) {
        i += carry;
        carry = i / 10;
        i %= 10;
    }
    if (carry != 0) _v.push_back(carry);
    return *this;
}
```

```
// factorial.cpp (Computing n! by Addition)

#include <iostream>
#include <limits>
#include "largeint.h"


LargeInt factorial(LargeInt::inputtype n)
// computes n! warning, slow: runtime O(n^3 log n)
{
    LargeInt result(1);
    for (LargeInt::inputtype i = 2; i <= n; ++i) {
        LargeInt sum(0);
        for (LargeInt::inputtype j = 1; j <= i; ++j) {
            sum += result;
        }
        result = sum;
    }
    return result;
}


int main()
{
    LargeInt::inputtype n;
    std::cout << "This program computes n!, for a natural number n up to "
              << std::numeric_limits<LargeInt::inputtype>::max() << "\n"
              << "Enter a natural number n: ";
    std::cin >> n;
    std::cout << n << "! = " << factorial(n).decimal() << "\n";
}
```

The constructor for `LargeInt` attaches elements stepwise to the end of the vector `_v` by means of `_v.push_back`. In lines 50 and 51 of `largeint.cpp` we see two new `vector` functions. The number of elements contained in a `vector` is returned by the function `size()`. One can change the size of a `vector` with `resize`. The first argument of this function is the number of elements and the optional second argument is a set of initial values for the new elements introduced by `resize`.

Addition as well as comparison of two `LongInts` necessitate running through all the elements of the relevant `vectors`. One could do this by accessing the various elements of `vector` by their indices. A more universal method, however, is based on the use of so-called iterators which are explained in more detail in the box **C++ in Detail (2.3)**.

C++ in Detail (2.3): Iterators and the Range-Based `for`

The C++ Standard Library provides several more so-called container types apart from the abstract data type `vector`. One can access the elements of a `vector` with the index operator `[]`. But this is not necessarily the case for other container types. In order to write code which is as independent of the used container type as possible, one can access the elements of a container by means of iterators. The function `begin()` yields an iterator referencing the first element of the container, the function `end()` yields an iterator referencing the next place after the last element of the container. In line 53 of `largeint.cpp` we have defined an iterator called `it1`, to which the value `_v.begin()` is assigned. Here we have used the type `auto`. This keyword orders the compiler to independently find the matching type by means of the assigned value. One can increase and decrease an iterator with `++` and `--`.

(continued)

This causes it to reference the next or the previous element, respectively. If one uses the functions `rbegin()` and `rend()` instead of `begin` and `end`, one can run through all the elements of the containers in reverse order by means of `++`. We have used this in line 30 of `largeint.cpp`, for example. With `*iter` one obtains the current element referenced by the iterator `iter`.

It often happens that one wants to run through all the elements of a container. For this purpose the C++ Standard Library provides a special version of the `for` instruction, called the range-based `for` instruction. We have used this in lines 19–21 of `largeint.cpp`, for example. The variable `i` defined there runs through all the elements of vector `_v`. Alternatively we could also have written the loop in lines 19–21 in the following way:

```
for (auto it = _v.begin(); it != _v.end(); ++it) {
    s = digits[*it] + s;
}
```

Note the following: in a range-based `for` loop one runs through all the elements of the container, hence one does not use the dereferencing operator `*` as one does with iterators. If one also wants to be able to change the elements that the variable runs through, then the variable must be defined by means of `&` as a reference to these elements. We have used this in the `for` loop in lines 58–62 of `largeint.cpp`, for example.

The addition algorithm for two `LargeInt` numbers is initiated with the `+=` operator in lines 48–65 of `largeint.cpp`. This operator is then used in lines 40–45 for initiating the `+` operator. This is done by first storing in line 42 a copy of the object to the left of the `+` operator. In order to do this, we must reference the object with which the `+` operator was called. We do this by using `this`, which contains the address of the object with which the corresponding function was called.

When there are several different variables of type `LargeInt`, as for example in the function `main` in the file `factorial.cpp`, these clearly occupy different areas of the memory. During every run through the `for` loop, a new `LargeInt` variable called `sum` is created in line 12 of `factorial.cpp`. At the end of the `for` loop this variable is deleted again by calling the destructor of `LargeInt`.

In our case all this takes place automatically: as we did not define our own destructor, the standard destructor is called (automatically); the latter then simply calls the destructor of `vector`. As `vector` puts data on top of the heap, this class has a more complicated destructor for vacating the relevant place in the memory.

The implementation of `factorial.cpp` is not very efficient and could clearly be simplified and improved by using multiplication. This, as well as other basic operations of arithmetic, can be supplemented in the class `LargeInt`. At the moment the running time is dominated by $\Theta(n^2)$ additions in each of which the larger summand has $\Theta(n \log n)$ digits, i.e. we have $\Theta(n^3 \log n)$ elementary operations. If one implements multiplication, one can compute $n!$ with $\Theta(n)$

multiplications in each of which one factor has at most $n \log n$ digits and the other at most $\log n$ digits; this yields a total running time of $O(n^2 \log^2 n)$ (if multiplication is implemented with the school method).

Such a new class is very easy to use. If, for example, one wants to implement the Collatz sequence (Program 1.25) for arbitrarily large numbers, one just has to replace line 5 of that program by `using myint = LargeInt;` and, of course, insert `#include "largeint.h"` as well. Also our class `Fraction` can work with all fractions by exchanging just one line (replace line 7 in `fraction.h` by `using inttype = LargeInt;`). Of course, we must first implement the missing operations, in particular multiplication.

3 Computing with Integers

In Chap. 1 we called the basic operations of arithmetic elementary operations and assumed that they run in constant time. It is often practical to make this assumption, although it is clearly by no means a realistic one for arbitrarily large numbers. Here we will investigate how fast one can actually compute with integers. Here, in a narrower sense, elementary operations only include operations and comparisons restricted to integers in some bounded interval.

Modern computers are able to perform elementary operations on numbers of up to 64 bits in just a few clock cycles. For the asymptotic running time this is, however, irrelevant here, as one can compose everything out of elementary operations with just one bit.

3.1 Addition and Subtraction

In Chap. 2 we have already seen how to perform the addition of two positive integers. An implementation of this was presented in Program 2.11. The algorithm clearly has running time $O(l)$, where l is the maximum number of digits of the two summands. By using either the sign or the 2's complement representation to represent negative numbers, one can easily extend this algorithm to work for negative numbers or for subtraction. As the 2's representation can be computed in $O(l)$ time, the running time of the extended algorithm remains $O(l)$. We summarize:

Proposition 3.1 *For two integers x and y their sum $x + y$ and difference $x - y$ can be computed in running time $O(l)$, where $l = 1 + \lfloor \log_2(\max\{|x|, |y|, 1\}) \rfloor$.* $\square$

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_3

3.2 Multiplication

For multiplication and division the obvious algorithms (school method) have running time $O(l^2)$. But one can do better.

Karatsuba [21] was the first to discover how one can do multiplication asymptotically faster. The idea is to divide the two l-digit multiplicands x and y into two roughly $\frac{l}{2}$-digit numbers, for example by putting $x = x' \cdot B + x''$ and $y = y' \cdot B + y''$, where B is a power of the base of the numerical representation used. It then suffices to compute the products $p := x' \cdot y'$, $q := x'' \cdot y''$ and $r := (x' + x'') \cdot (y' + y'')$ (recursively with the same algorithm) and then to obtain the desired product $x \cdot y$ by means of the following equation:

$$x \cdot y = (x' \cdot y')B^2 + (x' \cdot y'' + x'' \cdot y')B + x'' \cdot y'' = pB^2 + (r - p - q)B + q\,. \qquad (3.1)$$

From this point onwards we will assume that all numbers are represented to the base 2. Below is Karatsuba's algorithm in pseudocode. Clearly we can restrict ourselves to the natural numbers.

Algorithm 3.2 (Karatsuba's Algorithm)

Input: $x, y \in \mathbb{N}$, represented in binary notation.
Output: $x \cdot y$, represented in binary notation.

```
if $x < 8$ and $y < 8$ then output $x \cdot y$ (direct multiplication)
else
    $l \leftarrow 1 + \lfloor \log_2(\max\{x, y\}) \rfloor$, $k \leftarrow \lfloor \frac{l}{2} \rfloor$
    $B \leftarrow 2^k$
    $x' \leftarrow \lfloor \frac{x}{B} \rfloor, x'' \leftarrow x \mod B$
    $y' \leftarrow \lfloor \frac{y}{B} \rfloor, y'' \leftarrow y \mod B$
    $p \leftarrow x' \cdot y'$   (recursively)
    $q \leftarrow x'' \cdot y''$   (recursively)
    $r \leftarrow (x' + x'') \cdot (y' + y'')$   (recursively)
    output $pB^2 + (r - p - q)B + q$
```

Theorem 3.3 *The multiplication of two natural numbers x and y given in binary notation can be performed with Algorithm 3.2 in running time $O(l^{\log_2 3})$, where $l =$* $1 + \lfloor \log_2(\max\{x, y\}) \rfloor$.

Proof The correctness of the algorithm follows immediately from Eq. (3.1).

In order to prove the running time, we will first show that there exists a constant $c \in \mathbb{N}$ such that the number $T(l)$ of elementary steps (in the narrower sense) of Algorithm 3.2 can be bounded above by bounds dependent on l (the maximum number of digits in the numbers x and y) as follows:

$$\begin{aligned} T(l) &\le c && \text{for } l \le 3 \\ T(l) &\le c \cdot l + 3T(\lceil \tfrac{l}{2} \rceil + 1) && \text{for } l \ge 4\,. \end{aligned} \tag{3.2}$$

For the proof of the second inequality, one needs to use the fact that the three recursive calls of the algorithm are performed with numbers that each have at most $\lceil \frac{l}{2} \rceil + 1$ digits. This is clear for x'' and y''; they even have at most $k = \lfloor \frac{l}{2} \rfloor$ digits. Furthermore, we have $1 + \lfloor \log_2 x' \rfloor \le 1 + \lfloor \log_2 \frac{x}{B} \rfloor = 1 + \lfloor \log_2 x \rfloor - k \le l - k = \lceil \frac{l}{2} \rceil$, so x', and analogously y', have at most $\lceil \frac{l}{2} \rceil$ digits. Hence also $x' + x''$ and $y' + y''$ each have at most $\lceil \frac{l}{2} \rceil + 1$ digits. As this is less than l for all $l \ge 4$, the recursion terminates.

All other operations require only $O(l)$ steps; here one uses the fact that B is a power of 2 and that x and y (as well as all intermediate results) are given in binary notation. Of course, one also uses Proposition 3.1.

The recursion (3.2) can be solved as follows. For a start we assert that we have for all $m \in \mathbb{N} \cup \{0\}$:

$$T(2^m + 2) \;\le\; c \cdot (4 \cdot 3^m - 2 \cdot 2^m - 1) \tag{3.3}$$

This is proved very simply by induction over m: for $m = 0$ we have $T(3) \le c$ by (3.2). For $m \in \mathbb{N}$ we have, also by (3.2), that $T(2^m + 2) \le c \cdot (2^m + 2) + 3T(2^{m-1} + 2)$, and by the induction hypothesis this is bounded above as follows:

$$T(2^m + 2) \le c \cdot (2^m + 2 + 3 \cdot (4 \cdot 3^{m-1} - 2 \cdot 2^{m-1} - 1)) = c \cdot (4 \cdot 3^m - 2 \cdot 2^m - 1)\,,$$

which proves (3.3).

For an arbitrary $l \in \mathbb{N}$ with $l > 2$, let $m := \lceil \log_2(l-2) \rceil < 1 + \log_2 l$. Then the following holds: $T(l) \le T(2^m + 2) < 4c \cdot 3^m < 12c \cdot 3^{\log_2 l} = 12c \cdot l^{\log_2 3}$. □

Note that $\log_2 3$ is less than 1.59 and that the algorithm is therefore (especially with two numbers of roughly equal length) substantially faster than the school method with running time $O(l^2)$.

In practice one also multiplies larger numbers directly (with up to 32 or 64 binary places: such large numbers can be multiplied by today's processors in just a few clock cycles) and only chooses the recursive approach of Karatsuba for even larger numbers.

Still faster algorithms for multiplication were found by Schönhage and Strassen [30] and then by Fürer [17]. The (integer) division can be reduced to multiplication and be performed equally quickly; we will deal with this in Sect. 5.5.

3.3 The Euclidean Algorithm

The Euclidean Algorithm (defined by Euclid around 300 BC in Book VII of his *Elements*) enables one to compute efficiently the greatest common divisor of two numbers and thus to reduce fractions to lowest terms.

Definition 3.4 For $a, b \in \mathbb{N}$ the greatest common divisor of a and b is denoted by $\gcd(a, b)$ and is the largest natural number that divides a as well as b. If $\gcd(a, b) = 1$, then a and b are said to be coprime. We also define $\gcd(a, 0) := \gcd(0, a) := a$ for all $a \in \mathbb{N}$ (every natural number divides 0) and $\gcd(0,0) := 0$.

For $a, b \in \mathbb{N}$ the least common multiple of a and b is denoted by $\mathrm{lcm}(a, b)$ and is the smallest natural number that is divisible by a as well as by b.

Lemma 3.5 *For all $a, b \in \mathbb{N}$ the following statements hold:*

(a) $a \cdot b = \gcd(a, b) \cdot \mathrm{lcm}(a, b)$*;*
(b) $\gcd(a, b) = \gcd(a \bmod b, b)$.

Proof

(a) For every common divisor x of a and b (and thus also for $x = \gcd(a, b)$), $\frac{ab}{x}$ is a common multiple of a and b, hence "$\geq$" holds.
Conversely, $\frac{ab}{\mathrm{lcm}(a,b)}$ is a common divisor of a and b, hence $\gcd(a, b) \geq \frac{ab}{\mathrm{lcm}(a,b)}$.
(b) If x divides both a and b, then it clearly also divides $a - b\lfloor\frac{a}{b}\rfloor = a \bmod b$. Conversely, if x divides $a \bmod b$ and b, then it also divides $(a \bmod b) + b\lfloor\frac{a}{b}\rfloor = a$. □

Lemma 3.5(b) immediately yields the correctness of

Algorithm 3.6 (Euclidean Algorithm)

Input: $a, b \in \mathbb{N}$.
Output: $\gcd(a, b)$.

```
while $a > 0$ and $b > 0$ do
    if $a < b$ then $b \leftarrow b \bmod a$ else $a \leftarrow a \bmod b$
output $\max\{a, b\}$
```

Example 3.7 $\gcd(6314, 2800) = \gcd(2800, 714) = \gcd(714, 658) = \gcd(658, 56) = \gcd(56, 42) = \gcd(42, 14) = \gcd(14, 0) = 14$.

In Program 3.8 the Euclidean Algorithm is implemented recursively in the function `gcd`. One could also integrate this function into our class `LargeInt` (if the operator `%` has been implemented for `LargeInt`). A class `LargeInt` which has been extended by such a gcd-function can then be used as a type for numerators and denominators in `Fraction` and can always reduce all fractions to lowest terms after every elementary operation.

Program 3.8 (Euclidean Algorithm)

```cpp
// euclid.cpp (Euclidean Algorithm)

#include <iostream>

using myint = long long;

myint gcd(myint a, myint b)          // compute greatest common divisor
{                                    // using Euclidean algorithm
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}


int main()
{
    myint a, b;
    std::cout << "This program computes the greatest common divisor.\n"
              << "Enter two natural numbers, separated by blank: ";
    std::cin >> a >> b;
    std::cout << "gcd(" << a << "," << b << ") = " << gcd(a,b) << "\n";
}
```

The number of iterations (or recursive calls) of the Euclidean Algorithm can easily be bounded by $2 + \lfloor \log_2 \max\{a, 1\} \rfloor + \lfloor \log_2 \max\{b, 1\} \rfloor$ as at least one of the two numbers is halved, or decreased even more, in each iteration (with the possible exception of the first in `gcd`).

For a more careful analysis of the running time of Algorithm 3.6 we need the **Fibonacci numbers** F_i, $i = 0, 1, 2, \ldots$, defined by $F_0 := 0$, $F_1 := 1$ and

$$F_{n+1} := F_n + F_{n-1} \quad \text{for } n \in \mathbb{N}\,.$$

The first few Fibonacci numbers are thus 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. The Fibonacci numbers were in fact known in India more than 400 years before Fibonacci (ca. 1170–1240) introduced them.

Lemma 3.9

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

for all $n \in \mathbb{N} \cup \{0\}$.

Proof The proof is by induction over n. The formula is clearly correct for $n = 0$ and $n = 1$. For $n \geq 2$ we have

$$\begin{aligned}
F_n &= F_{n-1}+F_{n-2} \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \cdot \left(\frac{1+\sqrt{5}}{2}+1\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2} \cdot \left(\frac{1-\sqrt{5}}{2}+1\right)\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right),
\end{aligned}$$

as $\frac{1+\sqrt{5}}{2} + 1 = \frac{3+\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^2$ and $\frac{1-\sqrt{5}}{2} + 1 = \frac{3-\sqrt{5}}{2} = \left(\frac{1-\sqrt{5}}{2}\right)^2$. □

Lemma 3.10 *If $a > b > 0$ and Algorithm 3.6 performs $k \geq 1$ iterations of the* **while** *loop, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.*

Proof The proof is by induction over k. The statement holds for $k = 1$ because $b \geq 1 = F_2$ and $a > b \Rightarrow a \geq 2 = F_3$.

Now let $k \geq 2$. As $a > b \geq 1$ and $k \geq 2$, the next iteration will use the numbers b and $a \bmod b$. To these we can apply the induction hypothesis, because $k-1$ iterations remain and $b > a \bmod b > 0$ (as $k - 1 > 0$). Thus $b \geq F_{k+1}$ and $a \bmod b \geq F_k$. Hence $a = \lfloor a/b \rfloor \cdot b + (a \bmod b) \geq b + (a \bmod b) \geq F_{k+1} + F_k = F_{k+2}$. □

Theorem 3.11 (Lamé's Theorem) *If $a \geq b$ and $b < F_{k+1}$ for some $k \in \mathbb{N}$, then Algorithm 3.6 performs less than k iterations of the* **while** *loop.*

Proof If $b = 0$, then the algorithm does not perform a single iteration, and if $a = b > 0$, it performs exactly one. The remaining cases follow immediately from Lemma 3.10. □

Remark 3.12 If one computes $\gcd(F_{k+2}, F_{k+1})$ for some $k \in \mathbb{N}$, one needs exactly k iterations, because $\gcd(F_3, F_2) = \gcd(2,1)$ requires one iteration and $\gcd(F_{k+2}, F_{k+1})$ with $k \geq 2$ is reduced to $\gcd(F_{k+1}, F_{k+2}-F_{k+1}) = \gcd(F_{k+1}, F_k)$. So the statement of Theorem 3.11 is best possible.

Corollary 3.13 *If $a \geq b > 1$, then Algorithm 3.6 performs at most $\lceil \log_\phi b \rceil$ iterations, where $\phi = \frac{1+\sqrt{5}}{2}$. The total running time of Algorithm 3.6 for computing* $\gcd(a, b)$ *is $O((\log a)^3)$.*

Proof We first show by induction over n that $F_n > \phi^{n-2}$ for all $n \geq 3$. For $n = 3$ we have $F_3 = 2 > \phi$. As $1 + \phi = \phi^2$, applying the induction hypothesis for $n \geq 3$ yields

$$F_{n+1} = F_n + F_{n-1} > \phi^{n-2} + \phi^{n-3} = \phi^{n-3}(\phi + 1) = \phi^{n-1} .$$

For the proof of the first statement of Corollary 3.13, let $k := 1 + \lceil \log_\phi b \rceil \geq 2$. Then we have $b \leq \phi^{k-1} < F_{k+1}$ and the statement follows with Theorem 3.11. In every iteration one must compute $x \bmod y$ for two numbers x and y with $a \geq x \geq y$. With the school method this works in $O((\log a)^2)$ time and thus the total running time is $O((\log a)^3)$. □

Remark 3.14 The constant $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618\ldots$ is known as the golden ratio. It follows that $\frac{1-\sqrt{5}}{2} \approx -0.618\ldots$, hence $F_n \approx \frac{1}{\sqrt{5}} \cdot \phi^n$ by Lemma 3.9.

Approximate Representations of the Real Numbers 4

By combining the two classes `LargeInt` (Program 2.11) and `Fraction` (Program 2.10)—extended by adding the missing operations—one can work with rational numbers without the occurrence of rounding errors. The elementary operations are, however, comparatively slow because both numerators and denominators can get very large, even if one always reduces fractions to lowest terms with the Euclidean Algorithm.

Computations become substantially faster when one uses standard data types like `double`; here, however, one will encounter rounding errors and will have to control these.

Complex numbers can, of course, be stored (approximately) by storing the real and imaginary parts (approximately), similar to rational numbers where we store the numerator and denominator as a pair of integers. One can also define a class here as with the class `Fraction`. The C++ Standard Library in fact contains a type `complex<double>` which one can use for complex numbers. To keep things simple, we will consider only real numbers; clearly everything can be easily modified to apply to complex numbers.

4.1 The b-Adic Representation of the Real Numbers

The b-adic representation of the natural numbers can be generalized to the real numbers as follows:

Theorem 4.1 (The b-Adic Representation of the Real Numbers) *Let $b \in \mathbb{N}$ with $b \geq 2$ and $x \in \mathbb{R}\backslash\{0\}$. Then there exist uniquely defined numbers $E \in \mathbb{Z}$, $\sigma \in \{-1,1\}$ and $z_i \in \{0, 1, \dots, b-1\}$ for $i \in \mathbb{N} \cup \{0\}$ with $\{i \in \mathbb{N} : z_i \neq b-1\}$ an infinite set,*

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_4

$z_0 \neq 0$ *and*

$$x = \sigma \cdot b^E \cdot \left(\sum_{i=0}^{\infty} z_i \cdot b^{-i} \right) . \tag{4.1}$$

(4.1) *is called the (normalized)* ***b*-adic representation** *of x.*

Here we use the usual notation for infinite series as the limit of the sequence of partial sums: $\sum_{i=0}^{\infty} a_i := \lim_{n\to\infty} \sum_{i=0}^{n} a_i$. Note that the limit exists because the sequence is increasing and bounded from above by b.

Proof Let $x \in \mathbb{R} \setminus \{0\}$. Clearly, $\sigma = -1$ for $x < 0$ and $\sigma = 1$ for $x > 0$.

For $z_i \in \{0, 1, \dots, b-1\}$ ($i \in \mathbb{N} \cup \{0\}$) with $\{i \in \mathbb{N} : z_i \neq b-1\} \neq \emptyset$ and $z_0 \neq 0$ we have

$$1 \leq \sum_{i=0}^{\infty} z_i \cdot b^{-i} < \sum_{i=0}^{\infty} (b-1) \cdot b^{-i} = b .$$

So we must set $E := \lfloor \log_b |x| \rfloor$.

Put $a_0 := b^{-E}|x|$ and define recursively for $i \in \mathbb{N} \cup \{0\}$:

$$z_i := \lfloor a_i \rfloor \qquad \text{and} \qquad a_{i+1} := b(a_i - z_i) .$$

Note that $1 \leq a_0 < b$ and $0 \leq a_i < b$ for $i \in \mathbb{N}$, hence $z_i \in \{0, \dots, b-1\}$ for all i and $z_0 \neq 0$.

Claim: $a_0 = \sum_{i=0}^{n} z_i \cdot b^{-i} + a_{n+1} b^{-n-1}$ for all $n \in \mathbb{N} \cup \{0\}$.

The proof of the claim is by induction over n. For $n = 0$ we clearly have $a_0 = z_0 b^{-0} + a_1 b^{-1}$.

Now let $n \in \mathbb{N}$. By the induction hypothesis (applied to $n-1$) we have $a_0 = \sum_{i=0}^{n-1} z_i \cdot b^{-i} + a_n b^{-n}$. As $a_{n+1} = b(a_n - z_n)$, we obtain $a_0 = \sum_{i=0}^{n-1} z_i \cdot b^{-i} + z_n b^{-n} + a_{n+1} b^{-n-1}$, which proves the claim.

The claim then yields

$$x = \sigma \cdot b^E \cdot a_0 = \sigma \cdot b^E \cdot \lim_{n\to\infty} \left(\sum_{i=0}^{n} z_i \cdot b^{-i} \right) ,$$

which is (4.1).

Suppose that $\{i \in \mathbb{N} : z_i \neq b-1\}$ is a finite set, i.e. there exists an $n_0 \in \mathbb{N}$ with $z_i = b-1$ for all $i > n_0$. Then it follows that

$$a_0 = \sum_{i=0}^{n_0} z_i \cdot b^{-i} + \sum_{i=n_0+1}^{\infty} (b-1) \cdot b^{-i} = \sum_{i=0}^{n_0} z_i \cdot b^{-i} + b^{-n_0}.$$

But the claim implies that $a_{n_0+1} = b$, a contradiction. This proves the existence.

It remains to show the uniqueness. We have already shown the uniqueness of σ and E. Suppose we have the two following representations:

$$x = \sigma \cdot b^E \cdot \sum_{i=0}^{\infty} y_i \cdot b^{-i} = \sigma \cdot b^E \cdot \sum_{i=0}^{\infty} z_i \cdot b^{-i}$$

with the required properties. Let n be the smallest index with $y_n \neq z_n$. Without loss of generality we can set $y_n + 1 \leq z_n$. Then we obtain

$$\begin{aligned}
\frac{x}{\sigma b^E} &= \sum_{i=0}^{\infty} y_i \cdot b^{-i} \\
&\leq \sum_{i=0}^{n-1} y_i \cdot b^{-i} + (z_n - 1)b^{-n} + \sum_{i=n+1}^{\infty} y_i b^{-i} \\
&< \sum_{i=0}^{n-1} y_i \cdot b^{-i} + (z_n - 1)b^{-n} + \sum_{i=n+1}^{\infty} (b-1) b^{-i} \\
&= \sum_{i=0}^{n-1} z_i \cdot b^{-i} + z_n b^{-n} \\
&\leq \sum_{i=0}^{\infty} z_i \cdot b^{-i} \\
&= \frac{x}{\sigma b^E} ,
\end{aligned}$$

a contradiction. □

Example 4.2 For $b = 2$ we have the two following b-adic representations:

$$\begin{aligned}
\tfrac{1}{3} &= 2^{-2} \cdot (2^0 + 2^{-2} + 2^{-4} + 2^{-6} + \cdots) \;=\; (0.\overline{01})_2 \\
8.1 &= 2^3 \cdot (2^0 + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} \cdots) \;=\; (1000.0\overline{0011})_2 .
\end{aligned}$$

4.2 Machine Numbers

As a real number can only be represented by a finite number of digits in a computer, one uses the following representation suggested by Theorem 4.1:

Definition 4.3 Let $b, m \in \mathbb{N}$, with $b \geq 2$. A b-adic m-digit **normalized floating-point number** has the form

$$x = \sigma \cdot b^E \cdot \left(\sum_{i=0}^{m-1} z_i \cdot b^{-i} \right)$$

where $\sigma \in \{-1,1\}$ is its sign, $E \in \mathbb{Z}$ its exponent, and $\sum_{i=0}^{m-1} z_i \cdot b^{-i}$ its mantissa with $z_0 \neq 0$ and $z_i \in \{0, \dots, b-1\}$ for all $i \in \mathbb{N} \cup \{0\}$.

Example 4.4 For $b = 10$, the number 136.5 has the normalized 5-digit representation $136.5 = 1.3650 \cdot 10^2$. In C++ notation this is written as `1.365e+2`.

For $b = 10$, the number $\frac{1}{3}$ has no representation as an m-digit normalized floating-point number for any $m \in \mathbb{N}$. The number 0 has no representation at all as a normalized floating-point number.

For the representation of real numbers in a computer, one has to choose suitable values for b, m and the range $\{E_{\min}, \dots, E_{\max}\}$ of E. These parameters fix the so-called **machine-number range**

$$F(b, m, E_{\min}, E_{\max})$$

which comprises the set of b-adic m-digit normalized floating-point numbers with exponent $E \in \{E_{\min}, \dots, E_{\max}\}$ and the number 0. The elements of a (fixed) machine-number range are called **machine numbers**.

In 1985 the IEEE (*Institute of Electrical and Electronics Engineers*) adopted the IEEE-Standard 754 [19] in which, for example, a data type `double` with $b = 2$, $m = 53$ and $E \in \{-1022, \dots, 1023\}$ is defined. We will write $F_{\texttt{double}} := F(2, 53, -1022, 1023)$. For the representation of these numbers 64 bits suffice: one bit for the sign, 52 bits $z_1 \cdots z_{52}$ for the mantissa and 11 bits for the exponent E. As normalization ensures that the zeroth bit z_0 always has the value 1, one can omit it (i.e. it is a "hidden bit").

The exponent is represented by means of the so-called bias representation: a constant (the bias) is added in order to make all exponents positive. For the data type `double` defined above, the bias is 1023. The C++ Standard Library does not prescribe how to implement the data type `double`, so one usually proceeds as described in IEEE 754 mentioned above. This is the procedure adopted below.

Filing the normalized floating-point representation in bits takes the following form:

1 bit	11 bits	52 bits
$\frac{1+\sigma}{2}$	$E + 1023$	$z_1\, z_2\ \dots\ z_{52}$

The "free" exponents -1023 and 1024 (i.e. 0 and 2047 in the bias representation) are used in order to encode ± 0 and $\pm\infty$ (i.e. the exponent is greater than 1023), as well as for subnormal numbers (i.e. the exponent is less than -1022) and NaN (**N**ot **a** **N**umber) which indicates the result of an infeasible calculation, e.g. $\log(-1)$.

The largest number representable in this way is

$$\begin{aligned}\max F_{\texttt{double}} &= \texttt{std::numeric\_limits<double>::max()}\\ &= 1.\underbrace{1\dots1}_{52}\cdot 2^{1023} = (2-2^{-52})\cdot 2^{1023} = 2^{1024} - 2^{971} \approx 1.797693\cdot 10^{308}\,.\end{aligned}$$

The smallest positive machine number in $F_{\texttt{double}}$ is

$$\begin{aligned}\min\{f \in F_{\texttt{double}}, f > 0\} &= \texttt{std::numeric\_limits<double>::min()}\\ &= 1.0\cdot 2^{-1022} \approx 2.225074\cdot 10^{-308}\,.\end{aligned}$$

We define $\operatorname{range}(F) := [\min\{f \in F, f > 0\}, \max F]$, which is the smallest interval containing all the representable positive numbers.

In comparison, the C++ data type `float` usually uses only 32 bits and thus has a much smaller range and greatly reduced accuracy. It is therefore rarely recommended. The data type `long double`, on the other hand, is usually more accurate than `double`. Often, 80 bits are used for representing numbers in `long double` (64 bits for the mantissa and 16 bits for the exponent), as the FPUs (*floating-point unit*s) in today's processors frequently use such a representation of floating-point numbers. An FPU can perform elementary operations on such floating-point numbers in just a few clock cycles.

The data type `double` is the standard data type in C++ for the approximate representation of real numbers.

4.3 Rounding

As the set of machine numbers is finite, one has to accept rounded representations of real numbers:

Definition 4.5 Let $F = F(b, m, E_{\min}, E_{\max})$ be a machine-number range. A function $\operatorname{rd}\colon \mathbb{R} \to F$ is called a **rounding** (to F) if the following holds for all $x \in \mathbb{R}$:

$$|x - \operatorname{rd}(x)| = \min_{a\in F} |x - a|\,.$$

Note that this definition allows more than one rounding. In $F(10, 2, 0, 2)$, for example, the number 12.5 can be rounded down to 12 or up to 13. In commercial rounding there is no ambiguity: 12.5 is rounded up to 13. According to IEEE 754, however, x is rounded so that the last digit becomes even, e.g. $\mathrm{rd}(12.5) = 12$ and $\mathrm{rd}(13.5) = 14$. One reason for this is, that in calculations like $(\cdots(((12 + 0.5) - 0.5)+0.5)-\ldots$ one alternately rounds up and then down. With commercial rounding this series diverges.

Definition 4.6 Let $x, \tilde{x} \in \mathbb{R}$, where $\tilde{x}$ is an approximation of x. Then $|x-\tilde{x}|$ is called the **absolute error**. If $x \neq 0$, then $\left|\frac{x-\tilde{x}}{x}\right|$ is called the **relative error**.

The "machine epsilon" is a measure of machine accuracy and is given by the maximum relative error that can occur when rounding numbers in the range:

Definition 4.7 Let F be a machine-number range. The **machine epsilon** of F is denoted by **eps(F)** and is defined as follows:

$$\mathrm{eps}(F) := \sup\left\{\left|\frac{x - \mathrm{rd}(x)}{x}\right| : x \in \mathbb{R},\ |x| \in \mathrm{range}(F),\ \mathrm{rd}\text{ a rounding to } F\right\} .$$

Theorem 4.8 *For every machine-number range* $F = F(b, m, E_{\min}, E_{\max})$ *with* $E_{\min} < E_{\max}$ *we have*

$$\mathrm{eps}(F) = \frac{1}{1 + 2b^{m-1}} .$$

Proof Put $x = b^{E_{\min}} \cdot \left(1 + \frac{1}{2}b^{-m+1}\right)$. Then $x \in \mathrm{range}(F)$ and for every rounding rd to F we have

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| = \frac{|x - \mathrm{rd}(x)|}{x} = \frac{b^{E_{\min}} \cdot \frac{1}{2}b^{-m+1}}{b^{E_{\min}} \cdot \left(1 + \frac{1}{2}b^{-m+1}\right)} = \frac{1}{1 + 2b^{m-1}} ,$$

whence $\mathrm{eps}(F) \geq \frac{1}{1+2b^{m-1}}$.

For the reverse direction it suffices to consider only positive x. So let $x \in \mathrm{range}(F)$ but $x \notin F$ (otherwise there is no rounding error), and let $x = b^E \cdot \sum_{i=0}^{\infty} z_i \cdot b^{-i}$ be the normalized b-adic representation of x. Then

$$x' = b^E \cdot \sum_{i=0}^{m-1} z_i \cdot b^{-i} \qquad \text{and} \qquad x'' = b^E \cdot \left(\sum_{i=0}^{m-1} z_i \cdot b^{-i} + b^{-m+1}\right)$$

both lie in F, and $x' < x < x''$ holds.

For every rounding rd to F we have $|x - \mathrm{rd}(x)| \le \frac{1}{2}(x'' - x') = \frac{1}{2} \cdot b^E \cdot b^{-m+1}$. As $z_0 > 0$, it follows that $x = b^E + |x - b^E| \ge b^E + |x - \mathrm{rd}(x)|$, and hence, as desired,

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| \le \frac{|x - \mathrm{rd}(x)|}{b^E + |x - \mathrm{rd}(x)|} = \frac{1}{1 + \frac{b^E}{|x - \mathrm{rd}(x)|}} \le \frac{1}{1 + \frac{b^E}{\frac{1}{2} \cdot b^E \cdot b^{-m+1}}} = \frac{1}{1 + 2b^{m-1}}\,.$$

□

Example 4.9 The machine epsilon for the data type `double` defined in IEEE 754 is

$$\mathrm{eps}(F_{\texttt{double}}) = \frac{1}{1 + 2^{53}} \approx 1.11 \cdot 10^{-16}\,.$$

Note that `std::numeric_limits<double>::epsilon()` yields the smallest number x with $x > 0$ and $1 + x \in F_{\texttt{double}}$. Hence $x = 2^{-52} \approx 2 \cdot \mathrm{eps}(F_{\texttt{double}})$.

Definition 4.10 Let F be a machine-number range and $s \in \mathbb{N}$. A machine number $f \in F$ has (at least) s **significant digits** in its b-adic floating-point representation if $f \neq 0$ and for every rounding rd and every $x \in \mathbb{R}$ with $\mathrm{rd}(x) = f$ we have:

$$|x - f| \le \tfrac{1}{2} \cdot b^{\lfloor \log_b |f| \rfloor + 1 - s}\,.$$

Example 4.11 Every machine number $f \in F_{\texttt{double}} \setminus \{-\max F_{\texttt{double}}, 0, \max F_{\texttt{double}}\}$ has at least $\lfloor 52 \log_{10} 2 \rfloor = 15$ significant digits in its decimal representation, because the number $x \in \mathbb{R}$ with normalized floating-point representation $x = 2^E \cdot \sum_{i=0}^{\infty} z_i \cdot 2^{-i}$ and $\mathrm{rd}(x) = f$ satisfies

$$|x - f| \le \frac{1}{2} \cdot 2^E \cdot 2^{-52} \le 2^{-53}|f| < \tfrac{1}{2} \cdot 10^{\lfloor \log_{10} |f| \rfloor + 1 - 52 \log_{10} 2}\,,$$

as $2^E \le |f|$.

After inserting the line `#include <iomanip>` one can use `std::cout << std::setprecision(15);` to ensure that all these digits are shown in every subsequent output of the type `double`.

4.4 Machine Arithmetic

Let $\circ$ be one of the four operations $\{+, -, \cdot, /\}$. In Example 4.13 we see that $x \circ y \in F$ does not hold for all $x, y \in F$. The computer therefore performs an ancillary operation $\circledcirc$ with the property that $x \circledcirc y \in F$. We make the following

Assumption 4.12 $x \circledcirc y = \mathrm{rd}(x \circ y)$ for some rounding rd to F and all $x, y \in F$.

This, for example, is stipulated by IEEE 754. It is also stipulated in the same document for the square root function but not, for example, for the functions $\sin(x)$, $\exp(x)$, etc. Assumption 4.12 guarantees that the result of an elementary operation is the exact result of $x \circ y$ rounded to a nearest machine number. Note, however, that in order to obtain $x \circledcirc y$ one does not have to compute $x \circ y$ exactly. One can, in fact, terminate the calculation when sufficiently many digits have been determined.

Example 4.13 Let $F = F(10, 2, -5, 5)$.
Choose $x = 4.5 \cdot 10^1 = 45$ and $y = 1.1 \cdot 10^0 = 1.1$. Then

$$\begin{aligned}
x \oplus y &= \mathrm{rd}(x + y) = \mathrm{rd}(46.1) &&= \mathrm{rd}(4.61 \cdot 10^1) = 4.6 \cdot 10^1 \\
x \ominus y &= \mathrm{rd}(x - y) = \mathrm{rd}(43.9) &&= \mathrm{rd}(4.39 \cdot 10^1) = 4.4 \cdot 10^1 \\
x \odot y &= \mathrm{rd}(x \cdot y) \;\;= \mathrm{rd}(49.5) &&= \mathrm{rd}(4.95 \cdot 10^1) \in \{4.9 \cdot 10^1, \, 5.0 \cdot 10^1\} \\
x \oslash y &= \mathrm{rd}(x / y) \;\;= \mathrm{rd}(40.\overline{90}) &&= \mathrm{rd}(4.\overline{09} \cdot 10^1) = 4.1 \cdot 10^1 \,.
\end{aligned}$$

By Assumption 4.12 the relative error is

$$\left| \frac{x \circ y - x \circledcirc y}{x \circ y} \right| = \left| \frac{x \circ y - \mathrm{rd}(x \circ y)}{x \circ y} \right| \le \mathrm{eps}(F)$$

if $|x \circ y| \in \mathrm{range}(F)$.

The commutative law for addition and multiplication also holds in machine arithmetic. The associative and distributive laws, however, do not hold in general:

Example 4.14 In the machine-number range $F = F(10, 2, -5, 5)$ we let $x = 4.1 \cdot 10^0$ and $y = 8.2 \cdot 10^{-1}$ and $z = 1.4 \cdot 10^{-1}$. Then

$$\begin{aligned}
(x \oplus y) \oplus z &= 4.9 \oplus 0.14 = 5.0 \\
x \oplus (y \oplus z) &= 4.1 \oplus 0.96 = 5.1 \\
x \odot (y \oplus z) &= 4.1 \odot 0.96 = 3.9 \\
(x \odot y) \oplus (x \odot z) &= 3.4 \oplus 0.57 = 4.0
\end{aligned}$$

Clearly, therefore, mathematically equivalent expressions can yield substantially different results when executed in machine arithmetic, even when the input is representable using machine numbers. This can have dramatic consequences.

5 Computing with Errors

When solving numerical computational problems one always has to expect errors. Because there are only finitely many machine numbers, errors can occur not only when inputting data (the number 0.1, for example, has no exact binary representation with finitely many digits; cf. Example 4.2), but also when using the elementary operations $+$, $-$, $\cdot$ and $/$. Moreover, the desired answer may be a nonrepresentable number. One distinguishes between three types of errors:

Data Errors Before one can run a computation, one must enter input data. For numerical computational problems the input data is usually inexact (e.g. measurements) or cannot be represented exactly in the computer. Instead of an exact input value x one has a perturbed input value $\tilde{x}$. Then, even if the computation is exact, one cannot expect an exact result. Furthermore, errors are propagated by elementary operations and can grow in an extreme way, even if the computation is exact; see Sect. 5.2.

Rounding Errors Because the machine-number range is bounded, all intermediate results will be rounded (see Sect. 4.4). This can also falsify the output in an extreme way because these errors are also propagated.

Method Errors Many algorithms, even ones that work exactly, do not output an exact result after a finite number of steps, whatever that number may be. They do, however, get steadily closer to an exact solution, i.e. they compute a (theoretically infinite) sequence of results which converges to the exact solution. This is in fact unavoidable if the solution is an irrational number. The difference between the result obtained upon termination and the exact solution is known as the method error. We will see an example of this in Sect. 5.1.

DOI 10.1007/978-3-319-39558-6_5

Other sources of error arise in practice. There are, in particular, the so-called model errors which arise when a mathematical model is not a correct representation of the problem to be solved, as well as errors arising out of poor implementation (e.g. uncorrected out-of-range values).

5.1 Binary Search

If one has a strictly increasing function $f\colon \mathbb{R} \to \mathbb{R}$ and wants to find $f^{-1}(x)$ for a certain $x \in \mathbb{R}$, then binary search is an option if one has bounds L and U for which $f(L) \leq x \leq f(U)$. In this context it is not necessary to know the function f explicitly, it suffices to be able to evaluate it. Thus one assumes that there is a subroutine for computing the value $f(x)$ for a given x, but about which one knows nothing else. Here one also says that f is given by an **oracle**.

The basic idea of binary search is to halve a given interval and to decide in which half to continue the search.

Example 5.1 We can, for example, compute square roots by means of binary search, using only multiplication (for evaluating the function $f(x) = x^2$). The computation of $\sqrt{3}$, for example, with the initial bounds $L = 1$ and $U = 2$, runs as follows:

$$\begin{aligned}
& && \sqrt{3} \in [1, 2] \\
1.5^2 &= 2.25 &\Rightarrow\ & \sqrt{3} \in [1.5, 2] \\
1.75^2 &= 3.0625 &\Rightarrow\ & \sqrt{3} \in [1.5, 1.75] \\
1.625^2 &= 2.640625 &\Rightarrow\ & \sqrt{3} \in [1.625, 1.75] \\
1.6875^2 &= 2.84765625 &\Rightarrow\ & \sqrt{3} \in [1.6875, 1.75] \\
1.71875^2 &= 2.9541015625 &\Rightarrow\ & \sqrt{3} \in [1.71875, 1.75] \\
1.734375^2 &= 3.008056640625 &\Rightarrow\ & \sqrt{3} \in [1.71875, 1.734375]
\end{aligned}$$

The sequence of midpoints of the intervals clearly converges to the correct value (albeit rather slowly—after six iterations we know only the first digit after the decimal point—more on this later). We cannot, of course, ever compute the exact value of irrational numbers like $\sqrt{3}$.

Let $[l_i, u_i]$ denote the interval in the i-th iteration and $m_i = \frac{l_i+u_i}{2}$ its midpoint (in the above example $m_1 = 1.5$; $m_2 = 1.75$ etc.). Then we know *a priori* that $|m_i - \sqrt{3}| \leq \frac{1}{2}|u_i - l_i| = 2^{-i}|u_1 - l_1| = 2^{-i}|U - L|$ or, for example, that $|m_6 - \sqrt{3}| \leq 2^{-6} = \frac{1}{64} = 0.015625$. This is called an **a priori bound**.

After the computation one can often obtain a better bound, a so-called **a posteriori bound**. In the above example $|m_i - \sqrt{3}| = \frac{|m_i^2-3|}{m_i+\sqrt{3}} \leq \frac{|m_i^2-3|}{m_i+l_i}$, and for $i = 6$ we get $|m_6 - \sqrt{3}| \leq \frac{0.008056640625}{3.453125} \approx 0.002333$.

The discrete version of binary search is at least as useful as the real number version:

Algorithm 5.2 (**Binary Search (Discrete)**)

Input: an oracle for evaluating an increasing function $f: \mathbb{Z} \to \mathbb{R}$. Integers $L, U \in \mathbb{Z}$ with $L \leq U$, and a number $y \in \mathbb{R}$ with $y \geq f(L)$.
Output: the maximum $n \in \{L, \dots, U\}$ with $f(n) \leq y$.

```
$l \leftarrow L$
$u \leftarrow U + 1$
**while** $u > l + 1$ **do**
    $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
    **if** $f(m) > y$ **then** $u \leftarrow m$ **else** $l \leftarrow m$
**output** $l$
```

Theorem 5.3 *Algorithm 5.2 works correctly and terminates after $O(\log(U-L+2))$ iterations.*

Proof The correctness follows from the fact that at all times $L \leq l \leq u - 1 \leq U$ holds, as well as $f(l) \leq y$ and ($u > U$ or $f(u) > y$), i.e. the correct result is always an element of the set $\{l, \dots, u-1\}$.

The running time follows immediately by observing that $u - l - 1$ is reduced in each iteration to at most

$$\max\left\{\left\lfloor \frac{l+u}{2} \right\rfloor - l - 1, u - \left\lfloor \frac{l+u}{2} \right\rfloor - 1\right\} \leq \max\left\{\frac{l+u}{2} - l - 1, u - \frac{l+u-1}{2} - 1\right\} \leq \frac{u-l-1}{2},$$

i.e. is at least halved. □

Binary search is particularly useful for locating data in a sorted data set. The C++ Standard Library provides functions for binary search after inserting the line `#include <algorithm>`. One of these is the function `binary_search` with which one can, for example, quickly test in a sorted `vector` whether it contains a certain element.

5.2 Error Propagation

If at a certain point in time the stored numbers are error-prone (due to data errors and/or rounding errors, see above), then these errors will be propagated by operations performed on the relevant numbers. Here one speaks of error propagation. The following lemma describes how errors are propagated under the *exact* execution of elementary operations.

Lemma 5.4 *For given $x, y \in \mathbb{R} \setminus \{0\}$, let $\tilde{x}, \tilde{y} \in \mathbb{R}$ be approximate values with $\varepsilon_x := \frac{x-\tilde{x}}{x}$ and $\varepsilon_y := \frac{y-\tilde{y}}{y}$ (i.e. $|\varepsilon_x|$ and $|\varepsilon_y|$ are the corresponding relative errors). Setting $\varepsilon_\circ := \frac{x \circ y - (\tilde{x} \circ \tilde{y})}{x \circ y}$ with $\circ \in \{+, -, \cdot, /\}$, we then have:*

$$\begin{aligned}
\varepsilon_+ &= \varepsilon_x \cdot \frac{x}{x+y} + \varepsilon_y \cdot \frac{y}{x+y} \,, \\
\varepsilon_- &= \varepsilon_x \cdot \frac{x}{x-y} - \varepsilon_y \cdot \frac{y}{x-y} \,, \\
\varepsilon_\cdot &= \varepsilon_x + \varepsilon_y - \varepsilon_x \cdot \varepsilon_y \,, \\
\varepsilon_/ &= \frac{\varepsilon_x - \varepsilon_y}{1 - \varepsilon_y} \,.
\end{aligned}$$

Proof Clearly $\tilde{x} = x \cdot (1 - \varepsilon_x)$ and $\tilde{y} = y \cdot (1 - \varepsilon_y)$. Then:

$$\varepsilon_+ = \frac{x + y - (\tilde{x} + \tilde{y})}{x + y} = \frac{x + y - (1 - \varepsilon_x) \cdot x - (1 - \varepsilon_y) \cdot y}{x + y} = \frac{\varepsilon_x \cdot x}{x + y} + \frac{\varepsilon_y \cdot y}{x + y} \,,$$

$$\varepsilon_- \qquad \text{analogously} \,,$$

$$\begin{aligned}
\varepsilon_\cdot &= \frac{x \cdot y - \tilde{x} \cdot \tilde{y}}{x \cdot y} = \frac{x \cdot y - (1 - \varepsilon_x) \cdot x \cdot (1 - \varepsilon_y) \cdot y}{x \cdot y} \\
&= 1 - (1 - \varepsilon_x) \cdot (1 - \varepsilon_y) = \varepsilon_x + \varepsilon_y - \varepsilon_x \cdot \varepsilon_y \,,
\end{aligned}$$

$$\begin{aligned}
\varepsilon_/ &= \frac{x/y - \tilde{x}/\tilde{y}}{x/y} = \frac{x/y - ((1 - \varepsilon_x) \cdot x)/((1 - \varepsilon_y) \cdot y)}{x/y} \\
&= 1 - \frac{1 - \varepsilon_x}{1 - \varepsilon_y} = \frac{1 - \varepsilon_y - 1 + \varepsilon_x}{1 - \varepsilon_y} = \frac{\varepsilon_x - \varepsilon_y}{1 - \varepsilon_y} \,.
\end{aligned}$$

□

With the operations $\cdot$ and $/$ the relative errors essentially add and subtract respectively, at least when they are small. With the operations $+$ and $-$ the relative error can, however, be greatly increased if $|x \pm y|$ is very much smaller than both $|x|$ and $|y|$. This effect is called cancellation and should be avoided wherever possible.

With the operations $+$ and $-$ the *absolute* errors add and subtract respectively, but they can be greatly increased by the operations $\cdot$ and $/$. The combination of multiplication/division and addition/subtraction is therefore particularly error-prone.

Example 5.5 Consider the following system of linear equations:

$$\begin{aligned} 10^{-20}x + \quad 2y &= 1 \\ 10^{-20}x + 10^{-20}y &= 10^{-20}\,. \end{aligned}$$

Subtracting the second equation from the first yields

$$(2 - 10^{-20})y = 1 - 10^{-20}\,.$$

Rounding to the next representable numbers (in $F_{\texttt{double}}$) yields $2y = 1$, so $y = 0.5$. This is still approximately correct. But inserting this in the first equation yields $x = 0$ which is completely wrong (inserting in the second equation would have been better here). The correct solution is $x = \frac{1}{2-10^{-10}} = 0.5000\ldots$ and $y = \frac{1-10^{-20}}{2-10^{-20}} = 0.4999\ldots$

In Chap. 11 we will deal in more detail with the solution of systems of linear equations.

5.3 The Condition of a Numerical Computational Problem

We saw earlier that certain elementary operations can cause small (relative) errors in the input to result in large (relative) errors in the output. This is made precise in the following definition:

Definition 5.6 Let $P \subseteq D \times E$ be a numerical computational problem with $D, E \subseteq \mathbb{R}$. Let $d \in D$ be an instance of P with $d \neq 0$. Then the **(relative) condition number** of d, often denoted $\kappa(d)$, is defined to be

$$\lim_{\varepsilon \to 0} \sup \left\{ \inf \left\{ \frac{|\frac{e-e'}{e}|}{|\frac{d-d'}{d}|} \;:\, e \in E,\ (d,e) \in P \right\} \;:\, d' \in D,\ e' \in E,\ (d',e') \in P,\ 0 < |d-d'| < \varepsilon \right\}$$

(where we set $\frac{0}{0} := 0$).

If $\kappa(d)$ denotes the condition number of an instance $d \in D$, then the problem P itself is said to have the **(relative) condition number**

$$\sup\{\kappa(d) : d \in D,\ d \neq 0\}\ .$$

The condition number gives an indication of the unavoidable error in the output induced by an error in the input; or, more precisely, the maximum ratio of these errors, assuming that the input error is small.

The above definition can be generalized to multidimensional problems in several different ways; we will discuss this topic in Sect. 11.6.

The definition of the relative condition number given above refers to the ratio of the *relative* errors. One can define the absolute condition number analogously via the ratio of the absolute errors. This is, however, not often used.

Proposition 5.7 *Let $f: D \to E$ be an injective numerical computational problem with $D, E \subseteq \mathbb{R}$. If $d \in D$ is an instance with $d \neq 0$ and $f(d) \neq 0$, then its condition number is given by*

$$\frac{|d|}{|f(d)|} \cdot \lim_{\varepsilon \to 0} \sup \left\{ \frac{|f(d') - f(d)|}{|d' - d|} : d' \in D,\ 0 < |d - d'| < \varepsilon \right\} .$$

Corollary 5.8 *Let $f: D \to E$ be an injective numerical computational problem, where $D, E \subseteq \mathbb{R}$ and f is differentiable. Then the condition number of an instance $d \in D$ with $d \neq 0$ and $f(d) \neq 0$ is given by*

$$\kappa(d) = \frac{|f'(d)| \cdot |d|}{|f(d)|} .$$

This elegant formula can be readily generalized to multidimensional differentiable functions. We will not, however, pursue this direction here.

The smaller the value of $\kappa(d)$, the better. In particular, a condition number of at most 1 is good because it indicates that small errors will not grow while solving the problem. One speaks of a well-conditioned problem.

Example 5.9

- Let $f(x) = \sqrt{x}$. Then by Corollary 5.8 the condition number is $\frac{\frac{1}{2\sqrt{x}} \cdot x}{\sqrt{x}} = \frac{1}{2}$. Thus the square root function is well-conditioned.
- Let $f(x) = x + a$ (i.e. we only consider error propagation in the first summand x). Then the condition number is $|\frac{x}{x+a}|$. Thus addition is ill-conditioned if $|x + a|$ is very much smaller than $|x|$ (cancellation).
- Let $f(x) = a \cdot x$. Then the condition number is $|\frac{a \cdot x}{a \cdot x}| = 1$. Thus multiplication is always well-conditioned.

5.4 Error Analysis

An algorithm is said to be numerically stable (for a certain instance or generally) if each of its computational steps is well-conditioned. A numerically stable algorithm for a given problem can only exist if the problem itself is well-conditioned. As with the concept *well-conditioned*, the concept *numerically stable* is not defined precisely.

In error analysis one distinguishes between:

- **forward error analysis**, in which one investigates how the relative error accumulates during the computation, and
- **backward error analysis**, in which one interprets every intermediate result as the exact result of perturbed data and then estimates how great the input error would need to be for the given output to be correct.

Example 5.10 Consider the addition of two numbers in the machine-number range F:

- Forward error analysis: $x \oplus y = \mathrm{rd}(x+y) = (x+y)\cdot(1+\varepsilon)$ with $|\varepsilon| \leq \mathrm{eps}(F)$.
- Backward error analysis: $x \oplus y = x\cdot(1+\varepsilon)+y\cdot(1+\varepsilon) = \tilde{x}+\tilde{y}$ with $|\varepsilon| \leq \mathrm{eps}(F)$ and $\tilde{x} = x\cdot(1+\varepsilon)$ and $\tilde{y} = y\cdot(1+\varepsilon)$.

A combination of backward error analysis and condition can sometimes also yield an estimate of the error in the result. In Example 5.1, for example, 1.734375 is the correct solution for the input $1.734375^2 = 3.008056640625 = 3(1+\varepsilon)$ with $\varepsilon \leq 0.00269$. Because the condition number is $\frac{1}{2}$, it follows that the relative error $\left|\frac{\sqrt{3}-1.734375}{\sqrt{3}}\right|$ can be roughly bounded by $\frac{1}{2}\cdot 1.00269$. This, however, does not always hold exactly as the condition number only comprises the limit for errors close to 0.

An alternative approach to error analysis is via interval arithmetic. Instead of computing with numbers x that are not known exactly, one always computes with intervals $[a, b]$ containing x. Even in the input data one can replace x by the interval $\{y \in \mathbb{R} : \mathrm{rd}(y) = \mathrm{rd}(x)\}$. The addition of two intervals then yields $[a_1, b_1]\oplus[a_2, b_2] = [\mathrm{rd}^-(a_1+a_2), \mathrm{rd}^+(b_1+b_2)]$, where rd^- and rd^+ denote rounding down and up to the next machine number, respectively. This yields error bounds at every point in time; this benefit, however, necessitates more computational effort.

When using interval arithmetic in C++, one should define a class enabling one to store such intervals and providing the necessary elementary operations. Instead of rd^- and rd^+ one can use faster and possibly stronger rounding functions (clearly with less stringent error bounds), e.g. division or multiplication by $1 + b^{-m}$ in $F(b, m, E_{\min}, E_{\max})$, respectively.

5.5 Newton's Method

Here we will discuss a further general approximation method which is often superior to binary search. Note, however, that this method only works with differentiable functions whose derivative is computable and nonzero in a neighborhood of the desired value and, furthermore, it only converges under certain conditions.

The problem is to find a zero of a differentiable function $f\colon \mathbb{R} \to \mathbb{R}$. Beginning with a guessed starting value x_0, we set

$$x_{n+1} \leftarrow x_n - \frac{f(x_n)}{f'(x_n)} \qquad \text{for } n = 0, 1, \ldots$$

Clearly one must assume that $f'(x_n) \neq 0$. This method is due to Isaac Newton and is known as Newton's Method. Under certain conditions it converges very rapidly.

Here is an example: in order to find the (positive) square root of a real number $a \geq 1$, it suffices to compute the positive zero of the function $f\colon \mathbb{R} \to \mathbb{R}$ with $f(x) = x^2 - a$. Here Newton's Method is given as follows:

$$x_{n+1} \leftarrow x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \qquad \text{for } n = 0, 1, \ldots$$

and one can start with the value $x_0 = 1$, for example. This method for finding square roots was described by Heron of Alexandria nearly 2000 years ago, but was already known a further 2000 years earlier in Mesopotamia and is thus also known as the Babylonian method.

Example 5.11 For $a = 3$ and the starting value $x_0 = 1$ we get:

$$\begin{aligned} x_1 &= 2 &&= 2 \\ x_2 &= \tfrac{7}{4} &&= 1.75 \\ x_3 &= \tfrac{97}{56} &&= 1.732142857\ldots \\ x_4 &= \tfrac{18817}{10864} &&= 1.732050810\ldots \end{aligned}$$

This sequence seems to converge much more rapidly to $\sqrt{3} = 1.73205080756\,8877\ldots$ than the one obtained by binary search in Example 5.1, and that is indeed the case. We will now make this more precise.

Definition 5.12 Let $(x_n)_{n\in\mathbb{N}}$ be a convergent sequence of real numbers and $x^* := \lim\limits_{n\to\infty} x_n$.

If there exists a constant $c < 1$ such that

$$|x_{n+1} - x^*| \;\leq\; c \cdot |x_n - x^*|$$

for all $n \in \mathbb{N}$, then the convergence of the sequence is said to be of **order** (at least) 1.

Let $p > 1$. If there exists a constant $c \in \mathbb{R}$ such that

$$|x_{n+1} - x^*| \;\leq\; c \cdot |x_n - x^*|^p$$

for all $n \in \mathbb{N}$, then the convergence of the sequence is said to be of **order** (at least) p.

We say that the convergence of a sequence (and of an algorithm that computes such a sequence) is linear or quadratic (and that the sequence converges linearly or quadratically) if its order of convergence is 1 or 2, respectively.

Example 5.13 The sequence of interval lengths arising in binary search (cf. Example 5.1) converges linearly (here one can choose $c = \frac{1}{2}$).

We will now show that the Babylonian method for computing square roots converges quadratically: the error is squared with every step.

Theorem 5.14 *Let $a \geq 1$ and $x_0 \geq 1$. The sequence $(x_n)_{n\in\mathbb{N}}$ defined by $x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ for $n = 0, 1, \ldots$ has the following three properties:*

(a) $x_1 \geq x_2 \geq \ldots \geq \sqrt{a}$.

(b) *The sequence converges quadratically to the limit $\lim_{n\to\infty} x_n = \sqrt{a}$; more precisely, we have for all $n \geq 0$:*

$$x_{n+1} - \sqrt{a} \;\leq\; \frac{1}{2}(x_n - \sqrt{a})^2. \tag{5.1}$$

(c) *For all $n \geq 1$ the following holds:*

$$x_n - \sqrt{a} \;\leq\; x_n - \frac{a}{x_n} \;\leq\; 2(x_n - \sqrt{a})\,.$$

Proof

(a) By the AM-GM inequality (i.e. arithmetic mean $\frac{x+y}{2} \geq$ geometric mean $\sqrt{xy}$ for all $x, y \geq 0$, which follows immediately from $x+y-2\sqrt{xy} = (\sqrt{x}-\sqrt{y})^2 \geq 0$) we obtain for all $n \geq 0$:

$$\sqrt{a} = \sqrt{x_n \cdot \frac{a}{x_n}} \;\leq\; \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) = x_{n+1}\,. \tag{5.2}$$

Furthermore, we have for $n \geq 1$:

$$x_{n+1} - x_n = \frac{1}{2}\cdot\left(x_n + \frac{a}{x_n}\right) - x_n = \frac{1}{2x_n}\left(a - x_n^2\right) \overset{(5.2)}{\leq} 0\,.$$

(b) For all $n \geq 0$ we have:

$$x_{n+1} - \sqrt{a} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) - \sqrt{a} = \frac{1}{2x_n}\left(x_n^2 + a - 2\sqrt{a}x_n\right) = \frac{1}{2x_n}\left(x_n - \sqrt{a}\right)^2 .$$

As $\frac{1}{x_n}\left(x_n - \sqrt{a}\right) \leq 1$, we obtain $x_{n+1} - \sqrt{a} \leq \frac{1}{2}\left(x_n - \sqrt{a}\right)$, proving that $\lim_{n\to\infty} x_n = \sqrt{a}$. As $x_n \geq 1$, this proves (5.1).

(c) For $n \in \mathbb{N}$ we have:

$$x_n - \sqrt{a} \overset{(a)}{\leq} x_n - \frac{a}{x_n} = 2x_n - 2x_{n+1} \overset{(a)}{\leq} 2(x_n - \sqrt{a}) .$$

□

Theorem 5.14(c) suggests a termination criterion: for a desired absolute error bound $\varepsilon > 0$ one terminates when $x_n - \frac{a}{x_n} \leq \varepsilon$. This implies that $x_n - \sqrt{a} \leq \varepsilon$. As long as the termination criterion does not hold, $x_n - \sqrt{a} > \frac{\varepsilon}{2}$, i.e. in the worst case the termination criterion is too strong by a factor of 2.

Note that by (5.1) the convergence is especially rapid as soon as $|x_n - \sqrt{a}| \leq 1$. One can easily achieve this directly by setting $b := a \cdot 2^{-2\lfloor \log_4 a \rfloor}$ and then computing $\sqrt{b}$ with the above method, using $\sqrt{a} = \sqrt{b} \cdot 2^{\lfloor \log_4 a \rfloor}$ and $\sqrt{b} \in [1,2)$.

Moreover, Newton's Method is also suitable for the fast division of large numbers. In order to compute $\frac{a}{b}$, one can compute the zero of the function $f: x \mapsto \frac{1}{x} - b$ with Newton's Method sufficiently accurately and then multiply the result by a. An iteration is given as follows:

$$x_{n+1} \leftarrow x_n - \frac{\frac{1}{x_n} - b}{-\frac{1}{x_n^2}} = x_n + x_n(1 - bx_n).$$

Note that no division has been used here. As Newton's Method also converges quadratically here, $O(\log \log n)$ iterations suffice for computing n significant digits. For the required multiplications one uses a fast multiplication method (cf. Sect. 3.2).

6 Graphs

Numerous discrete structures can best be described using graphs. Furthermore, in countless applications graphs appear naturally. Thus graphs can well be considered to be the most important structure in discrete mathematics.

6.1 Basic Definitions

Definition 6.1 An **undirected graph** is a triple (V, E, ψ), where V and E are finite sets, $V \neq \emptyset$, and $\psi: E \to \{X \subseteq V : |X| = 2\}$. A **directed graph** (also called **digraph**) is a triple (V, E, ψ), where V and E are finite sets, $V \neq \emptyset$, and $\psi: E \to \{(v, w) \in V \times V : v \neq w\}$. A **graph** is a directed or an undirected graph. The elements of V are called **vertices** and the elements of E **edges**.

Example 6.2 Let $(\{1, 2, 3, 4, 5\}, \{a, b, c, d, e\}, \{a \mapsto (2,5), b \mapsto (4,5), c \mapsto (1,2), d \mapsto (2,1), e \mapsto (1,3)\})$ specify a directed graph with five vertices and five edges. Figure 6.1a depicts this graph as a drawing in $\mathbb{R}^2$. The vertices are represented as points and the edges as straight lines (or curves) joining vertices. If the graph is directed, then an arrowhead on an edge indicates the direction from the first to the second vertex. Figure 6.1b depicts an undirected graph with seven vertices and six edges.

Two edges $e \neq e'$ with $\psi(e) = \psi(e')$ are called **parallel**. A graph with no parallel edges is said to be **simple**. In a simple graph one identifies e with $\psi(e)$, i.e. one writes $G = (V(G), E(G))$ with $E(G) \subseteq \{\{v, w\} : v, w \in V(G),\ v \neq w\}$, or $E(G) \subseteq \{(v, w) : v, w \in V(G),\ v \neq w\}$. One often also uses this simplified notation for non-simple graphs; then $E(G)$ is a multiset. For example, the statement "$e = (x, y) \in E(G)$" is short for "$e \in E(G)$ with $\psi(e) = (x, y)$". Figure 6.1 depicts a simple digraph and a non-simple undirected graph.

DOI 10.1007/978-3-319-39558-6_6

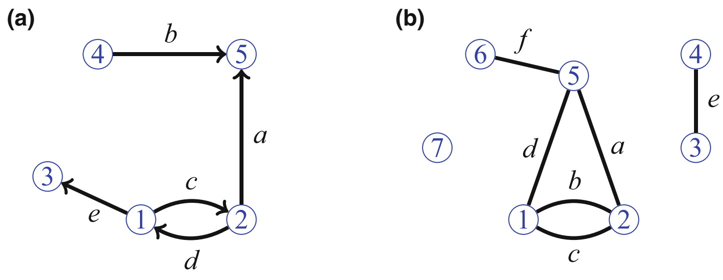


Fig. 6.1 A directed and an undirected graph

For a given edge $e = \{x, y\}$ or $e = (x, y)$ one says that e **joins** the two vertices x and y, and these vertices are then called **adjacent**. The vertices x and y are the **endpoints** of e; x is a **neighbor** of y and, similarly, y is a neighbor of x. The vertices x and y are said to be **incident** with the edge e. For an edge $e = (x, y)$ we say: e **begins in** x and **ends in** y, or: e **goes from** x **to** y.

The following notation is very useful:

Definition 6.3 Let G be an undirected graph and $X \subseteq V(G)$. Define

$$\delta(X) \ := \ \{\{x, y\} \in E(G) : x \in X,\ y \in V(G) \setminus X\}\,.$$

The **neighborhood** of X is the set $N(X) := \{v \in V(G) \setminus X : \delta(X) \cap \delta(\{v\}) \neq \emptyset\}$.

Let G be a digraph and $X \subseteq V(G)$. Define

$$\delta^+(X) \ := \ \{(x, y) \in E(G) : x \in X,\ y \in V(G) \setminus X\}\,,$$

$\delta^-(X) := \delta^+(V(G) \setminus X)$ and $\delta(X) := \delta^+(X) \cup \delta^-(X)$.

For a graph G and $x \in V(G)$, let $\delta(x) := \delta(\{x\})$, $N(x) := N(\{x\})$, $\delta^+(x) := \delta^+(\{x\})$ and $\delta^-(x) := \delta^-(\{x\})$. The **degree** of a vertex x is $|\delta(x)|$, i.e. the number of edges with which x is incident. If G is directed, then $|\delta^-(x)|$ and $|\delta^+(x)|$ are the **in-degree** and the **out-degree** of x respectively, and $|\delta(x)| = |\delta^+(x)| + |\delta^-(x)|$ holds.

When one is considering several different graphs with the same vertex set, it is often necessary to differentiate between them. This is achieved by using a subscript, for example $\delta_G(x)$.

Theorem 6.4 *The following holds for every graph G:* $\sum_{x \in V(G)} |\delta(x)| = 2|E(G)|$.

Proof For every edge there are exactly two vertices that are incident with it. Thus on the left-hand side of the equation every edge is counted exactly twice. □

Theorem 6.5 *The following holds for every digraph G:* $\sum_{x \in V(G)} |\delta^+(x)| = \sum_{x \in V(G)} |\delta^-(x)|$.

Proof On both sides of the equation every edge is counted exactly once. □

An immediate consequence of Theorem 6.4 is:

Corollary 6.6 *For every graph, the number of vertices with odd degree is even.* □

Definition 6.7 A graph H is a **subgraph** of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. (This shall of course also mean that $\psi(e)$ is identical in G and H for all $e \in E(H)$). One also says that G **contains** the graph H. If $V(G) = V(H)$, then H is called a **spanning subgraph**.

H is an **induced subgraph** if $E(H) = \{\{x, y\} \in E(G) \mid x, y \in V(H)\}$ or $E(H) = \{(x, y) \in E(G) \mid x, y \in V(H)\}$ respectively. An induced subgraph H of G is fully determined by its vertex set $V(H)$ alone. Thus one also says that H is the subgraph of G **induced by** $V(H)$ and denotes it with $G[V(H)]$.

Two ways of forming subgraphs are very common: let G be a graph. Then for $v \in V(G)$ we define $G - v := G[V(G) \setminus \{v\}]$. Analogously, for $e \in E(G)$ we define $G - e := (V(G), E(G) \setminus \{e\})$.

6.2 Paths and Circuits

Definition 6.8 An **edge progression** (from x_1 to x_{k+1}) in a graph G is a sequence $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_{k+1}$ with $k \in \mathbb{N} \cup \{0\}$ and $e_i = (x_i, x_{i+1}) \in E(G)$ or $e_i = \{x_i, x_{i+1}\} \in E(G)$ for $i = 1, \ldots, k$. If $x_1 = x_{k+1}$, then the edge progression is called **closed**.

A **path** is a graph $P = (\{x_1, \ldots, x_{k+1}\}, \{e_1, \ldots, e_k\})$, where $k \geq 0$, with the property that $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_{k+1}$ is an edge progression (with $k+1$ distinct vertices). One also says that P is an x_1-x_{k+1}-path or that P **joins** x_1 and x_{k+1}. The vertices x_1 and x_{k+1} are called the **endpoints** of the path P; the vertices in $V(P) \setminus \{x_1, x_{k+1}\}$ are the **inner vertices** of P.

A **circuit** is a graph $C = (\{x_1, \ldots, x_k\}, \{e_1, \ldots, e_k\})$, where $k \geq 2$, with the property that the sequence $x_1, e_1, x_2, e_2, \ldots, x_k, e_k, x_1$ is a closed edge progression (with k distinct vertices).

The **length** of a path or a circuit is the number of its edges. If a path or a circuit is a subgraph of G, then one speaks of a path or circuit **in** G.

In a graph G a vertex y is said to be **reachable** from a vertex x if there exists an x-y-path in G. In undirected graphs this property defines an equivalence relation, as the following lemma ensures transitivity:

Lemma 6.9 *Let G be a graph and $x, y \in V(G)$. Then there exists an x-y-path in G if and only if there exists an edge progression from x to y in G.*

Proof "$\Rightarrow$" : By definition, an x-y-path is an edge progression from x to y.

"$\Leftarrow$" : Assume that in a given edge progression from x to y the vertex v appears more than once. Then delete all vertices and edges lying between the first and the last appearance of v. Iterating this step finally results in an x-y-path. $\square$

Two graphs are called **edge-disjoint** or **vertex-disjoint**, if they have no common edges or vertices respectively.

Lemma 6.10 *Let G be an undirected graph with $|\delta(v)|$ an even number for all $v \in V(G)$, or a directed graph with $|\delta^-(v)| = |\delta^+(v)|$ for all $v \in V(G)$. Then for some $k \geq 0$ there exist pairwise edge-disjoint circuits $C_1, \dots, C_k$ with $E(G) = E(C_1) \cup \dots \cup E(C_k)$.*

Proof The proof is by induction over $|E(G)|$. The statement is trivial for $E(G) = \emptyset$. If $E(G) \neq \emptyset$, it suffices to find a circuit because deleting all of its edges leaves a graph that still satisfies the conditions of the lemma. Thus let $e = \{x, y\}$ or $e = (x, y)$ be an edge. Then the assumption ensures that there is another edge e' with which y is incident or which begins in y. We then continue with the other endpoint of e'. In this way we can construct an edge progression all of whose edges are distinct and will after at most $|V(G)|$ steps reach a vertex which we have met before. This then yields a circuit as desired. $\square$

The **symmetric difference** of two sets A and B is defined to be $A \triangle B := (A \setminus B) \cup (B \setminus A)$. Furthermore, $A \dot{\cup} B$ denotes the **disjoint union**, which comprises one copy of each of the elements of $A \triangle B$ and two copies of each of the elements of $A \cap B$.

Lemma 6.11 *Let G be a graph, P an s-t-path in G and Q a t-s-path in G with $P \neq Q$. Then $(V(P) \cup V(Q), E(P) \cup E(Q))$ contains a circuit.*

Proof We consider two cases: $C := E(P) \triangle E(Q)$ if G is undirected and $C := E(P) \dot{\cup} E(Q)$ if G is directed. Let $H := (V(G), C)$. In the first case every vertex of H has even degree. In the second, $|\delta_H^-(v)| = |\delta_H^+(v)|$ for every vertex $v \in V(G)$. Then we have by Lemma 6.10 that in both cases $E(H) = C$ is the disjoint union of edge sets of circuits. None of these circuits can contain an edge of G twice. As $P \neq Q$, we have $\emptyset \neq C \subseteq E(P) \dot{\cup} E(Q)$, which proves the lemma. $\square$

6.3 Connectivity and Trees

We will now study connected graphs and, in particular, minimally connected graphs with a certain vertex set.

Let $\mathcal{F}$ be a family of sets or of graphs. Then $F \in \mathcal{F}$ is minimal if no proper subset or proper subgraph of F is in $\mathcal{F}$. Analogously, $F \in \mathcal{F}$ is maximal if F is not a proper subset or a proper subgraph of an element in $\mathcal{F}$. Note that a minimal or a maximal set in $\mathcal{F}$ is not necessarily a minimum or a maximum set respectively. The latter terms refer to the cardinality of a set.

Definition 6.12 Let G be an undirected graph. G is called **connected** if for any two vertices $x, y \in V(G)$ there exists an x-y-path in G. Otherwise G is called **disconnected**. The maximal connected subgraphs of G are the **connected components** of G. A vertex v is called an **articulation vertex** if v is not the only vertex and $G - v$ has more connected components than G. An edge e is called a **bridge** if $G - e$ has more connected components than G.

For example, the undirected graph in Fig. 6.1b has three connected components, two bridges (e and f) and one articulation vertex (5). The connected components are the subgraphs induced by the equivalence classes of the reachability relation.

Theorem 6.13 *An undirected graph G is connected if and only if $\delta(X) \neq \emptyset$ for all $\emptyset \subset X \subset V(G)$.*

Proof "$\Rightarrow$" : Let $\emptyset \subset X \subset V(G)$ and $x \in X$, $y \in V(G) \setminus X$. As G is connected, there is an x-y-path in G. As $x \in X$ but $y \notin X$, this path contains an edge $\{a, b\}$ with $a \in X$ and $b \notin X$. Hence $\delta(X) \neq \emptyset$.

"$\Leftarrow$" : Suppose G is disconnected. Let a and b be vertices in $V(G)$ for which there is no a-b-path. Let X be the set of vertices reachable from a. As $a \in X$ and $b \notin X$, it follows that $\emptyset \subset X \subset V(G)$. Moreover, by the definition of X we have $\delta(X) = \emptyset$. This, however, contradicts the assumption made for X. □

Remark 6.14 Theorem 6.13 yields a method for checking whether a given graph G is connected or not: compute $\delta(X)$ for every set X with $\emptyset \subset X \subset V(G)$. However, as there are $2^{|V(G)|} - 2$ such sets X, this requires exponential running time. We will shortly present a far better algorithm.

Definition 6.15 An undirected graph is called a **forest** if it contains no circuits. A **tree** is a connected forest. A vertex of degree 1 in a tree is called a **leaf**.

Thus the connected components of a forest are trees.

Theorem 6.16 *Every tree with at least two vertices has at least two leaves.*

Proof Consider a maximal path in a tree. Clearly it has length ≥ 1 and its endpoints are leaves because there are no circuits. □

Lemma 6.17 *Let G be a forest with n vertices, m edges and p connected components. Then $n = m + p$.*

Proof The proof follows by induction over m. The lemma is trivial for $m = 0$. If $m \geq 1$, there exists a connected component of G containing at least two vertices and thus, by Theorem 6.16, also a leaf v. Deleting the edge with which v is incident increases the number of connected components by one. By the induction hypothesis $n = (m - 1) + (p + 1) = m + p$. □

Theorem 6.18 *Let G be an undirected graph with n vertices. Then the following seven statements are equivalent:*

(a) *G is a tree.*
(b) *G has $n - 1$ edges and contains no circuits.*
(c) *G has $n - 1$ edges and is connected.*
(d) *G is a minimal connected graph with vertex set $V(G)$ (i.e. G is connected and every edge of G is a bridge).*
(e) *G is a minimal graph with vertex set $V(G)$ and $\delta(X) \neq \emptyset$ for all $\emptyset \subset X \subset V(G)$.*
(f) *G is a maximal forest with vertex set $V(G)$ (i.e. adding any edge creates a circuit).*
(g) *For any two vertices of G there is exactly one path in G joining them.*

Proof

(a)⇒(g) In a forest any two vertices are, by Lemma 6.11, joined by at most one path, and in a connected graph any two vertices are joined by at least one path.
(g)⇒(d) By definition G is connected. Assume that $G - e$ is connected for some edge e. Then the endpoints of e in G are joined by two different paths, contradicting (g).
(e)⇔(d) This follows immediately from Theorem 6.13.
(d)⇒(f) If every edge is a bridge, then G contains no circuits. If G is connected, then adding any edge will create a circuit.
(f)⇒(b) The maximality of G implies that G is connected and hence, by Lemma 6.17, $|E(G)| = |V(G)| - 1$.
(b)⇒(c) This follows immediately from Lemma 6.17.
(c)⇒(a) If a connected graph contains a circuit, then we can delete an edge without the graph becoming disconnected. Repeating this step, we will, after deleting k edges, end up with a connected forest with $n - 1 - k$ edges. Then by Lemma 6.17 we have $k = 0$. □

Corollary 6.19 *An undirected graph is connected if and only if it contains a spanning tree.*

Proof This follows from (d)⇔(a) of Theorem 6.18. □

6.4 Strong Connectivity and Arborescences

Definition 6.20 For a given digraph G we will occasionally consider the **underlying undirected graph**, i.e. an undirected graph G' with $V(G) = V(G')$ and the property that there is a bijection $\phi: E(G) \to E(G')$ with $\phi((v, w)) = \{v, w\}$ for all $(v, w) \in E(G)$. One also calls G an **orientation** of G'.

A digraph G is called **(weakly) connected** if the underlying undirected graph is connected. G is called **strongly connected** if for all $s, t \in V(G)$ there is a path from s to t and a path from t to s. The **strongly connected components** of a digraph are the maximal strongly connected subgraphs.

The digraph depicted in Fig. 6.1a is (weakly) connected and has four strongly connected components.

We will now fix a certain vertex r and check whether all vertices are reachable from r.

Theorem 6.21 *Let G be a digraph and $r \in V(G)$. Then there is an r-v-path for every $v \in V(G)$ if and only if $\delta^+(X) \neq \emptyset$ for all $X \subset V(G)$ with $r \in X$.*

Proof The proof is analogous to that of Theorem 6.13. □

Definition 6.22 A digraph is called a **branching** if the underlying undirected graph is a forest and every vertex has at most one edge ending in it. A connected branching is called an **arborescence**. By Theorem 6.18 an arborescence with n vertices has $n - 1$ edges and thus has exactly one vertex r with $\delta^-(r) = \emptyset$. This vertex is called the **root** of the arborescence. For any edge (v, w) of a branching, w is called a **child** of v and v the **predecessor** of w. Vertices with no children are called **leaves**.

Theorem 6.23 *Let G be a digraph with n vertices and $r \in V(G)$. Then the following seven statements are equivalent:*

(a) *G is an arborescence with root r (i.e. a connected branching with $\delta^-(r) = \emptyset$).*
(b) *G is a branching with $n - 1$ vertices and $\delta^-(r) = \emptyset$.*
(c) *G has $n - 1$ edges and every vertex is reachable from r.*
(d) *Every vertex is reachable from r, but deleting any edge destroys this property.*
(e) *G satisfies $\delta^+(X) \neq \emptyset$ for all $X \subset V(G)$ with $r \in X$, but deleting any edge of G destroys this property.*
(f) *$\delta^-(r) = \emptyset$ and for every $v \in V(G)$ there exists a uniquely determined edge progression from r to v.*
(g) *$\delta^-(r) = \emptyset$ and $|\delta^-(v)| = 1$ for all $v \in V(G) \setminus \{r\}$, and G contains no circuits.*

Proof

(a)⇔(b) and (c)⇒(d) These implications follow by Theorem 6.18 applied to the underlying undirected graph of G.

(b)⇒(c) We have $|\delta^-(v)| = 1$ for all $v \in V(G) \setminus \{r\}$. Thus we have an r-v-path for every v (begin in v and keep following the arriving edges all the way to r).

(d)⇔(e) This follows with Theorem 6.21.

(d)⇒(f) The minimality property described in (d) implies $\delta^-(r) = \emptyset$. Assume that for some v there exist two different edge progressions P and Q from r to v. Then $v \neq r$. Choose v, P and Q so that the sum of the lengths of the two edge progressions is minimal. Then clearly P and Q already differ with respect to their last edge (ending in v). This implies that deleting one of these edges will not destroy the reachability of all vertices from r.

(f)⇒(g) If every vertex is reachable from r and $|\delta^-(v)| > 1$ for some $v \in V(G) \setminus \{r\}$, then there exist two edge progressions from r to v. If G contains a circuit C, take $v \in V(C)$ and consider the r-v-path P. Then P followed by C yields a second edge progression from r to v.

(g)⇒(b) If $|\delta^-(v)| \leq 1$ holds for all $v \in V(G)$, then every subgraph whose underlying undirected graph is a circuit, is itself a (directed) circuit. Thus (g) implies that G is a branching. □

Corollary 6.24 *A digraph G is strongly connected if and only if it contains, for every $r \in V(G)$, a spanning arborescence with root r.*

Proof This follows immediately from (d)⇔(a) of Theorem 6.23. □

6.5 Digression: Elementary Data Structures

A data structure is an object (called a class in C++) in which a set of different objects ("elements") of the same type can be stored, together with a set of operations which can be performed on the elements. Examples of typical operations are:

- creating an empty data structure (constructor)
- adding an element
- finding an element
- deleting an element
- scanning all elements
- deleting a data structure (destructor)

Perhaps the simplest data structure is an **array**, comprising a fixed number of sequentially arranged memory spaces of the same type. Each of these spaces is identified by an index, by means of which they can be addressed for reading them or writing in them (this is called random access). Arrays can also be multidimensional, e.g. for storing matrices. In C++ such an array can, after inserting `#include <array>`, be defined with `std::array<`*`typename`*`, `*`number`*`>;`. Here *typename* denotes the type of elements to be stored in the array and *number* denotes their number.

In C++, vector also comprises such an array. However, the length of an array implemented with vector is variable, so that an operation like push_back becomes possible. Here, if necessary, the array is copied internally into a larger array.

In an array (and also in vector), accessing an object (and therefore also deleting it) normally necessitates scanning all entries, unless, of course, its index is known. If the stored objects are in a certain order and if the entries of the array have been sorted accordingly, then one can access an entry far faster with binary search (Algorithm 5.2), namely in $O(\log n)$ time, where n is the length of the array. However, inserting an object is then more complicated if the order is to be preserved: it will require $O(n)$ time. Deleting an element (even when its index is known) will also require $O(n)$ time if one wants to have no gaps in the array and wants to preserve the order of the remaining elements.

One can achieve the latter very efficiently by using **lists**. Again, the elements of a list are always given in a certain order. But the various elements in a list can be stored anywhere in the memory, in places that are totally independent of one another. For every element one notes a pointer to the place of the next element (the successor). For the last element of the list this is the null-pointer (called nullptr in C++), which marks the end of the list. In addition one requires another pointer to the first element of a list in order to be able to scan the list.

Lists can be singly or doubly linked. For every element of a doubly linked list (cf. Fig. 6.2) one additionally notes a pointer to the predecessor. This enables the fast deletion of an element whose place is known: the number of computational steps is bounded by a constant that is independent of the size of the list. One also says: the running time is $O(1)$.

Lists have the following disadvantage: one cannot apply binary search to them. Moreover, scanning a list is slower (albeit by a constant factor only) than scanning an array, because addressing consecutive storage places is substantially faster with today's computers than addressing far apart ones.

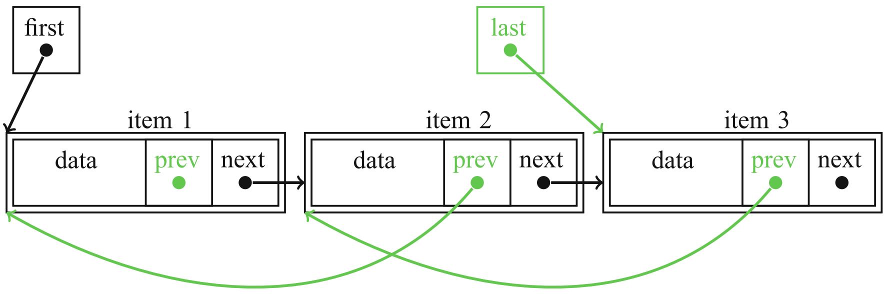


Fig. 6.2 A doubly linked list with three elements. Every element consists of a data entry, a pointer to the previous element and a pointer to the next element. Points with no leaving *arrow* symbolize the nullptr. The elements are stored in a heap. The variables *first* and *last* are pointers lying in the stack. Deleting all the *green* parts leaves a singly linked list

Another elementary data structure is the so-called **stack**. In a stack elements are stored by adding them "to the top", and only the "top" (i.e. the last added) element can be deleted. Here one speaks of a LIFO memory (last in, first out). This can, for example, be achieved with the operations `push_back` and `pop_back` of `vector`. A part of the main memory is organized as a stack; see Sect. 2.2.

On the other hand, a **queue** is, in a way, the opposite of a stack. Here, too, elements are always stored by adding them to one end, but they are deleted by removing them from the other end. Thus one speaks here of a FIFO memory (first in, first out).

A queue can very easily be implemented as a (singly linked) list, as Program 6.25 shows. As the type of objects that we want to store in the queue differs with the application, we do not fix it but instead use the `template` construct. This is explained in more detail in Box **C++ in Detail (6.1)**. The commands `new` and `delete` request the required storage space (on the heap; cf. Sect. 2.2) or release it, respectively. The class `queue` contains a subclass `item`, where an `item` comprises an element of type `T` (the template parameter) and a pointer to the next `item`.

Program 6.25 (Queue)

```
// queue.h (Queue)

template <typename T> class Queue {  // T is a type to be specified by user
public:
    ~Queue()                         // destructor
    {
        clear();
    }

    bool is_empty() const
    {
        return _front == nullptr;
    }

    void clear()
    {
        while (not is_empty()) {
            pop_front();
        }
    }

    void push_back(const T & object)   // insert object at end of queue
    {
        Item * cur = new Item(object); // get new memory for Item at address cur,
                                       // initialize with object and nullptr
        if (is_empty()) {
            _front = cur;
        }
        else {
            _back->_next = cur;      // p->n is abbreviation for (*p).n
        }
        _back = cur;
    }

    T pop_front()                    // delete and return first object of queue
    {                                // ATTENTION: queue must not be empty!
        Item * cur = _front;
        if (_back == _front) {
            _front = nullptr;
            _back  = nullptr;
        }
        else {
            _front = _front->_next;
        }
        T object = cur->_object;
        delete cur;                  // free memory for 1 Item at address cur
        return object;
    }

private:
    struct Item {                                    // struct is a class where by
        Item(const T & object) : _object(object) {} // default everything is public

        T _object;
        Item * _next = nullptr;     // pointer to the next Item (or nullptr)
    };

    Item * _front = nullptr;        // _front and _back are pointers to
    Item * _back = nullptr;         // variables of type Item, or the
};                                  // nullptr if queue is empty
```

C++ in Detail (6.1): Class Templates
With the help of so-called **template parameters** one can substantially increase the usability of a class and thereby avoid the repeated appearance of very similar pieces of code. We have already seen the use of template parameters in connection with the abstract data type `vector`: by means of `vector<`*`typename`*`>` one can define a `vector` whose elements are of type *`typename`*. In order to equip a class with a template parameter, one just writes, as on line 3 of Program 6.25, `template<typename T>` in front of the keyword `class`. Instead of `T` we can use an arbitrary symbol. Within the class definition of `queue` one can then define and use objects of type `T`. The command `queue<`*`typename`*`> q;` defines a queue whose elements are of the stated type. Here *`typename`* is not only a standard data type of C++, but can be an arbitrary type or class. When defining a class, one can also stipulate more than one template parameter. These must then be separated by commas.

The implementation of a queue shown in Program 6.25 just serves as a simple example of how to write a program for an abstract data type. Actually, our implementation is not very efficient, as every addition or removal of an element is accompanied by a request or a release respectively of storage space on the heap. A substantially more efficient implementation of a queue is contained in the C++ Standard Library. After inserting `#include <queue>` one can define a queue with `std::queue<`*`datatype`*`>`. This method supports far more operations than those implemented in Program 6.25. In the next chapter we will meet an application of the abstract data type queue.

6.6 Representations of Graphs

A graph is easily stored in memory by storing the number n of its vertices, the number m of its edges and, for every $i \in \{1, \dots, m\}$, the numbers of the endpoints of the i-th edge (with respect to a fixed numbering of the vertices and edges of the graph). This is, however, not very convenient. With other data structures it is possible to decide more quickly with which edges a certain vertex is incident; this is required in nearly every algorithm on graphs.

Definition 6.26 Let $G = (V, E, \psi)$ be a graph with n vertices and m edges. The matrix $A = (a_{x,y})_{x,y \in V} \in \mathbb{Z}^{n \times n}$ is called the **adjacency matrix** of G, where

$$a_{x,y} = |\{e \in E : \psi(e) = \{x, y\} \text{ or } \psi(e) = (x, y) \text{ respectively}\}| .$$

The matrix $A = (a_{x,e})_{x \in V, e \in E} \in \mathbb{Z}^{n \times m}$ is called the **incidence matrix** of G, where

$$a_{x,e} = \begin{cases} 1, & \text{if } x \text{ is an endpoint of } e \\ 0, & \text{otherwise ,} \end{cases}$$

if G is an undirected graph, and

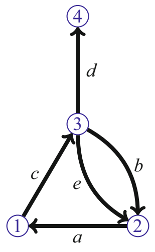
$$a_{x,e} = \begin{cases} -1, & \text{if } e \text{ begins in } x \\ 1, & \text{if } e \text{ ends in } x \\ 0, & \text{otherwise ,} \end{cases}$$

if G is a directed graph.

Example 6.27

adjacency matrix

	1	2	3	4
1	0	0	1	0
2	1	0	0	0
3	0	2	0	1
4	0	0	0	0



incidence matrix

	a	b	c	d	e
1	1	0	−1	0	0
2	−1	1	0	0	1
3	0	−1	1	−1	−1
4	0	0	0	1	0

An essential drawback of these matrices are their high memory requirements. In order to make this more precise, we need to extend the Landau symbols (Definition 1.13) to graphs.

Let $\mathcal{G}$ be the set of all graphs, $f: \mathcal{G} \to \mathbb{R}_{\geq 0}$ and $g: \mathcal{G} \to \mathbb{R}_{\geq 0}$. Then we will say that $f = O(g)$ if there exist $\alpha > 0$ and $n_0 \in \mathbb{N}$ such that $f(G) \leq \alpha \cdot g(G)$ for all $G \in \mathcal{G}$ with $|V(G)| + |E(G)| \geq n_0$. In other words: if f is greater than g, then by at most a constant factor, possibly with a finite number of exceptions (this notation can also be used for any countable set instead of $\mathcal{G}$). The Ω and Θ notations are extended analogously. The function f will usually describe the running time of an algorithm or some memory requirement; g will often depend only on the number of vertices and edges.

One can therefore say: the memory requirements for the adjacency matrix and the incidence matrix are $\Omega(n^2)$ and $\Theta(nm)$ respectively, where $n = |V(G)|$ and $m = |E(G)|$. For graphs, for example, with $\Theta(n)$ edges (as is often the case), this is far more than required.

Using **adjacency lists** for representing graphs often requires less memory. Here one notes, for every vertex, a list of all the edges with which it is incident (or, for simple graphs, sometimes only a list of all the adjacent vertices). For directed graphs one of course has two lists, one for incoming and another for outgoing edges.

Example 6.28 An adjacency list representation of the digraph depicted in Example 6.27 has the following form. Here the symbol ■ marks the end of a list in each case.

$$\boxed{\delta^+(1)} \to \boxed{c} \to \blacksquare \qquad \boxed{\delta^-(1)} \to \boxed{a} \to \blacksquare \qquad \boxed{a} = (2,1)$$

$$\boxed{\delta^+(2)} \to \boxed{a} \to \blacksquare \qquad \boxed{\delta^-(2)} \to \boxed{e} \to \boxed{b} \to \blacksquare \qquad \boxed{b} = (3,2)$$

$$\boxed{\delta^+(3)} \to \boxed{b} \to \boxed{d} \to \boxed{e} \to \blacksquare \qquad \boxed{\delta^-(3)} \to \boxed{c} \to \blacksquare \qquad \boxed{c} = (1,3)$$

$$\boxed{\delta^+(4)} \to \blacksquare \qquad \boxed{\delta^-(4)} \to \boxed{d} \to \blacksquare \qquad \boxed{d} = (3,4) \qquad \boxed{e} = (3,2)$$

The adjacency lists can, as required, comprise singly or doubly linked lists or arrays. In general, one must assume that the edges in the adjacency lists are not ordered in any way.

For most purposes an adjacency list is the preferred data structure, especially as it permits scanning $\delta(v)$ or $\delta^+(v)$ and $\delta^-(v)$ for every vertex v in linear time, and because the memory requirement is proportional to the number of vertices and edges (if one assumes, as usual, that pointers, vertex indices and edge indices require only a constant amount of memory each). Hence we will use this form in all the following algorithms. Accessing the next edge in a list of edges or an endpoint of an edge are thus considered to be elementary operations.

We will now present an implementation in C++. The class `graph` permits the storing of directed and undirected graphs. The variable `dirtype` determines the type of graph. There are two different constructors for the graph type. One constructor has the number of vertices as first parameter, while the other has a filename as first parameter. Both constructors have a second parameter which determines whether the generated graph is to be directed or undirected.

The class `Graph` contains the subclasses `Node` for vertices and `Neighbor` for the neighbors of a vertex. For a digraph we store as neighbors of a specific vertex all the vertices reachable via leaving edges. This allows one to implement some graph algorithms in such a way that they work on both directed and undirected graphs, making the separate treatment of these cases unnecessary.

The program `testgraph.cpp` reads a graph from a file. The name of the file is entered as a command line parameter into the program, see Box **C++ in Detail (6.2)**. We assume here that the relevant file contains the number of vertices of the graph in its first line and further, that the vertices are numbered consecutively beginning with zero. The file has a separate line for every edge of the graph, on which two values and optionally a third one are listed, namely the numbers of the endpoints of the relevant edge and its weight.

C++ in Detail (6.2): Command Line Parameters
The function `main` can have two parameters, in which case they should look as in the program `testgraph.cpp`. If the program is called up using "`programname par1`...`par`n", then `argc` $= n+1$, `argv[i]` $=$ `par`i for $i = 1, \dots, n$, and `argv[0]` $=$ `programname`. The type `char*` is used for C-style strings (which one in fact only requires as arguments in the `main`-function): a pointer to a symbol which can be followed by further symbols; these are all part of the string until the zero-symbol (ASCII-Code 0) appears (which clearly does not itself belong to the string).

If `T` is a type and `p` a pointer of type `T*`, then `p[i]` denotes a variable of type `T` with storage address `p` $+ i \cdot$ `sizeof(T)`.

Program 6.29 (Graphs)

```
// graph.h (Declaration of Class Graph)
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <vector>

class Graph {
public:
  using NodeId = int;  // vertices are numbered 0,...,num_nodes()-1

  class Neighbor {
  public:
        Neighbor(Graph::NodeId n, double w);
        double edge_weight() const;
        Graph::NodeId id() const;
  private:
        Graph::NodeId _id;
        double _edge_weight;
    };

  class Node {
  public:
        void add_neighbor(Graph::NodeId nodeid, double weight);
        const std::vector<Neighbor> & adjacent_nodes() const;
  private:
        std::vector<Neighbor> _neighbors;
    };

  enum DirType {directed, undirected};  // enum defines a type with possible values
  Graph(NodeId num_nodes, DirType dirtype);
  Graph(char const* filename, DirType dirtype);

  void add_nodes(NodeId num_new_nodes);
  void add_edge(NodeId tail, NodeId head, double weight = 1.0);

  NodeId num_nodes() const;
  const Node & get_node(NodeId) const;
  void print() const;

  const DirType dirtype;
  static const NodeId invalid_node;
  static const double infinite_weight;

private:
  std::vector<Node> _nodes;
};

#endif // GRAPH_H
```

```
// graph.cpp (Implementation of Class Graph)

#include <fstream>
#include <sstream>
#include <stdexcept>
#include <limits>
#include "graph.h"

const Graph::NodeId Graph::invalid_node = -1;
const double Graph::infinite_weight = std::numeric_limits<double>::max();


void Graph::add_nodes(NodeId num_new_nodes)
{
   _nodes.resize(num_nodes() + num_new_nodes);
}

Graph::Neighbor::Neighbor(Graph::NodeId n, double w): _id(n), _edge_weight(w) {}

Graph::Graph(NodeId num, DirType dtype): dirtype(dtype), _nodes(num) {}

void Graph::add_edge(NodeId tail, NodeId head, double weight)
{
   if (tail >= num_nodes() or tail < 0 or head >= num_nodes() or head < 0) {
       throw std::runtime_error("Edge cannot be added due to undefined endpoint.");
   }
   _nodes[tail].add_neighbor(head, weight);
   if (dirtype == Graph::undirected) {
        _nodes[head].add_neighbor(tail, weight);
   }
}

void Graph::Node::add_neighbor(Graph::NodeId nodeid, double weight)
{
   _neighbors.push_back(Graph::Neighbor(nodeid, weight));
}

const std::vector<Graph::Neighbor> & Graph::Node::adjacent_nodes() const
{
   return _neighbors;
}
```

```

Graph::NodeId Graph::num_nodes() const
{
   return _nodes.size();
}

const Graph::Node & Graph::get_node(NodeId node) const
{
   if (node < 0 or node >= static_cast<int>(_nodes.size())) {
        throw std::runtime_error("Invalid nodeid in Graph::get_node.");
   }
   return _nodes[node];
}

Graph::NodeId Graph::Neighbor::id() const
{
   return _id;
}

double Graph::Neighbor::edge_weight() const
{
   return _edge_weight;
}

void Graph::print() const
{
   if (dirtype == Graph::directed) {
        std::cout << "Digraph ";
   } else {
        std::cout << "Undirected graph ";
   }
   std::cout << "with " << num_nodes() << " vertices, numbered 0,...,"
              << num_nodes() - 1 << ".\n";

   for (auto nodeid = 0; nodeid < num_nodes(); ++nodeid) {
        std::cout << "The following edges are ";
        if (dirtype == Graph::directed) {
            std::cout << "leaving";
        } else {
            std::cout << "incident to";
        }
        std::cout << " vertex " << nodeid << ":\n";
        for (auto neighbor: _nodes[nodeid].adjacent_nodes()) {
            std::cout << nodeid << " - " << neighbor.id()
                      << " weight = " << neighbor.edge_weight() << "\n";
        }
   }
}

Graph::Graph(char const * filename, DirType dtype): dirtype(dtype)
{
   std::ifstream file(filename);                            // open file
   if (not file) {
        throw std::runtime_error("Cannot open file.");
   }

   Graph::NodeId num = 0;
   std::string line;
   std::getline(file, line);                 // get first line of file
   std::stringstream ss(line);               // convert line to a stringstream
   ss >> num;                                // for which we can use >>
   if (not ss) {
        throw std::runtime_error("Invalid file format.");
   }
   add_nodes(num);

   while (std::getline(file, line)) {
        std::stringstream ss(line);
        Graph::NodeId head, tail;
        ss >> tail >> head;
        if (not ss) {
            throw std::runtime_error("Invalid file format.");
        }
        double weight = 1.0;
        ss >> weight;
        if (tail != head) {
            add_edge(tail, head, weight);
        }
        else {
            throw std::runtime_error("Invalid file format: loops not allowed.");
        }
    }
}
```

```
// testgraph.cpp (Read Digraph from File and Print)

#include "graph.h"

int main(int argc, char* argv[])
{
    if (argc > 1) {
        Graph g(argv[1], Graph::directed);
        g.print();
    }
}
```

7 Simple Graph Algorithms

We will now go on to introduce some simple graph algorithms. We will begin with the "exploration" of a graph: to discover, for example, which vertices are reachable from a given vertex. The presented algorithms will provide much more information and are used in numerous applications.

7.1 Graph Traversal Methods

We begin by presenting the general Graph Traversal Algorithm in pseudocode.

Algorithm 7.1 (Graph Traversal)

Input: a graph G, a vertex $r \in V(G)$.
Output: the set $R \subseteq V(G)$ of vertices reachable from r and a set $F \subseteq E(G)$ such that (R, F) is an arborescence with root r or that (R, F) is a tree.

```
$R \leftarrow \{r\}, Q \leftarrow \{r\}, F \leftarrow \emptyset$
**while** $Q \neq \emptyset$ **do**
    choose a $v \in Q$
    **if** $\exists\, e = (v, w) \in \delta^+_G(v)$ or $e = \{v, w\} \in \delta_G(v)$ with $w \in V(G) \setminus R$
        **then** $R \leftarrow R \cup \{w\}, Q \leftarrow Q \cup \{w\}, F \leftarrow F \cup \{e\}$
        **else** $Q \leftarrow Q \setminus \{v\}$
**output**$(R, F)$
```

Theorem 7.2 *Algorithm 7.1 works correctly and can be implemented to run in $O(n + m)$ time, where $n = |V(G)|$ and $m = |E(G[R])| \leq |E(G)|$.*

Proof We begin with the following claim: at every point in time (R, F) is a tree or an arborescence with root r in G. This certainly holds at the start of the algorithm. A change in (R, F) occurs only when a vertex w and an edge e from v to w is added, where v was already in R but w was not. Thus (R, F) remains a tree or an arborescence with root r.

Suppose that at the end of the algorithm there exists a vertex w which is not in R but is reachable from r. Let P be an r-w-path and $\{x, y\}$ or (x, y) an edge in P with $x \in R$ and $y \notin R$. As $x \in R$, it follows that x was also in Q at some point in time. The algorithm does not end before x has been removed from Q. This, however, happens only if there is no edge (x, y) or $\{x, y\}$ with $y \notin R$; contradiction.

It remains to analyze the running time. Edges and vertices outside of $G[R]$ are never considered except when all the vertices are initially marked as "not in R". For every visited vertex x we will always note (in its list $\delta(x)$ or $\delta^+(x)$) the current position up to where the edges have already been considered. At the beginning (when adding to Q) this is the starting position of the list. Thus every edge of $G[R]$ is considered at most twice (in fact only once in the directed case).

Every vertex reachable from r is added to Q and removed from Q exactly once. So we require a data structure for Q which allows the addition, removal and selection of an element in constant time in each case. Stacks and queues, for example, have this property. Thus the total running time for all operations on Q is $O(m)$, because $|R| \leq |E(G[R])| + 1$, as $G[R]$ is connected. □

We also have:

Corollary 7.3 *The connected components of an undirected graph G can be found in $O(n + m)$ time, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof Start the Graph Traversal Algorithm in any vertex of G. If $R = V(G)$, then G is connected. If not, then $G[R]$ is a connected component of G and one iterates with $G[V(G) \setminus R]$. Initializing all the vertices need only be done once at the start. □

If a graph algorithm has running time $O(n+m)$ with $n = |V(G)|$ and $m = |E(G)|$, then we also say that it has linear running time.

Algorithm 7.1 can be implemented in different ways. In particular, there are several alternatives for choosing $v \in Q$. If one always chooses the vertex $v \in Q$ that was last added to Q (LIFO; so Q is a stack), then the algorithm is called **Depth-First Search** or DFS. If, on the other hand, one always chooses the vertex Q that was the first to be added to Q (FIFO; so Q is a queue), then the algorithm is called **Breadth-First Search** or BFS. A tree or arborescence (R, F) computed by DFS or by BFS is called a DFS-tree or a BFS-tree respectively.

Example 7.4 Figure 7.1 depicts examples for DFS and BFS. These two variants have interesting properties; we will take a closer look at BFS below.

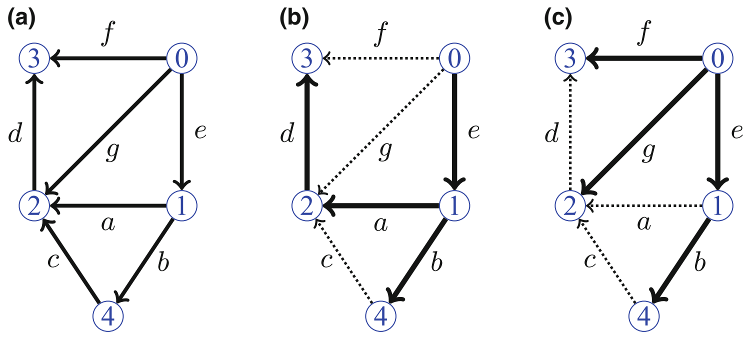


Fig. 7.1 (**a**) A digraph; (**b**) a possible result of DFS; (**c**) a result of BFS; both were started in vertex 0 and the edges of F are thick

7.2 Breadth-First Search

For a start we present an implementation of Breadth-First Search. For each vertex we store the length of the path in the BFS-tree by means of the variable `dist`. This is actually not necessary: see later in Theorem 7.6.

Program 7.5 (Breadth-First Search)

```cpp
// bfs.cpp (Breadth First Search)

#include "graph.h"
#include "queue.h"

Graph bfs(const Graph & graph, Graph::NodeId start_node)
{
    std::vector<bool> visited(graph.num_nodes(), false);
    std::vector<int> dist(graph.num_nodes(), -1);
    Graph bfs_tree(graph.num_nodes(), graph.dirtype);
    Queue<Graph::NodeId> queue;

    std::cout << "The following vertices are reachable from vertex "
              << start_node << ":\n";
    queue.push_back(start_node);
    visited[start_node] = true;
    dist[start_node] = 0;

    while (not queue.is_empty()) {
        auto cur_nodeid = queue.pop_front();
        std::cout << "Vertex " << cur_nodeid << " has distance "
                  << dist[cur_nodeid] << ".\n";
        for (auto neighbor: graph.get_node(cur_nodeid).adjacent_nodes()) {
            if (not visited[neighbor.id()]) {
                visited[neighbor.id()] = true;
                dist[neighbor.id()] = dist[cur_nodeid] + 1;
                bfs_tree.add_edge(cur_nodeid, neighbor.id());
                queue.push_back(neighbor.id());
            }
        }
    }

    return bfs_tree;
}
```

```cpp


int main(int argc, char* argv[])
{
    if (argc > 1) {
        Graph g(argv[1], Graph::directed);           // read digraph from file
        Graph bfs_tree = bfs(g, 0);
        std::cout << "The following is a BFS-tree rooted at 0:\n";
        bfs_tree.print();
    }
}
```

For any two vertices v and w in a graph G we denote by $\mathrm{dist}(v, w)$ the length of a shortest v-w-path in G. We also call $\mathrm{dist}(v, w)$ the **distance** from v to w in G. If there is no v-w-path in G, then we set $\mathrm{dist}(v, w) := \infty$. In undirected graphs $\mathrm{dist}(v, w) = \mathrm{dist}(w, v)$ holds for all $v, w \in V(G)$.

Theorem 7.6 *Every BFS-tree contains a shortest path from a vertex r to all other vertices reachable from r. The values of* $\mathrm{dist}(r, v)$ *can be found for all $v \in V(G)$ in linear time.*

Proof Program 7.5 is an implementation of Algorithm 7.1 with Q as a queue, so we have BFS. Clearly at every point in time the computed values of `dist[i]` yield the distances from r to all vertices in `bfs_tree` (this graph, called T from now on, was called (R, F) in Algorithm 7.1). Thus it remains to show:

$$\mathrm{dist}_G(r, v) = \mathrm{dist}_T(r, v) \qquad \text{for all } v \in V(G)\,.$$

As "$\leq$" is clear (T is a subgraph of G), we will suppose that there is a vertex $v \in V(G)$ with $\mathrm{dist}_G(r, v) < \mathrm{dist}_T(r, v)$. Let v be chosen such that $\mathrm{dist}_G(r, v)$ is minimum. Let P be a shortest r-v-path in G and (u, v) or $\{u, v\}$ its last edge. Then $\mathrm{dist}_G(r, u) = \mathrm{dist}_T(r, u)$ and hence

$$\mathrm{dist}_T(r, v) > \mathrm{dist}_G(r, v) = \mathrm{dist}_G(r, u) + 1 = \mathrm{dist}_T(r, u) + 1\,. \tag{7.1}$$

At any time during the algorithm we have that:

(a) $\mathrm{dist}_T(r, x) \leq \mathrm{dist}_T(r, y) + 1$ for all $x \in R$ and $y \in Q$.
(b) $\mathrm{dist}_T(r, x) \leq \mathrm{dist}_T(r, y)$ for all $x, y \in Q$ if x was added to Q before y.

This is easily proved by induction over the number of steps performed so far.

By (a) and (7.1), v will not be added to R until u has been removed from Q. But then, because of edge (u, v) or $\{u, v\}$, we have that $\mathrm{dist}_T(r, v) \leq \mathrm{dist}_T(r, u) + 1$ which is a contradiction. □

Properties like (a) and (b) which always hold during the algorithm, are also called invariants. They are often very useful when showing that an algorithm runs correctly.

7.3 Bipartite Graphs

The following special undirected graphs often appear:

Definition 7.7 A **complete graph** is a simple undirected graph with the property that any two of its vertices are joined by an edge. A complete graph with an n-element vertex set is often denoted by $\boldsymbol{K_n}$.

A **bipartition** of an undirected graph G consists of two disjoint sets of vertices A and B whose union is $V(G)$, with the property that every edge in $E(G)$ has exactly one endpoint in A. A graph is called **bipartite** if it possesses a bipartition. The notation $G = (A \dot{\cup} B, E(G))$ signifies that the sets A and B form a bipartition.

A **complete bipartite graph** is a graph G with the bipartition $V(G) = A \dot{\cup} B$ and $E(G) = \{\{a, b\} : a \in A,\ b \in B\}$. If $|A| = n$ and $|B| = m$, then it is often denoted by $\boldsymbol{K_{n,m}}$.

An **odd circuit** is a circuit of odd length.

Theorem 7.8 (König [23]) *An undirected graph is bipartite if and only if it contains no odd circuits. In a given undirected graph one can find either a bipartition or an odd circuit in linear time.*

Proof We start by showing that a bipartite graph contains no odd circuits. Let G be bipartite with bipartition $V(G) = A \dot{\cup} B$ and suppose that C is a circuit of length k in G with vertex set $\{v_1, \dots, v_k\}$ and edges $\{v_i, v_{i+1}\}$ for $i = 1, \dots, k$, where $v_{k+1} := v_1$. We can assume without loss of generality that $v_1 \in A$. Then we have $v_2 \in B$, $v_3 \in A$ etc., i.e. $v_i \in A$ if and only if i is odd. Then $v_{k+1} = v_1 \in A$ implies that k is even.

We will now prove the second statement of the theorem (which implies the first). We can assume without loss of generality that G is connected. (Otherwise, determine a bipartition for every connected component and combine these.)

Applying BFS to G starting with a vertex r yields a spanning tree $T = (V(G), F)$. Let

$$A := \{v \in V(G) : \mathrm{dist}_T(r, v) \text{ is even}\} \quad \text{and} \quad B := V(G) \setminus A.$$

If $e \in \delta(A)$ for all $e \in E(G)$, then $A \dot{\cup} B$ is a bipartition. Otherwise there is an edge $e = \{v, w\}$ with $v, w \in A$ or $v, w \in B$. Let C_e be the circuit in $(V(G), F \dot{\cup} \{e\})$. It consists of e, the u-v-path in T and the u-w-path in T, where u is the last common vertex of the r-v-path in T and the r-w-path in T. Then we have $\mathrm{dist}_T(r, v) + \mathrm{dist}_T(r, w) = 2\,\mathrm{dist}_T(r, u) + |E(C_e)| - 1$. As the left-hand side of this equation is even, C_e is an odd circuit. □

Note that one could, in the above proof, have taken any spanning tree instead of T.

Theorem 7.8 provides a good characterization of the "bipartite" property of graphs: it is not only easy to prove that a graph has this property (by defining a bipartition) but also, that a graph does not have this property (by showing an odd circuit). Both of these "proofs" are easily checked.

7.4 Acyclic Digraphs

The following definition describes an important type of directed graph:

Definition 7.9 A digraph is called **acyclic** if it contains no (directed) circuit.

Let G be a digraph with n vertices. A **topological order** of G is an ordering of the vertices $V(G) = \{v_1, \dots, v_n\}$ such that $i < j$ holds for every edge $(v_i, v_j) \in E(G)$.

We will require the following lemma:

Lemma 7.10 *Let G be a digraph with $\delta^+(v) \neq \emptyset$ for all $v \in V(G)$. Then one can find a circuit in G in $O(|V(G)|)$ time.*

Proof One simply begins at any vertex and keeps taking any leaving edge until one meets a former vertex. □

The following theorem provides a good characterization:

Theorem 7.11 *A digraph possesses a topological order if and only if it is acyclic. For a given digraph G one can find either a topological order of G or a circuit in G in linear time.*

Proof If G possesses a topological order, then G clearly does not contain a circuit.

We will now prove the second statement of the theorem (which implies the first). Let $n := |V(G)|$ and $m := |E(G)|$. First we will compute the out-degree $a(v) := |\delta^+(v)|$ for all $v \in V(G)$ and store all vertices v with $a(v) = 0$ in a list L_0. This we can do in linear time.

If L_0 is empty, then by Lemma 7.10 we can find a circuit in $O(n)$ steps. If not, then choose a vertex in L_0 and denote it v_n. Delete v_n in L_0 and do the following for all $(u, v_n) \in \delta^-(v_n)$: reduce $a(u)$ by 1 and if this yields $a(u) = 0$, then add u to L_0. This can clearly be done in $O(1 + |\delta^-(v_n)|)$ time.

We now have: L_0 and $a(v)$ with $v \in V(G) \setminus \{v_n\}$ are correct for $G - v_n$. So we reduce n by 1 and iterate while $n > 0$.

Thus we can find either a circuit or a topological order $v_1, \dots, v_n$ of G in a total of $O(n + m)$ time. □

Sorting Algorithms 8

On many occasions stored data files have to be sorted. There are essentially two reasons for this: on the one hand certain algorithms require that the objects are in a specified order and on the other hand, in a sorted data file with random access one can find individual objects much faster (with binary search, cf. Algorithm 5.2).

8.1 The General Sorting Problem

Definition 8.1 Let S be a set. A relation $R \subseteq S \times S$ is called a **partial order** (on S), if for all $a, b, c \in S$:

- $(a, a) \in R$ (reflexivity);
- $((a, b) \in R \wedge (b, a) \in R) \Rightarrow a = b$ (antisymmetry);
- $((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$ (transitivity).

Instead of $(a, b) \in R$ one often writes aRb. A partial order R is called a **total order** on S if $(a, b) \in R$ or $(b, a) \in R$ holds for all $a, b \in S$.

For example, the usual "less than or equal to" relation "$\leq$" is a total order on $\mathbb{R}$. On the other hand, the subset relation "$\subseteq$" is a partial but not generally a total order on an arbitrary family of sets. In a directed graph G the relation $R := \{(v, w) \in V(G) \times V(G) : w \text{ is reachable from } v\}$ is a partial order if and only if G is acyclic.

For finite sets S a total order $\preceq$ can be defined by numbering the elements, i.e. by a bijection $f : S \to \{1, \ldots, n\}$ with $f(s) = |\{a \in S : a \preceq s\}|$ for all $s \in S$. We can now define:

Computational Problem 8.2 (General Sorting Problem)

Input: a finite set S with a partial order $\preceq$ (given by an oracle).
Task: compute a bijection $f : \{1, \ldots, n\} \to S$ with $f(j) \not\preceq f(i)$ for all $1 \leq i < j \leq n$.

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_8

We will thus assume for the present that we know nothing more about the partial order than that we can, for any two elements a and b, ask whether or not $a \preceq b$ holds. We can, of course, use the known properties of the partial order, in particular its transitivity.

As a rule the input is given in the form of a bijection $h: \{1, \ldots, n\} \to S$ plus the oracle. Then the objective is to determine a permutation π (i.e. a bijection $\pi: \{1, \ldots, n\} \to \{1, \ldots, n\}$) such that $f: i \mapsto h(\pi(i))$ has the desired property.

It is sometimes possible to sort more quickly if the input is given differently (Theorem 7.11 provides an example of this), and/or the partial order has additional properties (e.g. is a total order); we will return to this from Sect. 8.3 onwards.

8.2 Sorting by Successive Selection

A naive sorting method would be, to scan all $n!$ permutations and, for each n, to test all the $n(n-1)/2$ ordered pairs (i, j) with $1 \le i < j \le n$ as to whether $f(j) \not\preceq f(i)$ holds. This is obviously very inefficient.

We will now present a better method: Sorting by Successive Selection. Here one determines $f(i) := s$ stepwise for $i = 1, \ldots, n$ in such a way that for all $t \in S \setminus \{f(1), \ldots, f(i-1)\}$ we have $t \preceq s \Rightarrow t = s$.

Algorithm 8.3 (**Sorting by Successive Selection**)

Input: a set $S = \{s_1, \ldots, s_n\}$; a partial order $\preceq$ on S (given by an oracle).
Output: a bijection $f: \{1, \ldots, n\} \to S$ with $f(j) \not\preceq f(i)$ for all $1 \le i < j \le n$.

```
for $i \leftarrow 1$ to $n$ do $f(i) \leftarrow s_i$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        if $f(j) \preceq f(i)$ then swap($f(i)$, $f(j)$)
output $f$
```

The iterations with $i = j$ are clearly superfluous, but they simplify the proof of the following theorem. The **swap** command interchanges the values of two variables. We will now show:

Theorem 8.4 *Algorithm 8.3 solves the general sorting Problem 8.2 correctly and has running time $O(n^2)$.*

Proof The running time is clear. The function f is always a bijection as the only thing that happens is an interchange of two values of f (by means of the **swap** command).

Next, we show that for all $1 \le i \le j \le n$, after iteration (i,j) the following two conditions hold:

(a) $f(k) \not\preceq f(h)$ for all $1 \le h < i$ and $h < k \le n$;
(b) $f(k) \not\preceq f(i)$ for all $i < k \le j$.

For iteration $(1, 1)$ both conditions are satisfied.

Condition (a) can only cease to hold when i increases; that this does not happen follows from condition (b) for $j = n$, because i is not increased until $j = n$.

It remains to show that condition (b) is always satisfied. Consider the state at the start of iteration (i,j). If $f(j) \not\preceq f(i)$, then the value of f does not change here and thus (b) continues to be satisfied.

If $f(j) \preceq f(i)$, let k be an index with $i < k \le j$. If $k = j$, then because $f(i) \not\preceq f(j)$ (antisymmetry) condition (b) is satisfied after the **swap** command. If $k < j$, condition (b) was satisfied earlier and so $f(k) \not\preceq f(i)$ which (together with transitivity) implies $f(k) \not\preceq f(j)$. Hence the **swap** command does not destroy condition (b).

At the end of the algorithm, i.e. after iteration (n, n), condition (a) implies its correctness. □

The function `sort1` in Program 8.5 presents an implementation of this algorithm. The implementation and also the example were chosen so as to allow us to illustrate several other possibilities offered by C++.

Note in particular the abstract interface of the function `sort1`. One is provided with only two iterators: a reference `first` to the first element (which can, for example, be a pointer or an index), and a reference `last` to the place *behind* the last element, as well as a function `comp` with which one can compare two elements. An iterator (see Box **C++ in Detail (2.3)**) must provide at least three operations:

`*` dereference operator (using the same syntax as with pointers);
`++` increment operator (moves iterator to the next element);
`!=` comparison operator (in particular combined with `last` for determining whether any further elements remain).

The comparison function `comp` could in fact be a Boolean function, but here it is a variable of type `BirthdayComparison`: a class which provides the operator `()`; the latter has two arguments and behaves like a Boolean function; it compares the arguments `b1` and `b2` and returns `true` if and only if `b1`$\preceq$`b2`$\wedge$`b1`$\neq$`b2`. Implementing the comparison oracle as a class makes it possible to draw on further information for comparison purposes (here the variable `_today`), which is sometimes necessary.

Program 8.5 (Sorting by Successive Selection)

```cpp
// sort.cpp (Sorting by Successive Selection)

#include <iostream>
#include <string>
#include <vector>
#include <ctime>
#include <random>
#include <iomanip>

template <class Iterator, class Compare>
void sort1(Iterator first, Iterator last, const Compare & comp)
// Iterator must have operators *, ++, and !=
{
    for (Iterator current = first; current != last; ++current) {
        Iterator cur_min = current;
        for (Iterator i = current; i != last; ++i) {
            if (comp(*i, *cur_min)) {
                cur_min = i;
            }
        }
        std::swap(*cur_min, *current);
    }
}


struct Date {
    Date()                                    // random constructor
    {
        time_t rdate = distribution(generator);
        _time = *localtime(&rdate);
    }

    Date(time_t date): _time (*localtime(&date)) {}     // constructor

    static std::uniform_int_distribution<time_t> distribution;
    static std::default_random_engine generator;
    static time_t today;
    tm _time;
};

time_t Date::today = time(nullptr);
std::uniform_int_distribution<time_t> Date::distribution (1, Date::today);
std::default_random_engine Date::generator (Date::today);


std::ostream & operator<<(std::ostream & os, const Date & date)
{
    static const std::string monthname[12] =  {"Jan", "Feb", "Mar",
        "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
    os << monthname[date._time.tm_mon] << " " << std::setw(2)
       << date._time.tm_mday << ", " << date._time.tm_year + 1900;
    return os;
}


class BirthdayComparison {
public:
    BirthdayComparison(const Date & today) : _today(today) {}
    bool operator()(const Date & b1, const Date & b2) const
    {
        return day_num (b1) < day_num (b2);
    }
private:
    int day_num(const Date & date) const
    {
        return (32 * (12 + date._time.tm_mon - _today._time.tm_mon) +
                date._time.tm_mday  - _today._time.tm_mday) % (12*32);
    }
    Date const & _today;
};


int main()
{
    std::cout << "Today is " << Date(Date::today) << ".\n"
              << "How many random birthdays do you want to sort? ";
    int n;
    std::cin >> n;
    std::vector<Date> dates(n);
    std::cout << "Here are " << n << " random dates:\n";
    for (auto d: dates) {
        std::cout << d << "    ";
    }
    std::cout << "\n\n";

    BirthdayComparison comparison(Date(Date::today));
    std::cout << "Sorting..." << std::endl;
    clock_t timer = clock();
    sort1(dates.begin(), dates.end(), comparison);
    timer = clock() - timer;

    std::cout << "The upcoming birthdays are, starting today:\n";
    for (auto d: dates) {
        std::cout << d << "    ";
    }
    std::cout << "\n\n" << "Sorting took "
              << static_cast<double>(timer)/CLOCKS_PER_SEC << " seconds.\n";
}
```

Had we supplemented our class `Queue` with another iterator (a subclass with a variable of type `Item*`, a constructor with argument of type `Item*`, operators `++`, `!=` and `*`) as well as functions `begin()` and `end()` that return an iterator containing `_front` and `nullptr` respectively, then one could also sort a queue (filled analogously with random data, for example) of type `Queue<Date>` with exactly the same function `sort1`.

C++ in Detail (8.1): Random Numbers and Clock Time
In C++ there are several ways of generating pseudo-random numbers. The simplest is to use the function `rand()` which is available after inserting `#include <cstdlib>`; cf. Program 8.20. This function returns an integer between 0 and `RAND_MAX`, where `RAND_MAX` is a predefined numerical constant not less than 32767. The pseudo-random numbers generated by `rand()` can, however, diverge widely from being uniformly distributed.
A somewhat more laborious way which, moreover, permits more influence on the generated random numbers and generates pseudo-random numbers of better quality, is available after inserting `#include <random>`. With

(continued)

`std::default_random_engine` one can define a random number generator which generates random numbers having a chosen distribution. When defining the random number generator one has the option of entering a certain argument used for initializing the generator. An instance of this can be found on line 43 of Program 8.5. For the distribution we have chosen `std::uniform_int_distribution` in line 42. In this way, uniformly distributed integers lying in the given interval can be generated. As template parameter one must enter an integer type; we have used the type `time_t` which we will explain below. Line 29 of Program 8.5 demonstrates how a single random number is generated.
After inserting `#include <ctime>`, several useful functions are available for finding the current clock time and computing time differences. Calling `time(nullptr)` returns the number of seconds that have elapsed since 1. 1. 1970. The result type is `time_t`. The function `localtime` (cf. line 30 of Program 8.5, for example) converts a value of type `time_t` into a value of type `tm`. Here `tm` is a class which contains, among others, the values `tm_year`, `tm_mon` and `tm_mday`, yielding the year, month and day of a certain date; see, for example, lines 50 and 51 of Program 8.5.
In order to determine the exact runtime of a program, a result given in seconds is not usually sufficient. The function `clock()` returns the CPU-time used by a program as a value of type `clock_t` to the nearest millisecond at least. Changing to seconds is done by division using `CLOCKS_PER_SEC`, see line 97 of Program 8.5.

Usually an iterator also allows the operators `--` (here a list has to be doubly linked) and `==`. Sometimes (e.g. for `vector`, but not for lists) random access is also allowed, i.e. one can, for example, apply the operation `+=i` with an arbitrary `int i` to iterators (taking care that one does not leave the given domain). For some algorithms, random access is indispensable. If necessary, one can, of course, first copy the data files into a `vector` and then copy them back again after sorting.

After inserting `#include <algorithm>`, the C++ Standard Library provides the function `std::sort`. The interface of this function has the same structure as our `sort1`. (One can, with `std::sort`, also omit the third argument; then comparison takes place only with <. Note, however, that `std::sort` requires random access.)

We have now become acquainted with an essential feature of the C++ Standard Library: the separation of data structures and algorithms. Algorithms like the sorting functions `std::sort` and `sort1` run to a large extent independently of the data structure in which the objects are stored.

Program 8.5 also demonstrates the use of pseudo-random numbers and clock times; see Box **C++ in Detail (8.1)**. For its output Program 8.5 uses the stream manipulator `setw`, which is defined in `iomanip` and is used for fixing the minimum number of places to be allotted to the very next output.

We will now show that the running time of Algorithm 8.3 is best possible for the General Sorting Problem. For $0 \le k \le n$ let $\binom{n}{k} := \frac{n!}{k!(n-k)!}$ (with $0! := 1$). For a finite set S and $0 \le k \le |S|$ let $\binom{S}{k} := \{A \subseteq S : |A| = k\}$. Note that $|\binom{S}{k}| = \binom{|S|}{k}$.

Theorem 8.6 *For every algorithm for Problem 8.2 and every $n \in \mathbb{N}$ there is an input $(S, \preceq)$ with $|S| = n$, for which the algorithm requires at least $\binom{n}{2}$ oracle calls.*

Proof For $S = \{1, \dots, n\}$ and $(a, b) \in S \times S$ with $a \neq b$ consider $R := \{(a, b)\} \cup \{(x, y) \in S \times S : x = y\}$. Then the algorithm must execute an oracle call for (a, b) or for (b, a), otherwise it will not know in which order to place a and b. As (a, b) is not known, at least one oracle call is necessary for each of the $\binom{n}{2}$ pairs in $\binom{S}{2}$. □

This is in fact the exact number of oracle calls required by Algorithm 8.3 if j is made to begin with $i + 1$ and not with i, which makes no difference, as was mentioned earlier. Then the algorithm is, in this sense, best possible.

8.3 Sorting by Keys

In order to obtain faster sorting methods, one thus has to involve additional properties. In most applications the partial order can be described by means of so-called keys: for every element s of the set S to be sorted, there is a key $k(s) \in K$, where K is a set with a total order $\le$ (often $\mathbb{N}$ or $\mathbb{R}$ with their usual ordering). The partial order $\preceq$ of S is then given by $a \preceq b \Leftrightarrow (a = b \vee k(a) < k(b))$. Such a partial order is said to be induced by keys. An example is given by the birthdays in Program 8.5; here `day_num` is such a function k.

We thus have the following problem:

Computational Problem 8.7 (Sorting by Keys)

Input: a finite set S, a function $k: S \to K$ and a total order $\le$ on K.
Task: compute a bijection $f: \{1, \dots, n\} \to S$ with $k(f(i)) \le k(f(j))$ for all $1 \le i < j \le n$.

Let $K = \{1, \dots, m\}$ with its natural ordering. If the keys $k(s) \in K$ are explicitly known for all $s \in S$, then the sorting problem can be solved in running time (and with a storage demand of) $O(|S| + m)$ by generating and then joining lists $k^{-1}(i)$ for $i = 1, \dots, m$ (Bucket Sort). This is best possible for $m = O(|S|)$.

We will, however, make no more assumptions about $(K, \le)$ in what follows below. We will again assume that we can only obtain information about the keys from an oracle: for $a, b \in S$ we can refer to the oracle as to whether or not $k(a) < k(b)$ (or equivalently: $a \preceq b$ and $a \neq b$).

Of course, Algorithm 8.3 also works for sorting by keys. Further examples of simple methods for this problem are:

- Insertion Sort: proceeding successively for $i = 1, \dots, n$, one sorts the first i elements by placing the i-th element in a correct place in the sorted list of the first $i - 1$ elements.
- Bubble Sort: one checks the list up to $(n - 1)$ times, each time interchanging two neighboring elements if they are not in the right order; after i checks the i greatest elements are in the right order at the end of the list.

These methods are sometimes faster, but the following also holds for both: for every n there are instances requiring $\binom{n}{2}$ comparisons (where n is the number of elements). In the next section we will present a better algorithm.

8.4 Merge Sort

We will now study sorting algorithms with $O(n \log n)$ running time. The algorithm Merge Sort is based on the "divide and conquer" principle: divide the original problem into a number of smaller subproblems, solve these separately (usually recursively), and merge the solutions of the subproblems to form a solution of the original problem.

Algorithm 8.8 (Merge Sort)

Input: a set $S = \{s_1, \dots, s_n\}$; a partial order $\preceq$ on S induced by keys (given by an oracle).
Output: a bijection $f\colon \{1, \dots, n\} \to S$ with $f(j) \npreceq f(i)$ for all $1 \le i < j \le n$.

```
if $|S| > 1$
    then let $S = S_1 \dot\cup S_2$ with $|S_1| = \lfloor \frac{n}{2} \rfloor$ and $|S_2| = \lceil \frac{n}{2} \rceil$
         sort $S_1$ and $S_2$ (recursively with Merge Sort)
         merge the sortings of $S_1$ and $S_2$ to a sorting of $S$ .
```

The partitioning of S into $S_1 \dot\cup S_2$ is clearly easy to achieve. After that, Merge Sort is called recursively for S_1 and S_2. Merging the sortings can easily be done in $O(|S|)$ time: one checks the sorted lists simultaneously, each time comparing the first ("least") elements of the two lists, deletes the lesser one from its list and places it at the end of a new list.

One can then copy the elements of the new list at the end into the memory places of the old list and delete the new list. If the pointers of a list are known (and not just its iterators), then one can dispense altogether with the formation of a new list and do the merging just by reassigning the pointers. For this reason Merge Sort is particularly useful for sorting lists (random access is not needed).

Theorem 8.9 *Merge Sort has running time $O(n \log n)$.*

Proof Let $T(n)$ denote the runtime of Merge Sort for n elements. Clearly there exists a $c \in \mathbb{N}$ with $T(1) \leq c$ and

$$T(n) \ \leq \ T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + cn \qquad \text{for all } n \geq 2\,.$$

Claim: $T(2^k) \leq c(k+1)2^k$ for all $k \in \mathbb{N} \cup \{0\}$.

The proof of the claim is by induction over k. For $k = 0$ it is trivial. For $k \in \mathbb{N}$ the induction hypothesis implies that

$$T(2^k) \ \leq \ 2 \cdot T(2^{k-1}) + c\,2^k \ \leq \ 2 \cdot c\,k\,2^{k-1} + c\,2^k \ = \ c(k+1)2^k\,.$$

This proves the claim and we have for all $n \in \mathbb{N}$ with $k := \lceil \log_2 n \rceil < 1 + \log_2 n$:

$$T(n) \ \leq \ T(2^k) \ \leq \ c(k+1)2^k \ < \ 2c(2 + \log_2 n)n \ = \ O(n \log n). \qquad \square$$

Merge Sort was already available as hardware in 1938, and in 1945 John von Neumann implemented it in software as one of the very first algorithms [22]. Merge Sort is also in some sense best possible, as is made precise in the following theorem:

Theorem 8.10 *Even when the inputs are restricted to totally ordered sets, every sorting algorithm obtaining information on orderings solely by the pairwise comparison of two elements, requires at least* $\log_2(n!)$ *comparisons for sorting an n-element set.*

Proof We consider only totally ordered input sets $S = \{1, \ldots, n\}$ as input and seek one of the $n!$ permutations on S as output. A comparison "$a < b$?" of two elements a and b returns "true" for some set of permutations and "false" for another set. The larger of these two sets contains at least $\frac{n!}{2}$ permutations. Every further comparison at most halves the larger of these two sets. Thus, after k comparisons at least $\frac{n!}{2^k}$ permutations remain that cannot be differentiated by the algorithm. Hence every sorting algorithm requires at least k comparisons, where $2^k \geq n!$, i.e. $k \geq \log_2(n!)$. $\square$

Remark 8.11 We have $n! \geq \lfloor \frac{n}{2} \rfloor^{\lceil n/2 \rceil}$, so $\log_2(n!) \geq \lceil n/2 \rceil \log \lfloor \frac{n}{2} \rfloor = \Omega(n \log(n))$. Hence Merge Sort is an asymptotically optimal sorting method. In general, however, Merge Sort requires more than $\lceil \log_2(n!) \rceil$ comparisons. One can prove that no sorting algorithm can always manage with $\lceil \log_2(n!) \rceil$ comparisons. There is, for example, no sorting algorithm that can sort 12 elements with only $29 = \lceil \log_2(12!) \rceil$ comparisons. For every algorithm there are instances with 12 elements for which it requires at least 30 comparisons [22]. For sorting a 12 element set, Merge Sort requires at least 20 and at most 33 comparisons.

8.5 Quick Sort

Quick Sort, suggested in 1960 by Antony Hoare, is also based on the "divide and conquer" principle but dispenses with the effort of the merge-step required by Merge Sort:

Algorithm 8.12 (Quick Sort)

Input: a set $S = \{s_1, \dots, s_n\}$; a partial order $\preceq$ on S induced by keys (given by an oracle).
Output: a bijection $f : \{1, \dots, n\} \to S$ with $f(j) \not\preceq f(i)$ for all $1 \le i < j \le n$.

```
if $|S| > 1$
    then choose any $x \in S$
        $S_1 := \{s \in S : s \preceq x\} \setminus \{x\}$
        $S_2 := \{s \in S : x \preceq s\} \setminus \{x\}$
        $S_x := S \setminus (S_1 \cup S_2)$
        sort $S_1$ and $S_2$ (if not empty) recursively with Quick Sort
        merge the sorted sets $S_1, S_x$ and $S_2$
```

An advantage of Quick Sort over Merge Sort when sorting data in an array (or `vector`) is that no additional temporary memory is required. On the other hand, we are only able to prove the following worse running time bound:

Theorem 8.13 *Algorithm 8.12 works correctly and has running time $O(n^2)$.*

Proof The correctness follows immediately from the fact that $S_1 = \{s \in S : k(s) < k(x)\}$ and $S_2 = \{s \in S : k(s) > k(x)\}$, where $k(s)$ again denotes the key of s.

Let $T(n)$ be the runtime of Quick Sort for n elements. Clearly there exists a $c \in \mathbb{N}$ with $T(1) \le c$ and

$$T(n) \le c\,n + \max_{0 \le l \le n-1} (T(l) + T(n-1-l)) \qquad \text{for all } n \ge 2\,,$$

where l is the cardinality of S_1 and we set $T(0) := 0$.

Claim: $T(n) \le c\,n^2$.

The proof of the claim follows by induction over n. For $n = 1$ it is trivial. For the induction step we calculate:

$$\begin{aligned} T(n) &\le c\,n + \max_{0 \le l \le n-1} (T(l) + T(n-1-l)) \\ &\le c\,n + \max_{0 \le l \le n-1} (c\,l^2 + c\,(n-1-l)^2) \\ &= c\,n + c\,(n-1)^2 \\ &< c\,n^2. \end{aligned}$$

□

Remark 8.14 The statement of the theorem is best possible because the running time is $\Omega(n^2)$ when the set is already sorted and x is always taken to be the first element.

Of course, one can also choose x differently, e.g. randomly. Ideally, x is a median of S, i.e. $|S_1| \le \frac{|S|}{2}$ and $|S_2| \le \frac{|S|}{2}$. One can determine a median of an n-element set in $O(n)$ time, but the effort involved is not so little as to pay off in every case, although the running time would then be reduced to $O(n \log n)$.

The main reason for the interest in the Quick Sort algorithm (with a simple choice of x) is its usually excellent running time in practice. We can give a partial theoretical explanation of this by considering its expected running time if all choices of x in step 2 are made independently of one another and always with the same probability. We then have a randomized algorithm which we will call Random Quick Sort. We will denote the expected value of its running time by $\overline{T}(n)$.

Theorem 8.15 *The expected value $\overline{T}(n)$ of the running time of Random Quick Sort with n elements is $O(n \log n)$.*

Proof There exists a $c \in \mathbb{N}$ with $\overline{T}(1) \le c$ and

$$\overline{T}(n) \;\le\; c\,n + \frac{1}{n}\sum_{i=1}^{n}\left(\overline{T}(i-1) + \overline{T}(n-i)\right) \;=\; c\,n + \frac{2}{n}\sum_{i=1}^{n-1}\overline{T}(i)\,,$$

for $n \ge 2$ and with $\overline{T}(0) := 0$, because for each $i = 1, \dots, n$ we choose, with probability $\frac{1}{n}$, the element in the i-th place of a fixed sorting as our x.

Claim: $\overline{T}(n) < 2cn(1 + \ln n)$.

The proof of the claim follows by induction over n. For $n = 1$ it is trivial. For $n \ge 2$ we have by the induction hypothesis:

$$\begin{aligned}
\overline{T}(n) &\le c\,n + \frac{2}{n}\sum_{i=1}^{n-1} 2ci(1 + \ln i) \\
&\le c\,n + \frac{2c}{n}\int_1^n 2x(1 + \ln x)\,\mathrm{d}x \\
&= c\,n + \frac{2c}{n}\left[x^2(1 + \ln x) - \frac{x^2}{2}\right]_1^n \\
&= c\,n + 2cn(1 + \ln n) - cn - \frac{c}{n} \\
&< 2cn(1 + \ln n)
\end{aligned}$$

□

8.6 Binary Heaps and Heap Sort

A **priority queue** is a data structure for storing elements with keys, which provides at least the following functions:

- `init`: initializing an empty queue (constructor);
- `insert` $(s, k(s))$: inserting the element s with key $k(s)$ in the queue;
- $s =$ `extract_min`: removing an element s with minimum key and returning s;
- `clear`: deleting a queue (destructor).

If one uses an array or a list as data structure, then `insert` needs $O(1)$ running time and `extract_min` needs $O(n)$ running time if n elements are in the priority queue. If one uses a sorted array or a sorted list, then `insert` needs $O(n)$ and `extract_min` needs $O(1)$ running time. Binary heaps allow one to execute both `insert` and `extract_min` in $O(\log n)$ running time.

Sometimes additional operations are required, in particular:

- `decrease_key` $(s, k(s))$: decreasing the key $k(s)$ of s;
- $s =$ `find_min`: returning an element s with minimum key (without removing it);
- `remove` (s): removing the element s from the queue.

Definition 8.16 A **heap** for a finite set S with keys $k: S \to K$, where K has a total order $\leq$, is an arborescence A with a bijection $f: V(A) \to S$ such that the heap order

$$k(f(v)) \leq k(f(w)) \qquad \textit{for all } (v, w) \in E(A) \tag{8.1}$$

holds.

An example is depicted in Fig. 8.1. The function `find_min` simply returns $f(r)$, where r is the root of the arborescence. In order to be able to implement the other functions efficiently, we restrict ourselves to certain very easily representable arborescences as in Fig. 8.1 for $n = 10$:

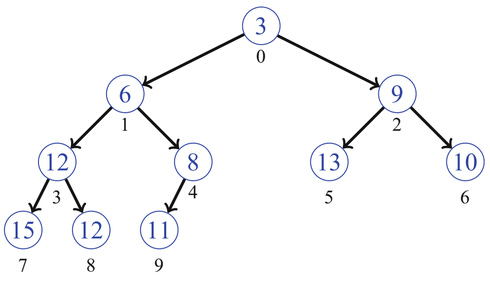


Fig. 8.1 A binary heap for a set of ten elements with keys (*blue*) 3, 6, 8, 9, 10, 11, 12, 12, 13, 15

Proposition 8.17 *Let $n \in \mathbb{N}$. Then the graph B_n with $V(B_n) = \{0, \ldots, n-1\}$ and $E(B_n) = \{(i,j) : i \in V(B_n), j \in \{2i+1, 2i+2\} \cap V(B_n)\}$ is an arborescence with root 0. There is no path in B_n longer than $\lfloor \log_2 n \rfloor$.*

Proof Clearly B_n is acyclic with $\delta^-(j) = \{(\lfloor \frac{j-1}{2} \rfloor, j)\}$ for $j \in \{1, \ldots, n-1\}$ and is therefore an arborescence with root 0. For every path with vertices $v_0, \ldots, v_k$ we have $v_{i+1} + 1 \geq 2(v_i + 1)$, hence $n \geq v_k + 1 \geq 2^k$. □

Arborescences of this type are also known as complete binary trees. A heap (B,f) for S is called a **binary heap** if $B = B_{|S|}$.

Program 8.18 presents an implementation with all the above-mentioned functions. When inserting an element in an n-element heap, it is initially assigned to vertex n and then this vertex is interchanged with its predecessor as long as the new element has a lesser key (`sift_up`). When removing an element from an n-element heap, one initially copies the element in vertex $n-1$ into the vacated vertex and then restores the heap order using `sift_up` or `sift_down`; see Fig. 8.2.

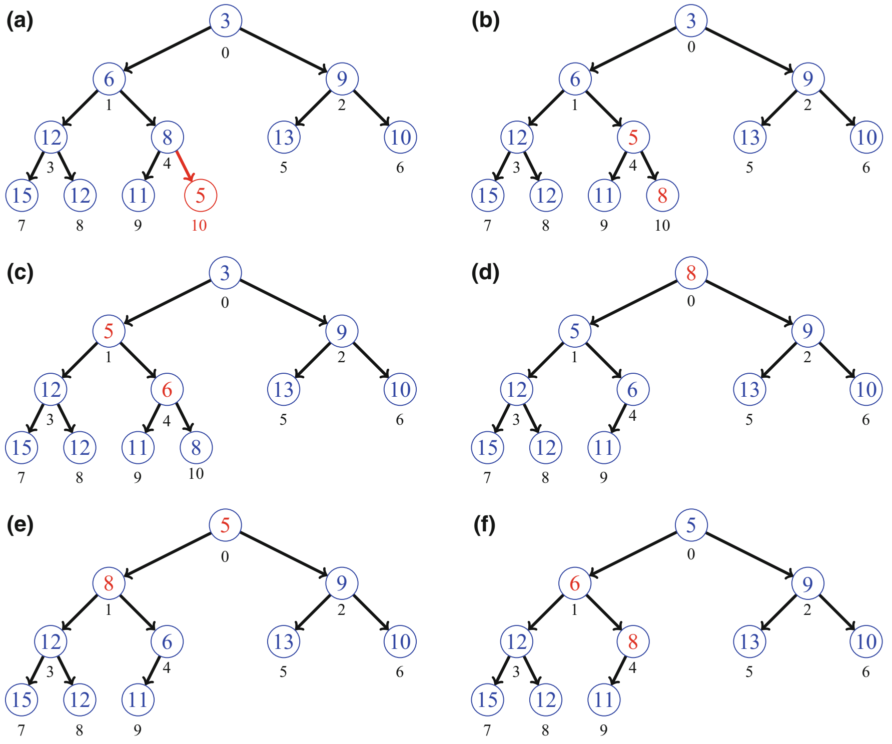


Fig. 8.2 An element with key 5 is inserted into the heap depicted in Fig. 8.1: (**a**) for a start it is written in vertex 10; (**b**) then the contents of vertices 4 and 10 are interchanged; (**c**) next, the contents of vertices 1 and 4 are interchanged, which restores the heap structure. If we now remove the element in the root, the content of vertex 10 is first written in the root (**d**) and then (**e**) and (**f**) passed downwards in two steps

Program 8.18 (Heap)

```cpp
// heap.h (Binary Heap)

#include <vector>
#include <stdexcept>

template <typename T>     // assume that T has the < operator
class Heap {
public:
    bool is_empty() const
    {
        return _data.size() == 0;
    }

    const T & find_min() const
    {
        if (is_empty()) {
            throw std::runtime_error("Empty heap; Heap::find_min failed.");
        }
        return _data[0];
    }

    T extract_min()
    {
        T result = find_min();
        remove(0);
        return result;
    }

    int insert(const T & object)
    {
        _data.push_back(object);
        sift_up(_data.size() - 1);
        return _data.size() - 1;
    }

protected:                           // accessible only for derived classes
    void remove(int index)
    {
        ensure_is_valid_index(index);
        swap(_data[index], _data[_data.size() - 1]);
        _data.pop_back();
        sift_up(index);
        sift_down(index);
    }

    void decrease_key(int index)
    {
        ensure_is_valid_index(index);
        sift_up(index);
    }

    virtual void swap(T & a, T & b)                 // virtual functions can be
    {                                               // overloaded by derived classes
        std::swap(a, b);
    }

    T & get_object(int index)
    {
        ensure_is_valid_index(index);
        return _data[index];
    }

private:
    void  ensure_is_valid_index(int index)
    {
        if (index >= static_cast<int>(_data.size()) or index < 0)
            throw std::runtime_error("Index error in heap operation");
    }

    static int parent(int index)          // do not call with index==0!
    {
        return (index - 1) / 2;
    }

    static int left(int index)            // left child may not exist!
    {
        return (2 * index) + 1;
    }

    static int right(int index)           // right child may not exist!
    {
        return (2 * index) + 2;
    }

    void sift_up(int index)
    {
        while ((index > 0) and (_data[index] < _data[parent(index)])) {
            swap(_data[index], _data[parent(index)]);
            index = parent(index);
        }
    }

    void sift_down(int index)
    {
        int smallest = index;
        while (true) {
            if ((left(index) < static_cast<int>(_data.size())) and
                (_data[left(index)] < _data[smallest]))
            {
                smallest = left(index);
            }
            if ((right(index) < static_cast<int>(_data.size())) and
                (_data[right(index)] < _data[smallest]))
            {
                smallest = right(index);
            }
            if (index == smallest) return;
            swap(_data[smallest], _data[index]);
            index = smallest;
        }
    }

    std::vector<T> _data;       // holds the objects in heap order
};
```

Theorem 8.19 *The functions of the class* `Heap` *in Program 8.18 are correct. The running time of each function is bounded by* $O(\log n)$*, where* n *is the number of elements in the heap.*

Proof The running time follows immediately with Proposition 8.17. To show the correctness we need to prove the following

Claim: Let (B_n, f) be a binary tree with the heap property for (S, k). If $k(f(i))$ is decreased or increased for some $i \in \{0, 1 \ldots, n-1\}$, then the heap property is restored by `sift_up(i)` or `sift_down(i)` respectively.

If $k(f(i))$ is decreased, then the heap property is only violated (if at all) for the edge in $\delta^-(i)$. This, however, gets restored by `sift_up`, where in each iteration the heap property is only violated (if at all) for the edge `(parent(index),index)`.

Analogously, if $k(f(i))$ is increased, then the heap property is only violated (if at all) for the edges in $\delta^+(i)$ and gets restored by `sift_down`. □

The functions `remove` and `decrease_key` can only be implemented meaningfully if one knows the number of the vertex in the heap where the relevant object is stored. As a rule, however, this is not the case because it keeps being changed by `swap`. For this reason we will shortly present a new class derived from `Heap`, which takes note of the vertex numbers and interchanges them whenever `swap` is used. This is, however, not necessary for the cardinal functions `insert` and `extract_min`.

With the help of binary heaps we obtain another algorithm for sorting by keys, with running time $O(n \log n)$. As the example dealt with in Program 8.20 shows, the objects are first sorted into a heap and then successively extracted again with `extract_min`. This algorithm is called **Heap Sort**.

Program 8.20 (Heap Sort)

```
// heapsort.cpp (Example for Heapsort)

#include <iostream>
#include <cstdlib>
#include "heap.h"

int main()
{
    int n;
    std::cout << "How many numbers do you want to sort? ";
    std::cin >> n;

    Heap<int> heap;
    int new_number;

    std::cout << "I will sort the following " << n << " numbers:\n";
    for (int i = 0; i < n; ++i) {
        new_number = rand() % 900 + 100;
        std::cout << new_number << " ";
        heap.insert(new_number);
    }

    std::cout << "\n" << "Sorted:\n";
    while (not(heap.is_empty())) {
        std::cout << heap.extract_min() << " ";
    }
    std::cout << "\n";
}
```

Theorem 8.21 *Heap Sort sorts n elements in running time $O(n \log n)$.*

Proof Follows from Theorem 8.19. □

8.7 Further Data Structures

There are also more powerful data structures enabling one in addition to find an element with a particular key in $O(\log n)$ time (with which the above-mentioned indexing becomes superfluous), to perform binary search, as well as to find the predecessor and successor of every element with respect to an ordering corresponding to the keys. Such balanced search trees (commonly called search tree, AVL tree, red-black tree) have, like heaps, also been implemented in the C++ Standard Library, where they are called `map`.

Hash tables (in C++ `unordered_map`) comprise a further useful type of data structure. For storing a set $S \subseteq U$ one here chooses a function $g: U \to \{0, \dots, k-1\}$ with the property that mostly only few elements have the same function value. For every $i = 0, \dots, k-1$ the (mostly only few) elements of $g^{-1}(i)$ are stored in another data structure, e.g. a list. When searching for an element $u \in U$ one then only has to compute $g(u)$ and scan a short list.

The books [33] and [7] contain further information on data structures.

9 Optimal Trees and Paths

In combinatorial optimization one aims to find an optimal element in a finite set of objects with a certain combinatorial structure. The objects (i.e. the feasible solutions) can usually be represented by subsets of a finite underlying set U. Often U is the edge set of a graph. In this case the objects can, for example, be s-t-paths (for given vertices s and t) or spanning trees; we will deal with these two cases below. If a weight function $c: U \to \mathbb{R}$ is given, then a feasible solution $X \subseteq U$ is called optimal if its weight (also called its cost) $c(X) := \sum_{u \in X} c(u)$ attains the minimum value over all feasible solutions.

The class `Graph` (Program 6.29) presented in Chap. 6 is designed also for weighted graphs. The function `add_edge` has an optional third argument which can be used to represent the edge weight.

Let G be a graph with edge weights $c: E(G) \to \mathbb{R}$. For any subgraph H of G we define the weight of H to be $c(E(H)) = \sum_{e \in E(H)} c(e)$; this is sometimes also called the cost or length of H.

9.1 Optimal Spanning Trees

We will begin by considering the following combinatorial optimization problem.

Computational Problem 9.1 (Minimum Spanning Tree Problem)

Input: a connected undirected graph G with edge weights $c: E(G) \to \mathbb{R}$.
Task: find a minimum weight spanning tree in G.

An optimal solution is often called a minimum spanning tree or MST for short. The following simple algorithm due to Kruskal [24] always yields an optimal solution. It is also known as the Greedy Algorithm because it acts “greedily” by always grabbing the cheapest edge.

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_9

Algorithm 9.2 (Kruskal's Algorithm)

Input: a connected undirected graph G with edge weights $c\colon E(G) \to \mathbb{R}$.
Output: a minimum weight spanning tree $(V(G), T)$ in G.

```
Sort $E(G) = \{e_1, \ldots, e_m\}$ such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$
$T \leftarrow \emptyset$
for $i \leftarrow 1$ to $m$ do
    if $(V(G), T \cup \{e_i\})$ is a forest then $T \leftarrow T \cup \{e_i\}$
```

Theorem 9.3 *Kruskal's Algorithm 9.2 is correct.*

Proof The graph $(V(G), T)$ constructed by Kruskal's Algorithm is clearly a spanning tree. Suppose that $(V(G), T)$ is not connected. Then by Theorem 6.13 there would exist an $\emptyset \neq X \subset V(G)$ with $\delta(X) \cap T = \emptyset$. As G is connected, this would imply that G contains an edge $e \in \delta(X)$. But then this edge would have been added to T when its turn came.

The output $(V(G), T)$ of Kruskal's Algorithm is thus always a spanning tree. We will now prove its optimality. Let $(V(G), T^*)$ be a minimum weight spanning tree which, moreover, is chosen to have the property that $|T^* \cap T|$ has maximum value.

Assume that $T^* \neq T$ (otherwise T is optimal). As $|T| = |T^*| = |V(G)| - 1$ (by Theorem 6.18), $T^* \setminus T \neq \emptyset$. Let $j \in \{1, \ldots, m\}$ be the least index satisfying $e_j \in T^* \setminus T$.

As Kruskal's Algorithm did not pick e_j, there must exist a circuit C with $E(C) \subseteq \{e_j\} \cup (T \cap \{e_1, \ldots, e_{j-1}\})$. Clearly $e_j \in E(C)$.

$(V(G), T^* \setminus \{e_j\})$ is not connected, i.e. there exists an $X \subset V(G)$ with $\delta_G(X) \cap T^* = \{e_j\}$. Now $|E(C) \cap \delta_G(X)|$ is an even number (which is true for every circuit and every X), so it is at least two. Let $e_i \in (E(C) \cap \delta_G(X)) \setminus \{e_j\}$. Note that $i < j$ and thus that $c(e_i) \le c(e_j)$.

Let $T^{**} := (T^* \setminus \{e_j\}) \cup \{e_i\}$. Then $(V(G), T^{**})$ is a spanning tree with $c(T^{**}) = c(T^*) - c(e_j) + c(e_i) \le c(T^*)$, hence it is optimal. But T^{**} has one more edge (namely e_i) in common with T than T^* does, which contradicts the choice of T^*. □

For most combinatorial optimization problems the Greedy Algorithm does not in general yield an optimal solution; on the contrary, its output can be very poor. For the Minimum Spanning Tree Problem, however, it works very well. It is also quite fast if carefully implemented:

Theorem 9.4 *Kruskal's Algorithm 9.2 can be implemented to have a running time of $O(m \log n)$, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof We can assume that $m \leq n^2$ because otherwise we can remove all parallel edges beforehand (keeping only the cheapest for every vertex pair); this can be done with bucket sort in $O(m + n^2)$ time.

Then sorting the m edges can be done in $O(m \log m) = O(m \log n)$ time, as described in Chap. 8.

The **for** loop is traversed m times. In order to facilitate testing whether $(V(G), T \cup \{e_i\})$ is a forest, we note the connected components of $(V(G), T)$ as follows. To each connected component we allot an index and we also note the number of its vertices. In addition, for each vertex we store the index of the connected component in which it lies. Then one can test in $O(1)$ time whether $(V(G), T \cup \{e_i\})$ is a forest, as we only need to test whether or not the endpoints of e_i lie in different connected components. If one adds e_i, then two connected components must be combined into one. One does this by adding the numbers of vertices in the two components and then, before the addition of e_i, using Algorithm 7.1 to traverse the vertices of the smaller component and assign them the index of the larger component. As a vertex can change components at most $\lfloor \log_2 n \rfloor$ times (because each time the size of the component doubles at least), the total running time of the **for** loop is $O(m \log n)$.

□

We will now present a second proof of the correctness. We call a vertex set F for an instance (G, c) of the MST Problem *good* if there exists a minimum weight spanning tree $(V(G), T)$ with $F \subseteq T$. Then we have:

Lemma 9.5 *Let (G, c) be an instance of the MST Problem, $F \subset E(G)$ a good set and $e \in E(G) \setminus F$. Then the set $F \cup \{e\}$ is good if and only if there exists an $X \subset V(G)$ with $\delta_G(X) \cap F = \emptyset$, $e \in \delta_G(X)$, and $c(e) \leq c(f)$ for all $f \in \delta_G(X)$.*

Proof

"$\Rightarrow$": Assume that $F \cup \{e\}$ is good and let $(V(G), T)$ be a minimum weight spanning tree with $F \cup \{e\} \subseteq T$. Then $(V(G), T \setminus \{e\})$ is not connected; so let X be the vertex set of a connected component, i.e. $\delta_G(X) \cap T = \{e\}$. If there exists an $f \in \delta_G(X)$ with $c(f) < c(e)$, then $(V(G), (T \setminus \{e\}) \cup \{f\})$ is a spanning tree with lower weight, contradicting optimality.

"$\Leftarrow$": Assume conversely that $X \subset V(G)$ with $\delta_G(X) \cap F = \emptyset$, $e \in \delta_G(X)$, and $c(e) \leq c(f)$ for all $f \in \delta_G(X)$. As F is good, there exists a minimum weight spanning tree $(V(G), T)$ with $F \subseteq T$. If $e \in T$, then $F \cup \{e\}$ is good and the proof is complete. If, on the other hand, $e \notin T$, then $(V(G), T \cup \{e\})$ contains a circuit containing e as well as another edge $f \in \delta_G(X)$. Let $T' := (T \setminus \{f\}) \cup \{e\}$. Then $(V(G), T')$ is a spanning tree which is also of minimum weight because $c(T') = c(T) - c(f) + c(e) \leq c(T)$. Moreover, $F \cup \{e\} \subseteq T'$ (as $f \notin F$), hence $F \cup \{e\}$ is good.

□

The above lemma immediately implies the correctness of Kruskal's Algorithm, because T is good at every point in time of the algorithm.

This lemma also yields another way of solving the MST Problem. We now present a further well-known algorithm, which was already found in 1930 by Jarník [20] but today bears the name of Prim [29] who rediscovered it.

Algorithm 9.6 (Prim's Algorithm)

Input: a connected undirected graph G with edge weights $c\colon E(G) \to \mathbb{R}$.
Output: a minimum weight spanning tree $(V(G), T)$ in G.

```
Choose an arbitrary $v \in V(G)$
$X \leftarrow \{v\}$
$T \leftarrow \emptyset$
while $X \neq V(G)$ do
    choose an $e = \{x, y\} \in \delta_G(X)$ with minimum weight; let $x \in X$ and $y \notin X$
    $T \leftarrow T \cup \{e\}$
    $X \leftarrow X \cup \{y\}$
```

Theorem 9.7 *Prim's Algorithm 9.6 works correctly. Using binary heaps, it can be implemented in such a way that its running time is $O(m \log n)$, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof The following holds for Prim's Algorithm: at every point in time (X, T) is clearly a tree and by Lemma 9.5 T is good and hence is, at the end, a minimum weight spanning tree.

In order to achieve the stated running time, we will always store vertices $v \in V(G) \setminus X$ with $\delta(v) \cap \delta(X) \neq \emptyset$ in a heap, where v has the key $\min\{c(e) : e \in \delta(v) \cap \delta(X)\}$. The running time is then dominated by n `insert`-, n `extract_min`- and at most m `decrease_key` operations (cf. Program 9.8). As $n \leq m + 1$, the statement follows by Theorem 8.19. □

9.2 Implementing Prim's Algorithm

We will now present an implementation of Prim's Algorithm and at the same time a nearly identical implementation of Dijkstra's Shortest Path Algorithm which we will discuss shortly. We will use not only the class `Graph` (Program 6.29) but also a class derived from `Heap<HeapItem>` (cf. Program 8.18), namely `NodeHeap`, which stores vertices of the graph with keys. Note the additional stored data and the `swap` function which overloads the one in `Heap`. In `NodeHeap` this new `swap` function is also addressed by the inherited functions `sift_up` and `sift_down`; this is possible because the already existing `swap` function in `Heap` has an identical interface and was declared to be `virtual`. Hence the operations `remove` and in particular `decrease_key` can now also be used.

Program 9.8 (Prim's Algorithm and Dijkstra's Algorithm)

```
// primdijkstra.cpp (Prim's Algorithm and Dijkstra's Algorithm)

#include "graph.h"
#include "queue.h"
#include "heap.h"

struct HeapItem
{
    HeapItem(Graph::NodeId nodeid, double key): _nodeid(nodeid), _key(key) {}
    Graph::NodeId _nodeid;
    double _key;
};

bool operator<(const HeapItem & a, const HeapItem & b)
{
    return (a._key < b._key);
}


class NodeHeap : public Heap<HeapItem> {
public:
    NodeHeap(int num_nodes): _heap_node(num_nodes, not_in_heap)
    {   // creates a heap with all nodes having key = inifinte weight
        for(auto i = 0; i < num_nodes; ++i) {
            insert(i, Graph::infinite_weight);
        }
    }

    bool is_member(Graph::NodeId nodeid) const
    {
        ensure_is_valid_nodeid(nodeid);
        return _heap_node[nodeid] != not_in_heap;
    }

    double get_key(Graph::NodeId nodeid)
    {
        return get_object(_heap_node[nodeid])._key;
    }

    Graph::NodeId extract_min()
    {
        Graph::NodeId result = Heap<HeapItem>::extract_min()._nodeid;
        _heap_node[result] = not_in_heap;
        return result;
    }

    void insert(Graph::NodeId nodeid, double key)
    {
          ensure_is_valid_nodeid(nodeid);
          HeapItem item(nodeid, key);
          _heap_node[nodeid] = Heap<HeapItem>::insert(item);
    }

    void decrease_key(Graph::NodeId nodeid, double new_key)
    {
        ensure_is_valid_nodeid(nodeid);
        get_object(_heap_node[nodeid])._key = new_key;
        Heap<HeapItem>::decrease_key(_heap_node[nodeid]);
    }

    void remove(Graph::NodeId nodeid)
    {
        ensure_is_valid_nodeid(nodeid);
        Heap<HeapItem>::remove(_heap_node[nodeid]);
        _heap_node[nodeid] = not_in_heap;
    }

private:

    void ensure_is_valid_nodeid(Graph::NodeId nodeid) const
    {
        if (nodeid < 0 or nodeid >= static_cast<int>(_heap_node.size()))
            throw std::runtime_error("invalid nodeid in NodeHeap");
    }

    void swap(HeapItem & a, HeapItem & b)
    {
        std::swap(a,b);
        std::swap(_heap_node[a._nodeid],_heap_node[b._nodeid]);
    }

    static const int not_in_heap;
    std::vector<int> _heap_node;
};

int const NodeHeap::not_in_heap = -1;


struct PrevData {
    Graph::NodeId id;
    double weight;
};


Graph mst(const Graph & g)
{   // Prim's Algorithm. Assumes that g is undirected and connected.
    Graph tree(g.num_nodes(), Graph::undirected);
    NodeHeap heap(g.num_nodes());
    std::vector<PrevData> prev(g.num_nodes(), {Graph::invalid_node, 0.0});

    const Graph::NodeId start_nodeid = 0;          // start at vertex 0
    heap.decrease_key(start_nodeid, 0);

    while (not heap.is_empty()) {
        Graph::NodeId nodeid = heap.extract_min();
        if (nodeid != start_nodeid) {
            tree.add_edge(prev[nodeid].id, nodeid, prev[nodeid].weight);
        }
        for (auto neighbor: g.get_node(nodeid).adjacent_nodes()) {
            if (heap.is_member(neighbor.id()) and
                neighbor.edge_weight() < heap.get_key(neighbor.id()))
            {
                prev[neighbor.id()] = {nodeid, neighbor.edge_weight()};
                heap.decrease_key(neighbor.id(), neighbor.edge_weight());
            }
        }
    }
    return tree;
}


Graph shortest_paths_tree(const Graph & g, Graph::NodeId start_nodeid)
{   // Dijkstra's Algorithm. The graph g can be directed or undirected.
    Graph tree(g.num_nodes(), g.dirtype);
    NodeHeap heap(g.num_nodes());
    std::vector<PrevData> prev(g.num_nodes(), {Graph::invalid_node, 0.0});

    heap.decrease_key(start_nodeid, 0);

    while (not heap.is_empty()) {
        double key = heap.find_min()._key;
        if (key == Graph::infinite_weight) {
            break;
        }
        Graph::NodeId nodeid = heap.extract_min();
        if (nodeid != start_nodeid) {
            tree.add_edge(prev[nodeid].id, nodeid, prev[nodeid].weight);
        }
        for (auto neighbor: g.get_node(nodeid).adjacent_nodes()) {
            if (heap.is_member(neighbor.id()) and
                (key + neighbor.edge_weight() < heap.get_key(neighbor.id())))
            {
                prev[neighbor.id()] = {nodeid, neighbor.edge_weight()};
                heap.decrease_key(neighbor.id(), key + neighbor.edge_weight());
            }
        }
    }
    return tree;
}


int main(int argc, char * argv[])
{
    if (argc > 1) {
        Graph g(argv[1], Graph::undirected);
        std::cout << "The following is the undirected input graph:\n";
        g.print();

        std::cout << "\nThe following is a minimum weight spanning tree:\n";
        Graph t = mst(g);
        t.print();

        Graph h(argv[1], Graph::directed);
        std::cout << "\nThe following is the directed input graph:\n";
        h.print();

        std::cout << "\nThe following is a shortest paths tree:\n";
        Graph u = shortest_paths_tree(h, 0);
        u.print();
    }
}
```

9.3 Shortest Paths: Dijkstra's Algorithm

We will now consider another important combinatorial optimization problem: the Shortest Path Problem. For a given graph G with edge weights $c\colon E(G) \to \mathbb{R}$ let

$$\mathrm{dist}_{(G,c)}(x, y) \;:=\; \min\{c(E(P)) : P \text{ is an } x\text{-}y\text{-path in } G\}$$

denote the **distance** from x to y in (G, c). For a given path P we call $c(E(P))$ the length of P (with respect to c).

Computational Problem 9.9 (Shortest Path Problem)

Input: a graph G with edge weights $c\colon E(G) \to \mathbb{R}$ and vertices $s, t \in V(G)$.
Task: compute a shortest s-t-path in (G, c) or decide that t is not reachable from s in G.

One need only consider directed graphs because every undirected edge $e = \{v, w\}$ can be replaced by two directed edges (v, w) and (w, v), both with weight $c(e)$.

If $c(e) = 1$ for all $e \in E(G)$, which amounts to considering only the number of edges (as we were doing earlier), then Theorem 7.6 immediately yields a solution: Breadth-First Search solves this special case of the Shortest Path Problem in linear time. The situation is totally different when the edges have differing weights (costs, lengths).

The problem turns out to be very much more difficult when there are negative edge weights. In many applications, however, this is not the case; e.g. when the edge weights denote traveling times or costs. Thus we will assume for the moment that all the edge weights are non-negative. Then the following famous algorithm due to Dijkstra [9] computes shortest paths from a fixed vertex s to all vertices reachable from s. It is used a great deal in practice.

Algorithm 9.10 (Dijkstra's Algorithm)

Input: a digraph G with edge weights $c: E(G) \to \mathbb{R}_{\geq 0}$, a vertex $s \in V(G)$.
Output: an arborescence $A := (R, T)$ in G such that R contains all vertices reachable from s and $\text{dist}_{(G,c)}(s, v) = \text{dist}_{(A,c)}(s, v)$ holds for all $v \in R$.

```
$R \leftarrow \emptyset, Q \leftarrow \{s\}, l(s) \leftarrow 0$
while $Q \neq \emptyset$ do
    choose a $v \in Q$ with minimum $l(v)$
    $Q \leftarrow Q \setminus \{v\}$
    $R \leftarrow R \cup \{v\}$
    for $e = (v, w) \in \delta^+_G(v)$ with $w \notin R$ do
        if $w \notin Q$ or $l(v) + c(e) < l(w)$ then $l(w) \leftarrow l(v) + c(e), p(w) \leftarrow e$
                                                     $Q \leftarrow Q \cup \{w\}$
$T \leftarrow \{p(v) : v \in R \setminus \{s\}\}$
```

An arborescence that is a correct output is called a shortest-path tree (with root s in (G, c)). More precisely, for a given graph G with edge weights $c: E(G) \to \mathbb{R}$, a **shortest-path tree with root** s is a subgraph H of G such that H is a tree or an arborescence with root s and H contains a shortest s-v-path for all vertices v reachable from s.

A shortest-path tree with root s immediately yields a solution for the Shortest Path Problem and, moreover, it does this for all $t \in V(G)$ at once. An implementation of Dijkstra's Algorithm is included in Program 9.8.

Theorem 9.11 *Dijkstra's Algorithm 9.10 works correctly and has running time $O(n^2 + m)$, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof Let $T := \{p(v) : v \in (R \cup Q) \setminus \{s\}\}$ and $A := (R \cup Q, T)$ at any point in time of Dijkstra's Algorithm. Then the following invariants hold at the end of every iteration of the **while** loop:

(a) $p(w) \in \delta^+_G(R) \cap \delta^-(w)$ for all $w \in Q \setminus \{s\}$;
(b) $A[R]$ is an arborescence with root s in G;
(c) $l(v) = \mathrm{dist}_{(A,c)}(s, v)$ for all $v \in R \cup Q$;
(d) For all $e = (v, w) \in \delta^+_G(R)$ we have $w \in Q$ and $l(v) + c(e) \geq l(w)$;
(e) $l(v) = \mathrm{dist}_{(G,c)}(s, v)$ for all $v \in R$.

Note that (b)–(e) and $Q = \emptyset$ at the end immediately imply the correctness.

(a)–(c) clearly always hold. (d) also holds at the end of the iteration in which v gets added to R; after that, $l(v)$ remains unchanged and no $l(w)$ is ever increased.

It remains to show that (e) remains valid when a vertex v gets added to R. On account of (c) we always have that $l(v) \geq \mathrm{dist}_{(G,c)}(s, v)$; hence we will prove the opposite inequality; let $v \in V(G) \setminus \{s\}$. Consider the point in time immediately before v gets added to R. Suppose there exists an s-v-path P in G with length less than $l(v)$. Let x be the last vertex of P that already belongs to R at this point in time (in particular $x \neq v$), and let the edge $e = (x, y) \in \delta^+(x) \cap E(P)$ be the successor of x (so $y \notin R$; possibly $y = v$). Then $y \in Q$ (because of (d)) and

$$c(E(P)) \;\geq\; \mathrm{dist}_{(G,c)}(s, x) + c(e) \;=\; l(x) + c(e) \;\geq\; l(y) \;\geq\; l(v)\,,$$

as was to be shown; here the first inequality follows because all the edge weights are non-negative, the equation holds because (e) held for x, the penultimate inequality follows from (d), and the last inequality holds by the choice of v in the algorithm.

Finally, the running time: the **while** loop clearly has at most n iterations, the choice of v can be made trivially in $O(n)$ time and in the inner **for** loop every edge is considered only once. The total running time $O(n^2 + m)$ then follows. □

Remark 9.12 The invariants (b), (c) and (e) given in the proof of Theorem 9.11 imply that we can, if we are only interested in a shortest s-t-path, terminate the algorithm as soon as $t \in R$. This can improve the running time in practice but does not lead to a better asymptotic running time.

As with Prim's Algorithm, Dijkstra's Algorithm can be implemented more efficiently by using heaps:

Theorem 9.13 *By using binary heaps one can implement Dijkstra's Algorithm 9.10 so that it has running time $O(m \log n)$.*

Proof Use a binary heap as the set Q, where v has, of course, the key $l(v)$. Then we have up to n `extract_min` operations, up to n `insert` operations and up to m `decrease_key` operations (cf. Program 9.8). The running time then follows with Theorem 8.19. □

Various different versions of Dijkstra's Algorithm are in use for solving the majority of Shortest Path Problems in practice. However, when the edge weights can also be negative, Dijkstra's Algorithm fails completely in general. In this case other, slower algorithms are needed.

9.4 Conservative Edge Weights

Definition 9.14 Let G be a graph with edge weights $c: E(G) \to \mathbb{R}$. Then c is called **conservative** if (G, c) contains no circuit with negative weight.

For a given path P and vertices $x, y \in V(P)$ let $P_{[x,y]}$ denote the subgraph of P that is an x-y-path. The following lemma is of cardinal importance because its statement is equivalent to the existence of a shortest-path tree.

Lemma 9.15 *Let G be a graph and $c: E(G) \to \mathbb{R}$ conservative. Let $s, w \in V(G)$ with $s \neq w$. Let P be a shortest s-w-path and $e = (v, w)$ the last edge in P. Then $P_{[s,v]}$ is a shortest s-v-path.*

Proof Suppose that there exists an s-v-path Q with $c(E(Q)) < c(E(P_{[s,v]}))$. Then $c(E(Q)) + c(e) < c(E(P))$.

If $w \notin V(Q)$, then $(V(Q) \cup \{w\}, E(Q) \cup \{e\})$ is an s-w-path which is shorter than P; a contradiction. So $w \in V(Q)$ and thus $C := (V(Q_{[w,v]}), E(Q_{[w,v]}) \cup \{e\})$ is a circuit in G for which

$$\begin{aligned} c(E(C)) &= c(E(Q_{[w,v]})) + c(e) \\ &= c(E(Q)) + c(e) - c(E(Q_{[s,w]})) \\ &< c(E(P)) - c(E(Q_{[s,w]})) \\ &\leq 0 \end{aligned}$$

holds, which contradicts the fact that c is conservative. □

In general, this property does not hold when arbitrary edge weights are allowed. The correctness of the following algorithm will, for the case of directed graphs, yield a second proof of Lemma 9.15:

Algorithm 9.16 (Moore-Bellman-Ford Algorithm)

Input: a digraph G with conservative edge weights $c: E(G) \to \mathbb{R}$, a vertex $s \in V(G)$.
Output: an arborescence $A := (R, T)$ in G such that R contains all vertices reachable from s and $\mathrm{dist}_{(G,c)}(s, v) = \mathrm{dist}_{(A,c)}(s, v)$ for all $v \in R$.

```
$n \leftarrow |V(G)|$
$l(v) \leftarrow \infty$ for all $v \in V(G) \setminus \{s\}$
$l(s) \leftarrow 0$
for $i \leftarrow 1$ to $n - 1$ do
    for $e = (v, w) \in E(G)$ do
        if $l(v) + c(e) < l(w)$ then $l(w) \leftarrow l(v) + c(e),\ p(w) \leftarrow e$
$R \leftarrow \{v \in V(G) : l(v) < \infty\}$
$T \leftarrow \{p(v) : v \in R \setminus \{s\}\}$
```

Theorem 9.17 *The Moore-Bellman-Ford Algorithm 9.16 works correctly and has running time $O(mn)$, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof The running time is clear. At any point in time of the algorithm define $A := (R, T)$ with $R := \{v \in V(G) : l(v) < \infty\}$ and $T := \{p(v) : v \in R \setminus \{s\}\}$. We will first show that with these definitions the following invariants always hold:

(a) For all $y \in R \setminus \{s\}$ with $p(y) = e = (x, y)$ we have $l(y) \geq l(x) + c(e)$;
(b) A is an arborescence with root s in G;
(c) $l(v) \geq \mathrm{dist}_{(A,c)}(s, v)$ for all $v \in R$.

For the proof of (a) we note: when $l(y)$ was last changed, $p(y)$ was also set to $e = (x, y)$ and then $l(y) = l(x) + c(e)$ was valid; since then $l(x)$ could only have decreased.

From (a) it follows in particular that $l(v) \geq l(w) + c(E(P))$ for every w-v-path P in A. We will now prove (b). Suppose that the insertion of an edge $e = (v, w)$ generates a circuit in A. Then it follows that immediately beforehand there already existed a w-v-path P in A and thus $l(v) \geq l(w) + c(E(P))$ was valid. On the other hand, $l(w) > l(v) + c(e)$ was valid, as otherwise $p(w)$ would not have been set to e. Hence $c(E(P)) + c(e) < 0$ and so $(V(P), E(P) \cup \{e\})$ is a circuit with negative weight; a contradiction because c is conservative. Thus A always satisfies the condition (g) in Theorem 6.23; i.e. (b) holds.

(a) and (b) immediately imply (c) and so we have that $l(v) \geq \mathrm{dist}_{(G,c)}(s, v)$ for all v. In order to complete the proof of correctness, we will prove the following

Claim For all $k \in \{0, \ldots, n-1\}$, for all $v \in V(G)$ and for all s-v-paths P in G with $|E(P)| \leq k$ we have that after k iterations $l(v) \leq c(E(P))$.

As no path in G has more than $n-1$ edges, the claim implies the correctness of the algorithm.

The proof of the claim follows by induction over k. For $k=0$ it is trivial because after zero iterations $l(s)=0=\mathrm{dist}_{(G,c)}(s,s)$.

So let $k\in\{1,\ldots,n-1\}$, and let $v\in V(G)\setminus\{s\}$ be a vertex and P an s-v-path in (G,c) with $|E(P)|\le k$. Let $e=(u,v)$ be the last edge of this path. By the induction hypothesis we have that $l(u)\le c(E(P_{[s,u]}))$ after $k-1$ iterations, and in iteration k one considers, among others, the edge e, with which $l(v)\le l(u)+c(e)\le c(E(P_{[s,u]}))+c(e)=c(E(P))$ is ensured. $\square$

This algorithm is due to Moore [26] and is based on papers of Bellman [3] and Ford [13]. It is the fastest algorithm to date for the Shortest Path Problem in directed graphs with conservative weights.

The Moore-Bellman-Ford Algorithm cannot be used for undirected graphs with conservative edge weights because replacing an edge $\{v,w\}$ with negative weight by two edges (v,w) and (w,v) with the same weight generates a circuit (of length 2) with negative weight.

It is also worth noting that all the algorithms discussed in this chapter work for real-numbered edge weights so long as one can work with real numbers, in particular do comparisons and the basic operations of arithmetic (here one in fact needs only addition). In practice one of course has to restrict oneself to rational or machine numbers. In order to avoid rounding errors, one often chooses to restrict oneself to integers.

9.5 Shortest Paths with Arbitrary Edge Weights

For every computational problem the exact method of encoding the input (with zeros and ones and/or sequences of real numbers) should be stipulated, even if we do not as a rule state this concretely as all reasonable encodings are equivalent.

Definition 9.18 An algorithm whose input consists of zeros and ones is called **polynomial** (one also says: it has polynomial running time), if its running time is $O(n^k)$ for some $k\in\mathbb{N}$, where n is the length of the input. An algorithm whose input may also contain real numbers is called **strongly polynomial** if it is polynomial for rational inputs and there is a $k\in\mathbb{N}$ such that the number of computational steps (including comparisons and basic operations of arithmetic involving real numbers) is $O(n^k)$, where n is the number of bits and real numbers of the input.

All the algorithms treated so far in this book, except for those in Chaps. 1 and 5, have polynomial running time. All the algorithms presented so far in this chapter (as well as those in Chap. 6) are even strongly polynomial.

If we permit arbitrary edge weights, however, no polynomial algorithm is known for the Shortest Path Problem. But one can at least do better than checking through all paths, as was shown by Held and Karp [18]:

Theorem 9.19 *The general Shortest Path Problem can be solved in $O(m + n^2 2^n)$ time, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof Let (G, c, s, t) be an instance of the Shortest Path Problem. We can, as we did in the proof of Theorem 9.4, remove all parallel edges beforehand. For $A \subseteq V(G)$ with $s \in A$ and $a \in A$ let $l_A(a)$ be the length of a shortest s-a-path with vertex set A, or ∞ if no such path exists.

We have $l_{\{s\}}(s) = 0$ and $l_A(s) = \infty$ for all $A \supset \{s\}$. For $A \subseteq V(G)$ with $s \in A$ and $a \in A \setminus \{s\}$ the following also holds:

$$l_A(a) = \min\bigl\{l_{A\setminus\{a\}}(b) + c(e) \,:\, b \in A \setminus \{a\},\, e = (b, a) \in \delta^-(a) \text{ or } \{b, a\} \in \delta(a)\bigr\}\,.$$

Computing all these numbers, of which there are not more than $2^{n-1}n$, can be done in $O(2^n n^2)$ time by computing $l_A(a)$ for increasing values of $|A|$.

We finally have that $\mathrm{dist}_{(G,c)}(s, t) = \min\{l_A(t) : \{s, t\} \subseteq A \subseteq V(G)\}$, and by means of a set A fulfilling this minimum, together with the numbers computed earlier, one can find a shortest path in $O(n^2)$ time. □

Of course, this algorithm is not polynomial and is useless for $n \geq 50$, if not earlier. A faster algorithm is, however, not known. A polynomial algorithm for the general Shortest Path Problem exists if and only if *P=NP* holds; this is arguably the most important open problem of Algorithmic Mathematics. Here *P* denotes the set of all decision problems for which a polynomial algorithm exists, while *NP* denotes the set of all decision problems with the following property: there exists a polynomial p such that for all $n \in \mathbb{N}$ the following holds: for each instance with n bits for which the correct answer is "yes", there exists a sequence of at most $p(n)$ bits (called the certificate of the instance) which "proves" the statement: there exists a polynomial algorithm that checks pairs of instances and their alleged certificates for correctness.

For example, the decision problem as to whether or not a given undirected graph contains a spanning circuit (often called a Hamiltonian circuit) is in *NP* because a Hamiltonian circuit serves as a certificate. Furthermore, for this problem a polynomial algorithm exists only if *P=NP* holds. The same holds for thousands of important computational problems, which is why this question is so significant.

One of these is the famous Traveling Salesman Problem: here one wants to find a minimum weight Hamiltonian circuit in a given complete graph with edge weights. Note that the Traveling Salesman Problem can also be solved using the numbers $l_A(a)$ computed in the proof of Theorem 9.19: the length of a shortest Hamiltonian circuit is clearly $\min\{l_{V(G)}(t) + c(e) : e = \{t, s\} \in \delta(s) \text{ or } e = (t, s) \in \delta^-(s)\}$. Here, too, the resulting algorithm of Held und Karp with running time $O(n^2 2^n)$ is still the one with the best known asymptotic running time. Even so, one has managed (with other algorithms) to solve instances with many thousands of vertices optimally [2].

10 Matchings and Network Flows

In this chapter we will solve two further fundamental combinatorial optimization problems: maximum matchings in bipartite graphs and maximum flows in networks. It will turn out that the first of these is a special case of the second. Thus it will not come as a surprise that the same basic method, namely augmenting paths, will be the key to solving both.

10.1 The Matching Problem

Definition 10.1 Let G be an undirected graph. A set of edges $M \subseteq E(G)$ is called a **matching** in G if $|\delta_G(v) \cap M| \leq 1$ for all $v \in V(G)$. A **perfect matching** is a matching with $\frac{|V(G)|}{2}$ edges.

Computational Problem 10.2 (Matching Problem)

Input: an undirected graph G.
Task: find a matching in G with maximum cardinality.

Proposition 10.3 *A maximal matching can be found in $O(m + n)$ time, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof This is achieved by the greedy algorithm which checks every edge and includes it in the initially empty set M if and only if M remains a matching. □

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_10

Theorem 10.4 *If M is a maximal matching and M^* a maximum matching, then $|M| \geq \frac{1}{2}|M^*|$.*

Proof Given a matching M, let $V_M := \{x \in V(G) : \delta_G(x) \cap M \neq \emptyset\}$ be the set of vertices covered by M. Then $|V_M| = 2|M|$. If M is maximal, then every $e \in E(G)$, and thus also every $e \in M^*$, has at least one vertex in V_M. As no two edges in M^* have a common vertex, $|M^*| \leq |V_M| = 2|M|$. □

The bound given in Theorem 10.4 cannot be improved, as is shown by a path of length 3, for example. Thus the greedy algorithm yields a matching which is smaller than an optimal one by at most a factor of 2. Here one also speaks of a 2-approximation algorithm.

The following definition is basic to nearly all matching algorithms:

Definition 10.5 Let G be an undirected graph and M a matching in G. An ***M*-augmenting path** in G is a path P in G with $|E(P) \cap M| = |E(P) \setminus M| - 1$, whose endpoints are not incident with any edge of M.

Thus every M-augmenting path has odd length and its edges lie alternately in $E(G) \setminus M$ and M. The following cardinal result due to Petersen [28] and Berge [4] characterizes maximum matchings.

Theorem 10.6 *Let G be an undirected graph and M a matching in G. Then there is a matching M' in G with $|M'| > |M|$ if and only if there is an M-augmenting path in G.*

Proof If P is an M-augmenting path, then $M \triangle E(P)$ is a larger matching.

Conversely, if M' is a matching with $|M'| > |M|$, then $(V(G), M \triangle M')$ is a graph with no vertices of degree greater than 2. Thus $M \triangle M'$ can be partitioned into edge sets of cycles of even length with edges belonging alternately to M and M', and of pairwise vertex-disjoint paths with edges also belonging alternately to M and M'. Then there exists at least one such path P for which $|E(P) \cap M| < |E(P) \cap M'|$ holds and thus P is M-augmenting. □

10.2 Bipartite Matchings

We will now present an algorithm going back to van der Waerden and König, which solves the Matching Problem in bipartite graphs optimally. An equally efficient but much more complicated algorithm exists for general graphs.

Algorithm 10.7 (Bipartite Matching Algorithm)

Input: a bipartite graph G.
Output: a matching M in G with maximum cardinality.

```
$M \leftarrow \emptyset$
$X \leftarrow \{v \in V(G) : \delta(v) \neq \emptyset\}$
Find a bipartition $V(G) = A \dot\cup B$ of $G[X]$
while true do
    $V(H) \leftarrow A \dot\cup B \dot\cup \{s,t\}$
    $E(H) \leftarrow (\{s\} \times (A \cap X)) \cup \{(a,b) \in A \times B : \{a,b\} \in E(G) \setminus M\}$
                $\cup \{(b,a) \in B \times A : \{a,b\} \in M\} \cup ((B \cap X) \times \{t\})$
    if $t$ is not reachable in $H$ from $s$ then stop
    Let $P$ be an $s$-$t$-path in $H$
    $M \leftarrow M \triangle \{\{v,w\} \in E(G) : (v,w) \in E(P)\}$
    $X \leftarrow X \setminus V(P)$
```

Theorem 10.8 *The Bipartite Matching Algorithm 10.7 works correctly and runs in $O(nm)$ time, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof Let P be an s-t-path in H. Then $(V(P) \setminus \{s,t\}, \{\{v,w\} \in E(G) : (v,w) \in E(P)\})$ is an M-augmenting path and therefore M is increased correctly in the algorithm. Furthermore, X always contains those non-isolated vertices of G which are not covered by M.

Conversely, every M-augmenting path P in G has endpoints $\bar a \in A \cap X$ and $\bar b \in B \cap X$. Thus $(V(P) \cup \{s,t\}, \{(s,\bar a), (\bar b, t)\} \cup \{(a,b) : \{a,b\} \in E(P) \setminus M\} \cup \{(b,a) : \{a,b\} \in E(P) \cap M\})$ is an s-t-path in H. Hence by Theorem 10.6 M is maximum when the algorithm terminates.

The running time of the algorithm follows from the fact that in each of the at most $\frac{n}{2}$ iterations of Algorithm 7.1 one either finds an s-t-path in $O(m)$ time or decides that none exists. $\square$

The following famous theorem was first proved by Frobenius [16]:

Theorem 10.9 (Marriage Theorem) *A bipartite graph $G = (A \dot\cup B, E(G))$ with $|A| = |B|$ has a perfect matching if and only if $|N(S)| \geq |S|$ for all $S \subseteq A$.*

Proof If G has a perfect matching M, then $|N_G(S)| \geq |N_{(V(G),M)}(S)| = |S|$ holds for all $S \subseteq A$. Thus the condition is necessary.

We will show by induction over $|A|$ that the condition is also sufficient. This clearly is the case if $|A| = 1$. So let $|A| \geq 2$.

If $|N(S)| \geq |S| + 1$ holds for all $\emptyset \neq S \subset A$, let $e = \{a,b\}$ be any edge in G. Then by the induction hypothesis the graph $G' := G - a - b$ has a perfect matching

M': because $|N_{G'}(S)| \geq |N_G(S)| - 1 \geq |S|$ holds for all $\emptyset \neq S \subseteq A \setminus \{a\}$. Therefore $M' \cup \{e\}$ is a perfect matching in G.

Thus we can assume that there exists a $\emptyset \neq S \subset A$ with $|N(S)| = |S|$. Then by the induction hypothesis $G[S \cup N(S)]$ has a perfect matching M. We now consider the graph $G' := G[(A \setminus S) \cup (B \setminus N(S))]$. For every $T \subseteq A \setminus S$ we have $|N_{G'}(T)| = |N_G(T \cup S) \setminus N_G(S)| = |N_G(T \cup S)| - |S| \geq |T \cup S| - |S| = |T|$. Therefore by the induction hypothesis G' has a perfect matching M'. Thus $M \cup M'$ is a perfect matching in G. ☐

10.3 Max-Flow-Min-Cut Theorem

Definition 10.10 Let G be a digraph with $u: E(G) \to \mathbb{R}_{\geq 0}$ (where $u(e)$ denotes the **capacity** of the edge e) and with two special vertices $s \in V(G)$, the **source**, and $t \in V(G)$, the **sink**. The 4-tuple (G, u, s, t) is called a **network**.

An ***s*-*t*-flow** in (G, u) is a function $f: E(G) \to \mathbb{R}_{\geq 0}$ with $f(e) \leq u(e)$ for all $e \in E(G)$ and

$$f(\delta^-(v)) = f(\delta^+(v)) \text{ for all } v \in V(G) \setminus \{s, t\} \tag{10.1}$$

as well as

$$\text{val}(f) := f(\delta^+(s)) - f(\delta^-(s)) \geq 0 .$$

We denote val(f) to be the **value** of f.

Here we have again used the very practical notation $f(A) := \sum_{e \in A} f(e)$. Condition (10.1) is called the flow conservation rule. An example is given in Fig. 10.1.

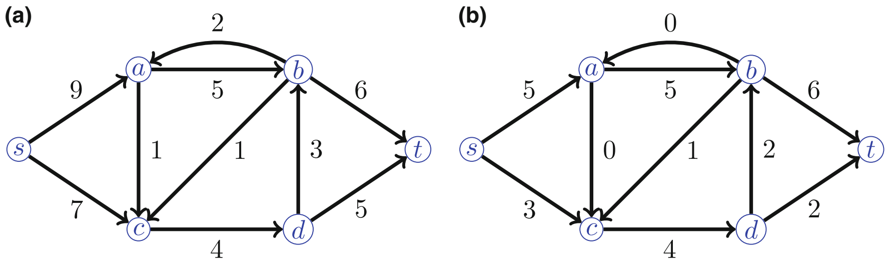


Fig. 10.1 (**a**) A network (the numbers along the edges denote their capacities) and (**b**) an s-t-flow of value 8 contained in it

Computational Problem 10.11 (Flow Problem)

Input: a network (G, u, s, t).
Task: find a maximum value s-t-flow in (G, u).

A maximum value s-t-flow (or more briefly a maximum flow) always exists because we are maximizing a continuous function over a compact set. We will, however, also prove this constructively (algorithmically).

Lemma 10.12 *Let (G, u, s, t) be a network and f an s-t-flow in (G, u). Then we have for all $A \subset V(G)$ with $s \in A$ and $t \notin A$:*

(a) $\mathrm{val}(f) \;=\; f(\delta^+(A)) - f(\delta^-(A))$
(b) $\mathrm{val}(f) \;\le\; u(\delta^+(A))$.

Proof

(a): By the flow conservation rule we have for all $v \in A \setminus \{s\}$ that:

$$\begin{aligned}\mathrm{val}(f) = f(\delta^+(s)) - f(\delta^-(s)) &= \sum_{v\in A}\big(f(\delta^+(v)) - f(\delta^-(v))\big)\\ &= f(\delta^+(A)) - f(\delta^-(A)) .\end{aligned}$$

(b): As $0 \le f(e) \le u(e)$ for all $e \in E(G)$, it follows from (a) that

$$\mathrm{val}(f) \;=\; f(\delta^+(A)) - f(\delta^-(A)) \;\le\; u(\delta^+(A)) .$$

□

This motivates the following definition:

Definition 10.13 Let (G, u, s, t) be a network. An ***s*-*t*-cut** in G is an edge set $\delta^+(X)$ with $X \subset V(G)$ and $s \in X, t \notin X$. The **capacity** of an s-t-cut $\delta^+(X)$ is $u(\delta^+(X))$ (i.e. the sum of the capacities of its edges).

By Lemma 10.12 the maximum value of an s-t-flow cannot be greater than the minimum capacity of an s-t-cut. In fact, equality holds, as we will now proceed to show algorithmically. First we will define how to extend ("augment") an s-t-flow stepwise along s-t-paths.

Definition 10.14 Let (G, u, s, t) be a network and f an s-t-flow. For a given $e = (x, y) \in E(G)$, let $\overleftarrow{e}$ denote a new (i.e. not belonging to G) **reverse edge** from y to x. Then the **residual graph** G_f is defined as follows: $V(G_f) := V(G)$ and $E(G_f) := \{e \in E(G) : f(e) < u(e)\} \,\dot\cup\, \{\overleftarrow{e} : e \in E(G), f(e) > 0\}$. An ***f*-augmenting**

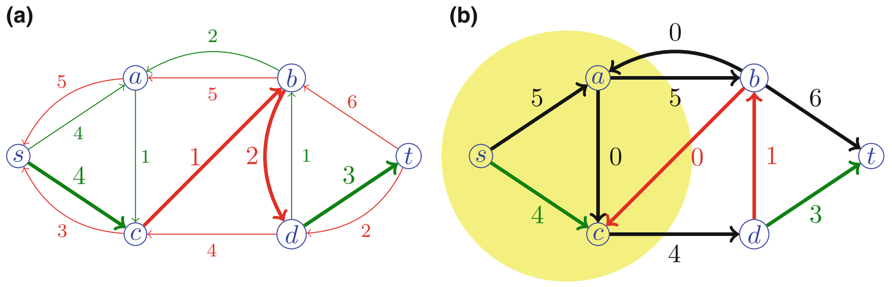


Fig. 10.2 (**a**) The residual graph G_f with residual capacities u_f for the network (G, u, s, t) and the s-t-flow f depicted in Fig. 10.1; an f-augmenting path P highlighted by thick edges (with the vertices s, c, b, d, t); (**b**) the s-t-flow obtained by augmenting f along P by the amount 1. This flow has maximum value, as is shown by the s-t-cut induced by the set in the shaded circular area

path is an s-t-path in G_f. The **residual capacities** $u_f\colon E(G) \mathbin{\dot\cup} \{\overleftarrow{e}: e \in E(G)\} \to \mathbb{R}_{\geq 0}$ are defined as follows: $u_f(e) := u(e) - f(e)$ and $u_f(\overleftarrow{e}) := f(e)$ for all $e \in E(G)$.

Figure 10.2 depicts an example. Note that G_f can contain parallel edges, even when G is simple. All the edges of the residual graph have positive residual capacities.

Lemma 10.15 *Let (G, u, s, t) be a network, f an s-t-flow in (G, u), and P an f-augmenting path. Let $0 \leq \gamma \leq \min_{e \in E(P)} u_f(e)$. Then we define the result of the augmentation of f along P by the amount γ to be $f'\colon E(G) \to \mathbb{R}_{\geq 0}$ with $f'(e) := f(e) + \gamma$ for $e \in E(G) \cap E(P)$, $f'(e) := f(e) - \gamma$ for $e \in E(G)$ with $\overleftarrow{e} \in E(P)$, and $f'(e) := f(e)$ for all the remaining edges e of G. Then this f' is an s-t-flow whose value is greater than that of f by the amount γ.*

Proof The lemma is a direct consequence of the definition of residual capacities. □

Theorem 10.16 *Let (G, u, s, t) be a network. An s-t-flow f in (G, u) has maximum value if and only if there is no f-augmenting path.*

Proof If an f-augmenting path P exists, then by Lemma 10.15, f can be augmented along P by the amount $\min_{e \in E(P)} u_f(e)$ to a flow of greater value.

If there is no f-augmenting path, then t is not reachable from s in G_f. Let R be the set of vertices that are reachable from s in G_f. Then $f(e) = u(e)$ for all $e \in \delta_G^+(R)$ and $f(e) = 0$ for all $e \in \delta_G^-(R)$, as there would otherwise (by the definition of G_f) be an edge of the residual graph which leaves R. Hence we have by Lemma 10.12(a):

$$\mathrm{val}(f) \;=\; f(\delta_G^+(R)) - f(\delta_G^-(R)) \;=\; u(\delta_G^+(R))\,,$$

which implies by Lemma 10.12(b) that f has maximum value. □

We can now state and prove a famous theorem of Dantzig, Ford and Fulkerson [8, 14]:

Theorem 10.17 (Max-Flow-Min-Cut Theorem) *In every network (G, u, s, t) the maximum value of an s-t-flow equals the minimum capacity of an s-t-cut.*

Proof By Lemma 10.12(b) the value of an s-t-flow is always less than or equal to the capacity of an s-t-cut. In the proof of Theorem 10.16 it was shown that for every maximum s-t-flow f there exists an s-t-cut $\delta^+(R)$ with $\mathrm{val}(f) = u(\delta^+(R))$. □

10.4 Algorithms for Maximum Flows

The results of the previous section suggest the following algorithm due to Ford and Fulkerson [15]:

Algorithm 10.18 (Ford-Fulkerson Algorithm)

Input:	a network (G, u, s, t) with integer capacities.
Output:	an s-t-flow f of maximum value, with the additional property that the flow is integer-valued.

```
$f(e) \leftarrow 0$ for all $e \in E(G)$
while there exists an $f$-augmenting path $P$ do
    $\gamma \leftarrow \min_{e \in E(P)} u_f(e)$
    augment $f$ along $P$ by the amount $\gamma$
```

Theorem 10.19 *The Ford-Fulkerson Algorithm 10.18 works correctly and can be implemented to run in $O(mW)$ time, where $m = |E(G)|$ and W is the value of a maximum s-t-flow. In particular $W \leq u(\delta^+(s))$.*

Proof In each iteration the algorithm increases the value of f by an integer amount γ (the residual capacities are always integer-valued). It will therefore terminate after at most W iterations and will, moreover, end with an s-t-flow of maximum value by Theorem 10.16. It follows by Lemma 10.12(a) that $W \leq u(\delta^+(s))$. Every iteration can be completed in $O(m)$ time using graph traversal methods. □

The following result is worth noting:

Corollary 10.20 *Let (G, u, s, t) be a network with integer capacities. Then there exists a maximum value s-t-flow which is integer-valued.*

Proof The Ford-Fulkerson Algorithm finds such a flow. □

Note that the Bipartite Matching Algorithm 10.7 is a special case of the Ford-Fulkerson Algorithm, namely when the latter is applied to the digraph H constructed in the first iteration with capacities equal to 1 on all its edges. One can also derive the Marriage Theorem 10.9 from the Max-Flow-Min-Cut Theorem.

The Ford-Fulkerson Algorithm does not in general run in polynomial time, as the example in Fig. 10.3 shows.

The Ford-Fulkerson Algorithm is totally unsuitable for networks with irrational capacities: one could in this case not even guarantee that it would ever terminate or that the value of f would necessarily converge to the optimum.

The Ford-Fulkerson Algorithm was improved by Edmonds and Karp [11], who introduced breadth-first search in G_f for finding the augmenting paths. This yielded a strongly polynomial algorithm, as we will now show.

Lemma 10.21 *Let G be a directed graph and s, t two vertices. Let F be the union of the edge sets of all the shortest s-t-paths in G, and $e \in F$. Then F is also the union of the edge sets of all the shortest s-t-paths in $(V(G), E(G) \cup \{\overleftarrow{e}\})$.*

Proof Let $k := \mathrm{dist}_G(s,t)$ and let P be an s-t-path with $|E(P)| = k$ and $e \in E(P)$. Assume that there exists an s-t-path Q in $(V(G), E(G) \cup \{\overleftarrow{e}\})$ with $|E(Q)| \le k$ and $\overleftarrow{e} \in E(Q)$. Consider $H := (V(G), (\{f, g\} \dot\cup E(P) \dot\cup E(Q)) \setminus \{e, \overleftarrow{e}\})$, where f and g are two new edges from t to s. As H satisfies the conditions of Lemma 6.10, there exist two edge-disjoint cycles in H, one containing f and the other g. These then yield two s-t-paths in G of total length not exceeding $|E(H)|-2 = |E(P)|+|E(Q)|-2 \le 2k-2$, in contradiction to the definition of k. $\square$

Theorem 10.22 *If one consistently chooses augmenting paths with as few edges as possible in the Ford-Fulkerson Algorithm, then it will terminate after at most $2|E(G)||V(G)|$ iterations, even in the case of arbitrary capacities $u: E(G) \to \mathbb{R}_{\ge 0}$.*

Proof It follows from Lemma 10.21 that $\mathrm{dist}_{G_f}(s,t)$ cannot diminish. In order to show that $\mathrm{dist}_{G_f}(s,t)$ cannot remain constant for more than $2|E(G)|$ iterations, we note that during such a time interval the union of the edge sets of all the shortest

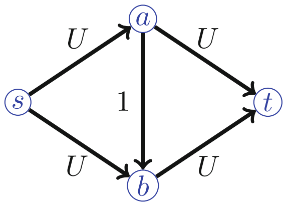


Fig. 10.3 This depicts an instance where the Ford-Fulkerson Algorithm requires U iterations, $U \in \mathbb{N}$, if one consistently augments along paths with three edges. Note here, that the input is encoded with $O(\log U)$ bits.

s-t-paths in G_f can never, by Lemma 10.21, be increased, but will in each iteration be reduced by those edges $e \in E(P)$ for which $u_f(e) = \gamma$. □

This is known as the Edmonds-Karp Algorithm. It thus runs in $O(m^2 n)$ time, where $n = |V(G)|$ and $m = |E(G)|$. Furthermore, it yields a constructive proof of the fact that every network has a maximum s-t-flow. If, however, the capacities are entered in the input as machine numbers, then it does not necessarily follow that a maximum s-t-flow consisting of machine numbers will exist.

Note that the Flow Problem is a special type of linear program: a certain linear objective function is maximized over a domain defined by a set of linear inequalities. One can, in fact, solve general linear programs in polynomial time, but that is a somewhat more complicated matter. In the next and last chapter we will deal with a further very special case: solving systems of linear equations.

Gaussian Elimination 11

In this chapter we will deal with the problem of solving systems of linear equations. These take the form

$$\begin{array}{ccccccccc}\alpha_{11}\xi_1 & + & \alpha_{12}\xi_2 & + & \ldots & + & \alpha_{1n}\xi_n & = & \beta_1\\ \vdots & & & & & & \vdots & & \vdots\\ \alpha_{m1}\xi_1 & + & \alpha_{m2}\xi_2 & + & \ldots & + & \alpha_{mn}\xi_n & = & \beta_m\end{array}$$

(or more briefly $Ax = b$), where $A = (\alpha_{ij})_{1\le i\le m,\,1\le j\le n} \in \mathbb{R}^{m\times n}$ and $b = (\beta_1, \ldots, \beta_m)^\top \in \mathbb{R}^m$ are given and one wishes to determine $x = (\xi_1, \ldots, \xi_n)^\top \in \mathbb{R}^n$. In other words, one wishes to solve the following numerical computational problem:

Computational Problem 11.1 (Systems of Linear Equations)

Input: a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $b \in \mathbb{R}^m$.
Task: find a vector $x \in \mathbb{R}^n$ with $Ax = b$ or decide that none exists.

We know from linear algebra that this problem is closely related to that of finding the rank and, if $m = n$, the determinant and, if A is nonsingular, the inverse A^{-1} of A; cf. the box **Rank and Determinant**. All these problems are solved by the Gaussian Elimination Method which we will study in this chapter. Problem 11.1 was defined for real numbers, but one can equally well substitute any other field ($\mathbb{C}$ for example) in which one is able to calculate.

Algorithms for solving systems of linear equations play a fundamental role not only in linear algebra but also in the numerical solution of systems of differential equations, in linear and nonlinear optimization and thus also in numerous applications.

DOI 10.1007/978-3-319-39558-6_11

Rank and Determinant

Let $A \in \mathbb{R}^{m\times n}$. Then the **rank** of A is defined to be the maximum number of linearly independent column vectors of A. This is equal to the maximum number of linearly independent row vectors of A. The **determinant** of a square matrix $A \in \mathbb{R}^{n\times n}$ with $A = (\alpha_{ij})_{1\le i,j\le n}$ is defined as follows:

$$\det(A) := \sum_{\sigma\in S_n} \operatorname{sign}(\sigma) \prod_{i=1}^{n} \alpha_{i\sigma(i)} ,$$

where S_n is the set of all permutations of the set $\{1,\dots,n\}$ and $\operatorname{sign}(\sigma)$ denotes the sign of the permutation σ. A matrix $A \in \mathbb{R}^{n\times n}$ is **nonsingular** if there exists a matrix $B \in \mathbb{R}^{n\times n}$ with $AB = I$, otherwise A is **singular**. A matrix A is singular if and only if $\det(A) = 0$. If there exists a matrix B with $AB = I$, then B is called the **inverse** of A and is denoted by A^{-1}. The inverse of a matrix A is unique and the following equations hold: $AA^{-1} = A^{-1}A = I$. Furthermore, $(AB)^{-1} = B^{-1}A^{-1}$ holds for two nonsingular matrices A and B. If A and B are two matrices in $\mathbb{R}^{n\times n}$, then one can easily check that $\det(AB) = \det(A)\det(B)$ holds. An immediate consequence of the definition of a determinant is the Laplace Expansion Theorem, which states that for every $j \in \{1,\dots,n\}$ the following holds:

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+j} \alpha_{ij} \det(A_{ij}) ,$$

where A_{ij} denotes the matrix obtained from A by deleting the i-th row and the j-th column.

Vectors will always be assumed to be column vectors, unless they have been transposed (i.e. for $x \in \mathbb{R}^n$, $x^\top$ is a row vector). For $x, y \in \mathbb{R}^n$, $x^\top y \in \mathbb{R}$ is their standard scalar product (also called inner product). For $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, $xy^\top \in \mathbb{R}^{m\times n}$ is their outer product. I denotes an identity matrix, i.e. a square matrix $(\delta_{ij})_{1\le i,j\le n}$ with $\delta_{ij} = 1$ for $i = j$ and $\delta_{ij} = 0$ for $i \ne j$. The columns of I are called unit vectors and are denoted by $e_1,\dots,e_n$. The dimensions of the unit vectors and the identity matrices will always be clear from the context. Analogously, the symbol 0 denotes the number zero or a vector or matrix with all entries equal to zero.

11.1 The Operations of Gaussian Elimination

This fundamental method of linear algebra bears the name of Carl Friedrich Gauss, but it was already known in China more than 2000 years before his time. It is based on the following elementary operations:

(1) interchange two rows
(2) interchange two columns
(3) add a multiple of one row to another row

Here the term row denotes a row of A together with the corresponding entry in b, and the term column denotes a column of A together with the corresponding entry of x. These operations have no effect on the solution set, except that (2) changes the order of the variables. This is clear for the first two operations. For the third operation we have:

Lemma 11.2 *Let $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Let $p, q \in \{1, \dots, m\}$ with $p \neq q$ and $\delta \in \mathbb{R}$. Adding δ times the p-th row to the q-th row of A means replacing A by GA, where $G := I + \delta e_q e_p^\top \in \mathbb{R}^{m \times m}$.*

G is nonsingular with $G^{-1} = I - \delta e_q e_p^\top$, and $\{x : Ax = b\} = \{x : GAx = Gb\}$. Moreover, A and GA have the same rank and, if $m = n$, the same determinant.

Proof The first statement is clear. One easily checks that $(I + \delta e_q e_p^\top)(I - \delta e_q e_p^\top) = I^2 - \delta^2 e_q e_p^\top e_q e_p^\top = I$. If $Ax = b$, then $GAx = Gb$. As G is nonsingular, the converse holds. The nonsingularity of G also implies that GA and A have the same rank. If $m = n$, then we have $\det(GA) = (\det G)(\det A) = \det A$. □

The main part of the Gaussian Elimination Method comprises the conversion of the given matrix into upper triangular form by means of the above operations:

Definition 11.3 A matrix $A = (\alpha_{ij}) \in \mathbb{R}^{m \times n}$ is called an **upper triangular matrix** if $\alpha_{ij} = 0$ for all $i > j$. A is called a **lower triangular matrix** if $\alpha_{ij} = 0$ for all $i < j$. A square triangular matrix $A \in \mathbb{R}^{m \times m}$ is called **unitriangular** if $\alpha_{ii} = 1$ for all $i = 1, \dots, m$.

The matrix G in Lemma 11.2 is a unitriangular matrix.

Proposition 11.4 *A lower unitriangular matrix is nonsingular and its determinant is 1. Its inverse is again a lower unitriangular matrix.*

Proof Let $A \in \mathbb{R}^{m \times m}$ be a lower unitriangular matrix. The proof follows by induction over m. The statement is clear for $m = 1$. Write A in the form $\left(\begin{smallmatrix} B & 0 \\ c^\top & 1 \end{smallmatrix}\right)$, where B clearly is again a lower unitriangular matrix and 0 the vector consisting of

$m-1$ zeros. Then $\det A = \det B$. Using the induction hypothesis one checks readily that $A^{-1} = \left(\begin{smallmatrix} B^{-1} & 0 \\ -c^\top B^{-1} & 1 \end{smallmatrix}\right)$. □

Similarly, the inverse of a nonsingular upper triangular matrix is again an upper triangular matrix. One converts A into upper triangular form by the repeated use of the following lemma (using the interchange of rows and/or columns where necessary in order to ensure that $\alpha_{pp} \neq 0$).

Lemma 11.5 *Let $A = (\alpha_{ij}) \in \mathbb{R}^{m\times n}$ and $p \in \{1,\dots,\min\{m,n\}\}$ with $\alpha_{ii} \neq 0$ for all $i = 1,\dots,p$ and $\alpha_{ij} = 0$ for all $j < p$ and $i > j$. Then there exists a lower unitriangular matrix $G \in \mathbb{R}^{m\times m}$ satisfying $GA = (\alpha'_{ij})$ with $\alpha'_{ii} \neq 0$ for all $i = 1,\dots,p$ and $\alpha'_{ij} = 0$ for all $j \leq p$ and $i > j$. Such a G as well as the matrix GA can be computed in $O(mn)$ steps.*

Proof Subtract $\frac{\alpha_{ip}}{\alpha_{pp}}$ times row p from row i for $i = p+1,\dots,m$. By Lemma 11.2 we obtain $G = \prod_{i=p+1}^{m}(I - \frac{\alpha_{ip}}{\alpha_{pp}} e_i e_p^\top) = I - \sum_{i=p+1}^{m} \frac{\alpha_{ip}}{\alpha_{pp}} e_i e_p^\top$. □

This first phase of the Gaussian Elimination Method requires r iterations and therefore $O(mn(r+1))$ elementary operations, where $r \leq \min\{m,n\}$ is the rank of A. The remaining part is simple, because systems of linear equations with triangular matrices can easily be solved with $O(n\min\{m,n\})$ elementary operations:

Lemma 11.6 *Let $A = (\alpha_{ij}) \in \mathbb{R}^{m\times n}$ be an upper triangular matrix and $b = (\beta_1,\dots,\beta_m)^\top \in \mathbb{R}^m$. Assume that for all $i = 1,\dots,m$ either ($i \leq n$ and $\alpha_{ii} \neq 0$) holds or ($\beta_i = 0$ and $\alpha_{ij} = 0$ for all $j = 1,\dots,n$). Then the solution set of the system $Ax = b$ comprises all $x = (\xi_1,\dots,\xi_n)$ with*

$$\xi_i \in \mathbb{R} \text{ is arbitrary} \quad \text{for all } i \text{ with } i > m \text{ or } \alpha_{ii} = 0, \text{ and}$$

$$\xi_i = \frac{1}{\alpha_{ii}}\left(\beta_i - \sum_{j=i+1}^{n} \alpha_{ij}\xi_j\right) \quad \text{for } i = \min\{m,n\},\dots,1 \text{ with } \alpha_{ii} \neq 0\,. \tag{11.1}$$

Proof By rearranging (11.1) one obtains

$$\alpha_{ii}\xi_i + \sum_{j=i+1}^{n} \alpha_{ij}\xi_j = \beta_i\,,$$

which corresponds exactly to the i-th row of the system $Ax = b$. All the other rows vanish. Note that in order to calculate ξ_i using (11.1) one only requires the values of variables with higher index. □

Clearly an analogous statement holds for lower triangular matrices. It may also occur that the i-th row of A has all its entries zero but $\beta_i \neq 0$. Obviously such a system of linear equations has no solutions.

With Lemma 11.6 we have completed the Gaussian Elimination Method. Summarizing, we have:

Theorem 11.7 *Systems of linear equations $Ax = b$ with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ can be solved with $O(mn(r+1))$ elementary operations, where r is the rank of A.*

Proof Using the elementary operations (1)–(3) we convert A into upper triangular form; see Lemma 11.5. If we now find that a row of the matrix has all its entries zero but the corresponding entry on the right-hand side is not zero, then clearly $Ax = b$ has no solution. Otherwise we apply Lemma 11.6. □

Later in this chapter we will give a formal description of this algorithm.

Example 11.8 We will solve the system of linear equations

$$\begin{array}{rcrcrcrcr} 3\xi_1 &+& \xi_2 &+& 4\xi_3 &-& 6\xi_4 &=& 3 \\ 6\xi_1 &+& 2\xi_2 &+& 6\xi_3 && &=& 2 \\ 9\xi_1 &+& 4\xi_2 &+& 7\xi_3 &-& 5\xi_4 &=& 0 \\ && \xi_2 &-& 3\xi_3 &+& \xi_4 &=& -5 \end{array}$$

as follows:

$$\begin{array}{rrrr|r} \mathbf{3} & 1 & 4 & -6 & 3 \\ 6 & 2 & 6 & 0 & 2 \\ 9 & 4 & 7 & -5 & 0 \\ 0 & 1 & -3 & 1 & -5 \end{array} \rightarrow \begin{array}{rrrr|r} 3 & 1 & 4 & -6 & 3 \\ 0 & 0 & -2 & 12 & -4 \\ 0 & \mathbf{1} & -5 & 13 & -9 \\ 0 & 1 & -3 & 1 & -5 \end{array} \rightarrow \begin{array}{rrrr|r} 3 & 1 & 4 & -6 & 3 \\ 0 & \mathbf{1} & -5 & 13 & -9 \\ 0 & 0 & -2 & 12 & -4 \\ 0 & 1 & -3 & 1 & -5 \end{array}$$

$$\rightarrow \begin{array}{rrrr|r} 3 & 1 & 4 & -6 & 3 \\ 0 & 1 & -5 & 13 & -9 \\ 0 & 0 & \mathbf{-2} & 12 & -4 \\ 0 & 0 & 2 & -12 & 4 \end{array} \rightarrow \begin{array}{rrrr|r} 3 & 1 & 4 & -6 & 3 \\ 0 & 1 & -5 & 13 & -9 \\ 0 & 0 & -2 & 12 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}$$

In the first, third and fourth steps we used Lemma 11.5, while in the second step we interchanged the second and third rows. The fourth equation is redundant. At the end ξ_4 can be chosen arbitrarily; choosing $\xi_4 = 0$ and using (11.1) yields a solution $x = (-2, 1, 2, 0)^\top$.

Instead of using Lemma 11.6 at the end of the method, one can convert the matrix into diagonal form by further application of Lemma 11.5 (here from the bottom upwards), i.e. into the form $\left(\begin{smallmatrix} D & B \\ 0 & 0 \end{smallmatrix}\right)$, where D is a diagonal matrix (i.e. a square lower and upper triangular matrix). This is known as the Gauss-Jordan Method.

11.2 LU Decomposition

The Gaussian Elimination Method yields further information:

Definition 11.9 An **LU decomposition** of a matrix $A \in \mathbb{R}^{m\times n}$ consists of a lower unitriangular matrix L and an upper triangular matrix U with $LU = A$.

Proposition 11.10 *Every nonsingular matrix has at most one LU decomposition.*

Proof Suppose $A = L_1U_1 = L_2U_2$. As U_2 is nonsingular because A is, we have $U_1U_2^{-1} = L_1^{-1}L_2$. The inverse of an upper triangular matrix and the product of two upper triangular matrices are both again upper triangular matrices. The inverse of a lower unitriangular matrix and the product of two lower unitriangular matrices are both again lower unitriangular matrices (see Proposition 11.4). Thus $U_1U_2^{-1} = L_1^{-1}L_2$ implies that both sides of this equation are equal to the identity matrix. □

Note, however, that an LU decomposition does not always exist. The matrix $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, for example, has no LU decomposition. Thus we require a more general definition:

Definition 11.11 Let $n \in \mathbb{N}$ and $\sigma\colon \{1,\ldots,n\} \to \{1,\ldots,n\}$ be a permutation. The **permutation matrix** (of order n) corresponding to σ is the matrix $P_\sigma = (\pi_{ij}) \in \{0,1\}^{n\times n}$ with $\pi_{ij} = 1$ if and only if $\sigma(i) = j$.

A **fully pivotized LU decomposition** of a matrix $A \in \mathbb{R}^{m\times n}$ consists of a lower unitriangular matrix L, an upper triangular matrix U and two permutations σ and τ with $A = P_\sigma^\top LUP_\tau$, together with the property that a diagonal element υ_{ii} of $U = (\upsilon_{ij})$ is zero only if the rows i to m of U have all their entries equal to zero.

Thus a permutation matrix is a matrix obtained from an identity matrix $I = P_{\mathrm{id}}$ (here id denotes an identity function) by interchanging rows (or columns). Multiplying a matrix on the left with $P_\sigma^\top$ has the effect of interchanging the rows according to the permutation σ (the i-th row becomes the $\sigma(i)$-th row), while multiplying a matrix on the right with P_τ has the effect of interchanging the columns according to the permutation τ. We obviously have:

Proposition 11.12 *For all $n \in \mathbb{N}$ the permutation matrices of order n together with matrix multiplication form a group (with I as neutral element). Moreover, $PP^\top = I$ holds for every permutation matrix P.* □

Nonsingular matrices A always have a partially pivotized LU decomposition, i.e. a fully pivotized one with $\tau = \mathrm{id}$. In fact, every matrix possesses a fully pivotized LU decomposition and can be obtained by Gaussian Elimination, as we will see in a moment. If one has such a decomposition, then one can readily determine the rank, the determinant (for square matrices) and the inverse (for nonsingular matrices). We have, for example:

Theorem 11.13 *Let $Ax = b$ be a system of linear equations with $A \in \mathbb{R}^{m\times n}$. If one has a fully pivotized LU decomposition of A, then one can solve the system with $O(m \max\{m, n\})$ elementary operations.*

Proof Let $A = P_\sigma^\top LUP_\tau$ be a fully pivotized LU decomposition of A. Systems of linear equations with the matrices $P_\sigma^\top$ and P_τ (trivial) as well as L and U (Lemma 11.6) can all be solved with $O(m \max\{m, n\})$ elementary operations. Thus we will successively solve $P_\sigma^\top z' = b$ (i.e. $\zeta'_i = \beta_{\sigma(i)}$ for $i = 1, \dots, m$), then $Lz'' = z'$, then $Uz''' = z''$, and finally $P_\tau x = z'''$ (i.e. $\xi_{\tau(j)} = \zeta'''_j$ for $j = 1, \dots, n$). It may happen that the system $Uz''' = z''$ has no solution; this occurs only if $Ax = b$ also has no solution. □

A fully pivotized LU decomposition is particularly useful when one has to solve many systems of linear equations with the same matrix A but different right-hand sides b. One can, of course, also use the (fully pivotized) LU decomposition to determine the rank (i.e. the rank of U) and (if the matrix is square) the determinant; here one just needs to consider the diagonal elements of U (cf. Corollary 11.16).

We will next give a formal version of the Gaussian Elimination Method which will, in particular, determine explicitly a fully pivotized LU decomposition. In practice one often has prior knowledge that A is nonsingular and can therefore do without column interchanges (i.e. $P_\tau = I$). They may, however, make sense for reasons of numerical stability; see Sect. 11.4.

The algorithm keeps track of the invariant $P_\sigma^\top LUP_\tau = A$, where σ and τ are always permutations and $L = (\lambda_{ij})$ is a lower unitriangular matrix. The algorithm terminates when U has the required properties. We will denote the i-th row of U by $\upsilon_{i\cdot}$ and the i-th column by $\upsilon_{\cdot i}$, and analogously for L.

Algorithm 11.14 (**Gaussian Elimination**)

Input: a matrix $A \in \mathbb{R}^{m \times n}$.
Output: a fully pivotized LU decomposition (σ, L, U, τ) of A. The rank r of A.

```
$\sigma(i) \leftarrow i \; (i = 1, \ldots, m)$
$L = (\lambda_{ij}) \leftarrow I \in \mathbb{R}^{m \times m}$
$U = (\upsilon_{ij}) \leftarrow A$
$\tau(i) \leftarrow i \; (i = 1, \ldots, n)$
$r \leftarrow 0$
while there exist $p, q > r$ with $\upsilon_{pq} \neq 0$ do
    choose $p \in \{r+1, \ldots, m\}$ and $q \in \{r+1, \ldots, n\}$ with $\upsilon_{pq} \neq 0$
    $r \leftarrow r + 1$
    if $p \neq r$ then swap$(\upsilon_{p\cdot}, \upsilon_{r\cdot})$, swap$(\lambda_{p\cdot}, \lambda_{r\cdot})$, swap$(\lambda_{\cdot p}, \lambda_{\cdot r})$, swap$(\sigma(p), \sigma(r))$
    if $q \neq r$ then swap$(\upsilon_{\cdot q}, \upsilon_{\cdot r})$, swap$(\tau(q), \tau(r))$
    for $i \leftarrow r + 1$ to $m$ do
        $\lambda_{ir} \leftarrow \frac{\upsilon_{ir}}{\upsilon_{rr}}$
        for $j \leftarrow r$ to $n$ do $\upsilon_{ij} \leftarrow \upsilon_{ij} - \lambda_{ir}\upsilon_{rj}$.
```

The element υ_{pq} chosen at the start of every iteration is called pivot element. If the input matrix is nonsingular, then one can always choose $q = r + 1$ and can thus avoid column interchanges. This is called partial pivotization.

After every iteration of the algorithm the following clearly holds: $\lambda_{ij} \neq 0$ implies $i = j$ (and thus $\lambda_{ij} = 1$) or $\upsilon_{ij} = 0$ (where $U = (\upsilon_{ij})$). Hence we can, in a memory-saving implementation, store L and U in the same matrix.

Theorem 11.15 *Gaussian Elimination (Algorithm 11.14) works correctly and requires $O(mn(r+1))$ elementary operations, where r is the rank of A.*

Proof In order to prove the correctness, we will first show that the invariant $P_\sigma^\top L U P_\tau = A$ always holds. This is obviously so at the beginning.

A row interchange ($p \neq r$) has the following effect: interchanging columns in the matrix L and interchanging the corresponding rows in U does not alter the product LU, as $(LP_\pi)(P_\pi^\top U) = LU$. The interchange of rows in L is compensated by the change in σ; this does not alter the product $P_\sigma^\top L$.

In a column interchange ($q \neq r$) the columns of U are interchanged; this is compensated by the change in τ; the product UP_τ remains unaltered.

It remains to check the change in the values of λ_{ir} and υ_{ij} ($j = r, \ldots, n$) in the **for** loop; we need to show that LU remains constant here. For this we apply Lemma 11.5, i.e. Lemma 11.2 successively for $i = r + 1, \ldots, m$. This involves multiplying U on the left by $I - \lambda_{ir} e_i e_r^\top$. At the same time, however, λ_{ir} is set, which is equivalent to multiplying L on the right with $I + \lambda_{ir} e_i e_r^\top$ (because at this point in time $\lambda_{\cdot i} = e_i$, whereas we had $\lambda_{ir} = 0$ earlier). As $(I + \lambda_{ir} e_i e_r^\top)(I - \lambda_{ir} e_i e_r^\top) = I$, the product LU remains unaltered. This completes the proof of the invariance of $P_\sigma^\top LUP_\tau = A$.

Clearly σ and τ are always permutations. Moreover, L is always a lower unitriangular matrix: all entries that are changed lie below the diagonal and interchanges of pairs of rows and the corresponding pairs of columns always occur simultaneously. After iteration r we have $\upsilon_{rr} \neq 0$ and $\upsilon_{ir} = 0$ for all $i = r+1, \dots, m$, and these values remain unaltered from this point onwards. When the algorithm terminates, we have $\upsilon_{ij} = 0$ for all $i = r+1, \dots, m$ and all $j = 1, \dots, n$.

It also follows that at the end r is the rank of U and therefore also that of A (as the permutation matrices and L are nonsingular). The running time is obvious. □

The following result is worth noting:

Corollary 11.16 *The determinant of a given matrix $A \in \mathbb{R}^{n\times n}$ can be computed in $O(n^3)$ steps.*

Proof The proof follows by applying Algorithm 11.14. Each row or column interchange implies multiplying the determinant with -1. After the completion of k row and l column interchanges, we have $\det A = (\det P_\sigma^\top)(\det L)(\det U)(\det P_\tau) = (-1)^k(\det U)(-1)^l = (-1)^{k+l}\prod_{i=1}^n \upsilon_{ii}$. The running time follows from Theorem 11.15. □

Corollary 11.17 *Let $A \in \mathbb{R}^{n\times n}$ be given. Then one can compute A^{-1} in $O(n^3)$ steps or decide that A is singular.*

Proof By Theorem 11.15 we can, using Algorithm 11.14, compute a fully pivotized LU decomposition in $O(n^3)$ steps. If the rank of A is n (i.e. A is nonsingular), then we can, by Theorem 11.13, solve the systems $Ax = e_i$ for $i = 1, \dots, n$ in $O(n^2)$ steps. The columns of A^{-1} are the solution vectors. □

11.3 Gaussian Elimination with Rational Numbers

We have until now assumed that we could implement the Gaussian Elimination Method exactly when using real numbers. This is, of course, not the case. In practice there are two choices: exact computation with rational numbers or computing with machine numbers with the implicit acceptance of rounding errors.

Consider the first choice. We assume that the input is rational, i.e. $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^m$. Then clearly every number appearing in the computational steps is rational, because only the basic operations of arithmetic are used. But it is not at all clear that the number of bits needed for the exact storage of all numbers occurring in the algorithm does not grow faster than polynomially (one could, for example, apply n multiplications to the number 2 to compute the number 2^{2^n}, whose binary representation requires $2^n + 1$ bits).

Example 11.18 The numbers occurring in the algorithm can certainly grow exponentially, as the following matrix shows:

$$A_n := \begin{pmatrix} 1 & -1 & 0 & \cdots & 0 & 0 & 2 \\ -1 & 2 & -1 & \ddots & 0 & 0 & 2 \\ -1 & 0 & 2 & \ddots & 0 & 0 & 2 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ -1 & 0 & 0 & \ddots & 2 & -1 & 2 \\ -1 & 0 & 0 & \cdots & 0 & 2 & 2 \\ -1 & 0 & 0 & \cdots & 0 & 0 & 2 \end{pmatrix} \in \mathbb{Z}^{n \times n}$$

Its LU decomposition is

$$A_n = \begin{pmatrix} 1 & 0 & & & \cdots & & 0 \\ -1 & 1 & 0 & & & & \\ & -1 & 1 & \ddots & & & \vdots \\ & & \ddots & \ddots & \ddots & & \\ \vdots & & & \ddots & 1 & 0 & \\ & & & & -1 & 1 & 0 \\ -1 & & & \cdots & & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 0 & \cdots & & 0 & 2 \\ 0 & 1 & -1 & \ddots & & 0 & 4 \\ & 0 & 1 & \ddots & & 0 & 8 \\ \vdots & & \ddots & \ddots & \ddots & \vdots & \vdots \\ & & & \ddots & 1 & -1 & 2^{n-2} \\ & & & & 0 & 1 & 2^{n-1} \\ 0 & & & \cdots & & 0 & 2^n \end{pmatrix}.$$

Edmonds [10] showed, however, that the Gaussian Elimination Method is in fact a polynomial (and thus also a strongly polynomial) algorithm.

We will show for a start that the binary representations of the numerator and denominator of the determinant of a square rational matrix require at most double the number of bits as all the entries of the matrix together:

Lemma 11.19 *Let $n \in \mathbb{N}$ and $A = (\alpha_{ij}) \in \mathbb{Q}^{n\times n}$ with $\alpha_{ij} = \frac{p_{ij}}{q_{ij}}$, $p_{ij} \in \mathbb{Z}$ and $q_{ij} \in \mathbb{N}$. Let $k := \sum_{i,j=1}^{n} \left(\lceil \log_2(|p_{ij}| + 1)\rceil + \lceil \log_2(q_{ij})\rceil\right)$. Then there exist $p \in \mathbb{Z}$ and $q \in \mathbb{N}$ with $\det A = \frac{p}{q}$ and $\lceil \log_2(|p| + 1)\rceil + \lceil \log_2(q)\rceil \leq 2k$.*

Proof Put $q := \prod_{i,j=1}^{n} q_{ij}$ and $p := q \cdot \det A$. Then $\log_2 q = \sum_{i,j=1}^{n} \log_2 q_{ij} < k$ and $p \in \mathbb{Z}$. By the Laplace Expansion Theorem and induction over n one obtains $|\det A| \leq \prod_{i=1}^{n} \sum_{j=1}^{n} |\alpha_{ij}|$. It then follows that $|\det A| \leq \prod_{i=1}^{n} \sum_{j=1}^{n} |p_{ij}| < \prod_{i,j=1}^{n}(1 + |p_{ij}|)$, hence $\log_2 |p| = \log_2 q + \log_2 |\det A| < \log_2 q + \log_2 \prod_{i,j=1}^{n}(1 + |p_{ij}|) = \sum_{i,j=1}^{n}(\log_2 q_{ij} + \log_2(1 + |p_{ij}|)) < k$. $\square$

Example 11.18 shows that the bound is tight up to a constant factor.

We will use this upper bound for subdeterminants of the given matrix $A \in \mathbb{Q}^{m\times n}$; these are the determinants of submatrices $A_{IJ} := (\alpha_{ij})_{i\in I, j\in J}$ with $I \subseteq \{1,\dots,m\}$, $J \subseteq \{1,\dots,n\}$ and $|I| = |J| \neq 0$. The reason is given in the following lemma:

Lemma 11.20 *All the numbers occurring while Gaussian Elimination (Algorithm 11.14) runs are either 0 or 1 or entries of A or (up to a sign) quotients of subdeterminants of A.*

Proof It suffices to consider a particular iteration r. Here entries υ_{ij} of U with $i > r$ and $j \geq r$ are altered. If $j = r$, the new entry is zero. If not, then $i > r$ and $j > r$, and at the end of the iteration we have

$$\upsilon_{ij} = \frac{\det U_{\{1,\dots,r,i\}\{1,\dots,r,j\}}}{\det U_{\{1,\dots,r\}\{1,\dots,r\}}}, \tag{11.2}$$

which one obtains with the Laplace Expansion Theorem applied to the last row (with index i) of the matrix appearing in the numerator; this row has at most one nonvanishing entry, namely υ_{ij}.

The values of υ_{ir} and υ_{rr} used for calculating λ_{ir} are, when $r = 1$, entries of A and therefore subdeterminants of A. When $r > 1$, they are altered for the last time in iteration $r - 1$. With (11.2) we thus have

$$\lambda_{ir} = \frac{\det U_{\{1,\dots,r-1,i\}\{1,\dots,r\}}}{\det U_{\{1,\dots,r-1\}\{1,\dots,r-1\}}} \Big/ \frac{\det U_{\{1,\dots,r\}\{1,\dots,r\}}}{\det U_{\{1,\dots,r-1\}\{1,\dots,r-1\}}} = \frac{\det U_{\{1,\dots,r-1,i\}\{1,\dots,r\}}}{\det U_{\{1,\dots,r\}\{1,\dots,r\}}}. \tag{11.3}$$

We will now show that the numerators and denominators of the quotients occurring in (11.2) and (11.3) are not only subdeterminants of U but also of A.

Because of the invariant $P_\sigma^\top LUP_\tau = A$ we have that $LU = P_\sigma AP_\tau^\top$ always holds. Thus we have at the end of iteration r:

$$\begin{aligned}(P_\sigma AP_\tau^\top)_{\{1,\dots,r,i\}\{1,\dots,r,j\}} &= (LU)_{\{1,\dots,r,i\}\{1,\dots,r,j\}} \\ &= L_{\{1,\dots,r,i\}\{1,\dots,m\}}U_{\{1,\dots,m\}\{1,\dots,r,j\}} \\ &= L_{\{1,\dots,r,i\}\{1,\dots,r,i\}}U_{\{1,\dots,r,i\}\{1,\dots,r,j\}},\end{aligned}$$

where the last equation holds because the columns of L beyond the first r are still the columns of the identity matrix. Then for all $i > r$ and all $j > r$:

$$\begin{aligned}\det\big(P_\sigma AP_\tau^\top\big)_{\{1,\dots,r,i\}\{1,\dots,r,j\}} &= \det L_{\{1,\dots,r,i\}\{1,\dots,r,i\}} \det U_{\{1,\dots,r,i\}\{1,\dots,r,j\}} \\ &= \det U_{\{1,\dots,r,i\}\{1,\dots,r,j\}}\end{aligned}$$

and analogously

$$\det\big(P_\sigma AP_\tau^\top\big)_{\{1,\dots,r\}\{1,\dots,r\}} = \det U_{\{1,\dots,r\}\{1,\dots,r\}}.$$

In each case the left-hand side of this equation is (up to a sign) a subdeterminant of A. □

We thus have:

Theorem 11.21 *The Gaussian Elimination Method (Algorithm 11.14) is a polynomial algorithm.*

Proof By Lemma 11.20, all the numbers occurring in the algorithm are 0 or 1 or entries of A or (up to a sign) quotients of subdeterminants of A. By Lemma 11.19 every subdeterminant of A can be stored with $2k$ bits and thus a quotient with at most $4k$ bits, where k is the number of input bits. In order that these numbers do not in fact use more memory, all fractions must be reduced to lowest terms. By Corollary 3.13 this can be done with the Euclidean Algorithm in polynomial time. □

11.4 Gaussian Elimination with Machine Numbers

Computing with rational numbers with arbitrarily large numerators and denominators is rather slow. In practice, therefore, one usually solves systems of linear equations using machine numbers like the data type `double` and accepts the occurrence of rounding errors. Example 5.5 and the following example show that in the Gaussian Elimination Method rounding errors can be amplified by cancellation and thus lead to totally false results.

Example 11.22 The matrix $A = \begin{pmatrix} 2^{-k} & -1 \\ 1 & 1 \end{pmatrix}$ has LU decomposition

$$\begin{pmatrix} 2^{-k} & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 2^k & 1 \end{pmatrix} \begin{pmatrix} 2^{-k} & -1 \\ 0 & 2^k + 1 \end{pmatrix} .$$

For $k \in \mathbb{N}$ with $k > 52$, computing in $F_{\texttt{double}}$ leads to the following rounded result:

$$\begin{pmatrix} 2^{-k} & -1 \\ 1 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & 0 \\ 2^k & 1 \end{pmatrix} \begin{pmatrix} 2^{-k} & -1 \\ 0 & 2^k \end{pmatrix} = \begin{pmatrix} 2^{-k} & -1 \\ 1 & 0 \end{pmatrix} ;$$

i.e. the rounded LU decomposition yields a product which multiplies out to a matrix that is totally different from A. Partial pivotization does somewhat better: interchanging the first and second rows yields

$$\begin{pmatrix} 2^{-k} & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 2^{-k} & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -1 - 2^{-k} \end{pmatrix} ,$$

which is nearly correct, even after rounding.

Thus the Gaussian Elimination Method without appropriate pivot search is numerically unstable. One therefore tries to achieve numerical stability by a clever choice of the pivot element. One of the suggested strategies, for example, is to choose an element υ_{pq} (preferably with $q = r + 1$, i.e. without column interchange) with the property that $\frac{|\upsilon_{pq}|}{\max_j |\upsilon_{pj}|}$ is as large as possible. But one has not been able to prove rigorously that any particular pivot strategy always leads to numerical stability. At least Wilkinson [37] was able to give bounds for the backward stability, leading to a partial explanation of the usually good performance in practice. From bounds for the backward stability and for the condition one can derive bounds for the forward stability, as we will elucidate below.

Here we will only give the backward analysis for the simpler second phase of the Gaussian Elimination Method (Lemma 11.6):

Theorem 11.23 *Let $A = (\alpha_{ij}) \in \mathbb{R}^{m\times n}$ be an upper triangular matrix and $b = (\beta_1, \dots, \beta_m)^\top \in \mathbb{R}^m$, where the entries of A and b are machine numbers in a machine number domain F. Let F have the machine epsilon* $\mathrm{eps}(F) < \frac{1}{3}$*. Suppose that for all $i = 1, \dots, m$ we have either ($i \le n$ and $\alpha_{ii} \neq 0$) or ($\beta_i = 0$ and $\alpha_{ij} = 0$ for all $j = 1, \dots, n$). If one applies the method of Lemma 11.6 using machine arithmetic and if the absolute values of all intermediate results are equal to zero or in* $\mathrm{range}(F)$*, then one obtains a vector $\tilde{x}$ for which an upper triangular matrix $\tilde{A} = (\tilde{\alpha}_{ij})$ exists with $\tilde{A}\,\tilde{x} = b$ and*

$$|\alpha_{ij} - \tilde{\alpha}_{ij}| \le \max\{3, n\}\,\mathrm{eps}\,(1 + \mathrm{eps})^{n-1}|\alpha_{ij}|$$

for $i = 1, \dots, m$ and $j = 1, \dots, n$.

Proof We will successively consider the indices $i = \min\{m, n\}, \dots, 1$ with $\alpha_{ii} \neq 0$. With the formula given in (11.1) we have:

$$\tilde{\xi}_i = \mathrm{rd}\left(\frac{\mathrm{rd}\,(\beta_i - s_{i+1})}{\alpha_{ii}}\right),$$

where

$$s_k = \mathrm{rd}\left(\mathrm{rd}(\alpha_{ik}\tilde{\xi}_k) + s_{k+1}\right) \tag{11.4}$$

($k = i + 1, \dots, n$), $s_{n+1} := 0$ and rd is a rounding to F.

Claim: Let $k \in \{i + 1, \dots, n + 1\}$. Then there exists $\widehat{\alpha}_{ij} \in \mathbb{R}$ ($j = k, \dots, n$) with $s_k = \sum_{j=k}^{n} \widehat{\alpha}_{ij}\tilde{\xi}_j$ and $|\widehat{\alpha}_{ij} - \alpha_{ij}| \le (n+2-k)\,\mathrm{eps}(1 + \mathrm{eps})^{n+1-k}|\alpha_{ij}|$ for $j = k, \dots, n$.

We will prove this claim by induction over $n + 1 - k$. For $k = n + 1$ there is nothing to show.

For $k \le n$ we have by the induction hypothesis that $s_{k+1} = \sum_{j=k+1}^{n} \widehat{\widehat{\alpha}}_{ij}\tilde{\xi}_j$ with $|\widehat{\widehat{\alpha}}_{ij} - \alpha_{ij}| \le (n+1-k)\,\mathrm{eps}(1+\mathrm{eps})^{n-k}|\alpha_{ij}|$ for $j = k+1, \ldots, n$. Let rd be a rounding to F. For suitably chosen $\epsilon_1, \epsilon_2 \in \mathbb{R}$ with $|\epsilon_1|, |\epsilon_2| \le \mathrm{eps}$ (cf. Definition 4.7) Eq. (11.4) implies

$$\begin{aligned}
s_k &= \mathrm{rd}\left(\mathrm{rd}(\alpha_{ik}\tilde{\xi}_k) + s_{k+1}\right) \\
&= \left((\alpha_{ik}\tilde{\xi}_k)(1+\epsilon_1) + s_{k+1}\right)(1+\epsilon_2) \\
&= \left((\alpha_{ik}\tilde{\xi}_k)(1+\epsilon_1) + \sum_{j=k+1}^{n} \widehat{\widehat{\alpha}}_{ij}\tilde{\xi}_j\right)(1+\epsilon_2) \\
&= \alpha_{ik}(1+\epsilon_1)(1+\epsilon_2)\tilde{\xi}_k + \sum_{j=k+1}^{n} \widehat{\widehat{\alpha}}_{ij}(1+\epsilon_2)\tilde{\xi}_j
\end{aligned}$$

We define $\widehat{\alpha}_{ik} := \alpha_{ik}(1+\epsilon_1)(1+\epsilon_2)$ and $\widehat{\alpha}_{ij} := \widehat{\widehat{\alpha}}_{ij}(1+\epsilon_2)$ for $j = k+1, \ldots, n$. Then we get $|\widehat{\alpha}_{ik} - \alpha_{ik}| \le |\alpha_{ik}||\epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2| \le |\alpha_{ik}|(2\,\mathrm{eps} + \mathrm{eps}^2) < |\alpha_{ik}|2\,\mathrm{eps}(1+\mathrm{eps})$.

For $j = k+1, \ldots, n$ we now calculate

$$\begin{aligned}
|\widehat{\alpha}_{ij} - \alpha_{ij}| &= |(\widehat{\widehat{\alpha}}_{ij} - \alpha_{ij})(1+\epsilon_2) + \epsilon_2\alpha_{ij}| \\
&\le |\widehat{\widehat{\alpha}}_{ij} - \alpha_{ij}|(1+|\epsilon_2|) + |\epsilon_2||\alpha_{ij}| \\
&\le (n+1-k)\,\mathrm{eps}(1+\mathrm{eps})^{n-k}|\alpha_{ij}|(1+|\epsilon_2|) + |\epsilon_2||\alpha_{ij}| \\
&\le \left((n+1-k)\,\mathrm{eps}(1+\mathrm{eps})^{n+1-k} + \mathrm{eps}\right)|\alpha_{ij}| \\
&< \left((n+2-k)\,\mathrm{eps}(1+\mathrm{eps})^{n+1-k}\right)|\alpha_{ij}|
\end{aligned}$$

This proves the claim.

For $k = i+1$ the claim implies that there exists $\tilde{\alpha}_{ij} \in \mathbb{R}$ $(j = i+1, \ldots, n)$ with $s_{i+1} = \sum_{j=i+1}^{n} \tilde{\alpha}_{ij}\tilde{\xi}_j$ and $|\tilde{\alpha}_{ij} - \alpha_{ij}| \le (n+1-i)\,\mathrm{eps}(1+\mathrm{eps})^{n-i}|\alpha_{ij}|$ for $j = i+1, \ldots, n$.

For $i = \min\{m, n\}, \ldots, 1$ with $\alpha_{ii} \ne 0$ we have for suitably chosen $\epsilon_1, \epsilon_2 \in \mathbb{R}$ with $|\epsilon_1|, |\epsilon_2| \le \mathrm{eps}$:

$$\tilde{\xi}_i = \mathrm{rd}\left(\frac{\mathrm{rd}\,(\beta_i - s_{i+1})}{\alpha_{ii}}\right) = \left(\frac{(\beta_i - s_{i+1})\,(1+\epsilon_1)}{\alpha_{ii}}\right)(1+\epsilon_2)\,.$$

So in addition we define $\tilde{\alpha}_{ii} := \alpha_{ii}/((1+\epsilon_1)(1+\epsilon_2))$ and, because $1 - |\epsilon| \le \frac{1}{1+\epsilon} \le 1 + \frac{3}{2}|\epsilon|$ for $\epsilon \in \mathbb{R}$ with $|\epsilon| \le \frac{1}{3}$, we obtain $|\tilde{\alpha}_{ii} - \alpha_{ii}| \le ((1 + \frac{3}{2}\,\mathrm{eps})(1 + \frac{3}{2}\,\mathrm{eps}) - 1)|\alpha_{ii}| < 3\,\mathrm{eps}(1+\mathrm{eps})|\alpha_{ii}|$, as desired. If $i = n = 1$, then $\epsilon_1 = 0$ and therefore even $|\tilde{\alpha}_{ii} - \alpha_{ii}| \leq \frac{3}{2}\,\mathrm{eps}\,|\alpha_{ii}|$. It also follows that

$$\tilde{\xi}_i = \frac{\beta_i - \sum_{j=i+1}^{n} \tilde{\alpha}_{ij}\tilde{\xi}_j}{\tilde{\alpha}_{ii}} .$$

We now have from the above that $\tilde{A}\,\tilde{x} = b$. □

For $F_{\texttt{double}}$ and all practically relevant sizes of n the term $(1+\mathrm{eps})^{n-1}$ is so close to 1 that it can safely be neglected. For $F_{\texttt{double}}$ and $n = 10^6$ we have, for example, that $|\alpha_{ij} - \tilde{\alpha}_{ij}| \leq 2^{-33}|\alpha_{ij}|$. Thus the second phase of the Gaussian Elimination Method is backward stable.

Actually, however, one is more interested in the relative error of $\tilde{x}$. One can derive it with the help of the condition from the backward stability, as we will show below. For this we will, however, need some further theory.

11.5 Matrix Norms

In order to generalize the definition of condition number so that it is applicable to higher dimensional problems, we require norms. Mappings can decrease or increase errors. Here we will focus on linear mappings which are relevant for the solution of systems of linear equations.

Definition 11.24 Let V be a vector space over $\mathbb{R}$ (e.g. $V = \mathbb{R}^n$ or $V = \mathbb{R}^{m\times n}$; similarly with $\mathbb{R}$ replaced by $\mathbb{C}$). Then a **norm** on V is a mapping $\|\cdot\|: V \to \mathbb{R}$ with

- $\|x\| > 0$ for all $x \neq 0$;
- $\|\alpha x\| = |\alpha| \cdot \|x\|$ for all $\alpha \in \mathbb{R}$ and all $x \in V$;
- $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in V$ (triangle inequality).

A **matrix norm** is a norm on $\mathbb{R}^{m\times n}$.

Example 11.25 The following vector norms (for $x = (\xi_i) \in \mathbb{R}^n$) are well-known:

- $\|x\|_1 := \sum_{i=1}^{n} |\xi_i|$ (absolute value sum norm, ℓ_1 norm, Manhattan norm);
- $\|x\|_\infty := \max_{1\leq i\leq n} |\xi_i|$ (maximum norm, ℓ_∞ norm).

Well-known matrix norms (for $A = (\alpha_{ij}) \in \mathbb{R}^{m\times n}$) are, for example:

- $\|A\|_1 := \max_{1\leq j\leq n} \sum_{i=1}^{m} |\alpha_{ij}|$ (maximum absolute column sum norm);
- $\|A\|_\infty := \max_{1\leq i\leq m} \sum_{j=1}^{n} |\alpha_{ij}|$ (maximum absolute row sum norm).

We will from this point on consider only square matrices.

Definition 11.26 A matrix norm $\|\cdot\|^{\mathrm{M}}$ on $\mathbb{R}^{n\times n}$ is called **compatible** with the vector norm $\|\cdot\|$ on $\mathbb{R}^n$ if the following holds for all $A \in \mathbb{R}^{n\times n}$ and all $x \in \mathbb{R}^n$:

$$\|Ax\| \;\le\; \|A\|^{\mathrm{M}} \cdot \|x\| \,.$$

A matrix norm $\|\cdot\|\colon \mathbb{R}^{n\times n} \to \mathbb{R}$ is called **submultiplicative**, if

$$\|AB\| \;\le\; \|A\| \cdot \|B\|$$

for all $A, B \in \mathbb{R}^{n\times n}$.

Not all matrix norms $\|\cdot\| : \mathbb{R}^{n\times n} \to \mathbb{R}$ are submultiplicative: e.g. $\|A\| := \max_{1\le i,j,\le n} |a_{ij}|$ defines a matrix norm with $\left\|\left(\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right)\left(\begin{smallmatrix}1&0\\1&0\end{smallmatrix}\right)\right\| = \left\|\left(\begin{smallmatrix}2&0\\0&0\end{smallmatrix}\right)\right\| = 2 > 1\cdot 1 = \left\|\left(\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right)\right\| \cdot \left\|\left(\begin{smallmatrix}1&0\\1&0\end{smallmatrix}\right)\right\|$. Moreover, this matrix norm is also not compatible with the vector norm $\|\cdot\|_\infty$, as the example $\left\|\left(\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right)\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\right\|_\infty = \left\|\left(\begin{smallmatrix}2\\0\end{smallmatrix}\right)\right\|_\infty = 2 > 1\cdot 1 = \left\|\left(\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right)\right\| \cdot \left\|\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\right\|_\infty$ shows.

Theorem 11.27 *Let $\|\cdot\|$ be a vector norm on $\mathbb{R}^n$. Then the mapping $|||\cdot|||\colon \mathbb{R}^{n\times n} \to \mathbb{R}$ defined by*

$$|||A||| \;:=\; \max\left\{ \frac{\|Ax\|}{\|x\|} \,\middle|\, x \in \mathbb{R}^n \setminus \{0\} \right\} \;=\; \max\bigl\{\|Ax\| : x \in \mathbb{R}^n,\ \|x\| = 1\bigr\}$$

for all $A \in \mathbb{R}^{n\times n}$ is a submultiplicative matrix norm compatible with $\|\cdot\|$.

Proof We will first show that $|||\cdot|||$ is a norm:

- if $A \neq 0$, then clearly there exists an entry $\alpha_{ij} \neq 0$ and thus $Ae_j \neq 0$, so that $\|Ae_j\| > 0$. It follows that $|||A||| = \max_{x\neq 0} \frac{\|Ax\|}{\|x\|} \ge \frac{\|Ae_j\|}{\|e_j\|} > 0$.
- For all $\alpha \in \mathbb{R}$ and $A \in \mathbb{R}^{n\times n}$ we have $|||\alpha A||| = \max_{x\neq 0} \frac{\|\alpha Ax\|}{\|x\|} = |\alpha| \max_{x\neq 0} \frac{\|Ax\|}{\|x\|} = |\alpha| \cdot |||A|||$.
- For all $A, B \in \mathbb{R}^{n\times n}$ we have

$$\begin{aligned}
|||A+B||| &= \max_{\|x\|=1} \|(A+B)x\| \\
&\le \max_{\|x\|=1} (\|Ax\| + \|Bx\|) \\
&\le \max_{\|x\|=1} \|Ax\| + \max_{\|x\|=1} \|Bx\| \\
&= |||A||| + |||B||| \,.
\end{aligned}$$

Moreover, $|||\cdot|||$ is submultiplicative, because for all $A, B \in \mathbb{R}^{n\times n}$ with $B \neq 0$ we have

$$\begin{aligned}|||AB||| &= \max_{x\neq 0} \frac{\|ABx\|}{\|x\|} \\ &= \max_{Bx\neq 0} \left(\frac{\|A(Bx)\|}{\|Bx\|} \cdot \frac{\|Bx\|}{\|x\|} \right) \\ &\leq \max_{y\neq 0} \frac{\|Ay\|}{\|y\|} \cdot \max_{x\neq 0} \frac{\|Bx\|}{\|x\|} \\ &= |||A||| \cdot |||B||| \ .\end{aligned}$$

Finally, $|||\cdot|||$ is compatible with $\|\cdot\|$, because for all $A \in \mathbb{R}^{n\times n}$ and all $x \in \mathbb{R}^n$ we have

$$|||A\,||| \cdot \|x\| \ = \ \max_{y\neq 0} \frac{\|Ay\|}{\|y\|}\|x\| \ \geq \ \|Ax\| \, .$$

□

The matrix norm $|||\cdot|||$ defined in Theorem 11.27 is called the norm **induced** by $\|\cdot\|$. The matrix norm induced by a particular vector norm is obviously the least of all the matrix norms compatible with this vector norm.

Example 11.28 The ℓ_1 norm $x \mapsto \|x\|_1$ ($x \in \mathbb{R}^n$) induces the maximum absolute column sum norm $A \mapsto \|A\|_1$ ($A \in \mathbb{R}^{n\times n}$) because the following always holds:

$$\max\{\|Ax\|_1 : \|x\|_1 = 1\} = \max\left\{ \sum_{i=1}^{n} \left| \sum_{j=1}^{n} \alpha_{ij}\xi_j \right| \ : \sum_{j=1}^{n} |\xi_j| = 1 \right\} = \max_{1\leq j\leq n} \sum_{i=1}^{n} |\alpha_{ij}|$$

(the first maximum is attained by a unit vector).

The norm $x \mapsto \|x\|_\infty$ induces the maximum absolute row sum norm $A \mapsto \|A\|_\infty$ because the following always holds:

$$\max\{\|Ax\|_\infty : \|x\|_\infty = 1\} = \max\left\{ \max_{1\leq i\leq n} \left| \sum_{j=1}^{n} \alpha_{ij}\xi_j \right| \ : |\xi_j| \leq 1 \ \forall j \right\} = \max_{1\leq i\leq n} \sum_{j=1}^{n} |\alpha_{ij}| \, .$$

11.6 The Condition of Systems of Linear Equations

We remind the reader of the definition of the condition number of a numerical computational problem given in Sect. 5.3. Here we generalize Definition 5.6 to multidimensional problems by replacing every absolute value by a norm. Obviously the condition number now depends on the chosen norm.

To keep things simple we will only consider systems of linear equations $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ nonsingular; then the problem is uniquely defined: we have to find the unique solution $x = A^{-1}b$.

We will initially keep the matrix A fixed and consider only b as input. Then we have:

Proposition 11.29 *Let $A \in \mathbb{R}^{n\times n}$ be nonsingular and $\|\cdot\|: \mathbb{R}^n \to \mathbb{R}$ a fixed vector norm. Then the condition number of the problem $b \mapsto A^{-1}b$ (with respect to this norm) is*

$$\kappa(A) \;:=\; \|A^{-1}\| \cdot \|A\| \,,$$

where the notation $\|\cdot\|$ also denotes the matrix norm induced by the vector norm $\|\cdot\|$.

Proof We calculate the condition number according to the definition (see above):

$$\begin{aligned}
&\sup\left\{ \lim_{\epsilon\to 0}\ \sup\left\{ \frac{\frac{\|A^{-1}b-A^{-1}b'\|}{\|A^{-1}b\|}}{\frac{\|b-b'\|}{\|b\|}} \;:\; b'\in\mathbb{R}^n,\ 0<\|b-b'\|<\epsilon \right\} \;:\; b\in\mathbb{R}^n, b\neq 0\right\}\\
&= \sup\left\{ \frac{\|A^{-1}(b-b')\|}{\|b-b'\|}\cdot\frac{\|A(A^{-1}b)\|}{\|A^{-1}b\|} \;:\; b,b'\in\mathbb{R}^n,\ b\neq 0, b'\neq b\right\}\\
&= \sup\left\{ \frac{\|A^{-1}x\|}{\|x\|} \;:\; x\neq 0\right\}\cdot\sup\left\{\frac{\|Ay\|}{\|y\|} \;:\; y\neq 0\right\}\\
&= \|A^{-1}\|\cdot\|A\| \,.
\end{aligned}$$

□

$\kappa(A)$ is also called the **condition number of the matrix** A. As $\|A^{-1}\| \cdot \|A\| \geq \|A^{-1}A\| = \|I\| = 1$, it cannot be less than 1.

If we know the condition number of A (or at least an upper bound), then we can estimate how exact an approximate solution $\tilde{x}$ is: we calculate the residual vector $r := A\tilde{x} - b$ and then get, setting $\tilde{x} = A^{-1}(b + r)$ and $x = A^{-1}b$,

$$\frac{\|x-\tilde{x}\|}{\|x\|} = \frac{\|A^{-1}b - A^{-1}(b+r)\|}{\|x\|} = \frac{\|A^{-1}r\|\cdot\|Ax\|}{\|b\|\cdot\|x\|} \leq \frac{\|A^{-1}\|\cdot\|r\|}{\|b\|}\|A\| = \kappa(A)\frac{\|r\|}{\|b\|} \,.$$

If the error is too large, one can try a post-iteration step: one solves (also approximately) the system $Ax = r$ and subtracts the solution vector from $\tilde{x}$.

Example 11.30 Consider the system of linear equations given in Example 5.5:

$$\begin{pmatrix} 10^{-20} & 2 \\ 10^{-20} & 10^{-20} \end{pmatrix} \begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 10^{-20} \end{pmatrix}.$$

Gaussian Elimination (without pivoting) using machine arithmetic in $F_{\texttt{double}}$ yields the approximate solution $\xi_2 = \frac{1}{2}$ and $\xi_1 = 0$. This has the very small residual vector

$$\begin{pmatrix} 10^{-20} & 2 \\ 10^{-20} & 10^{-20} \end{pmatrix} \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} - \begin{pmatrix} 1 \\ 10^{-20} \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{1}{2} \cdot 10^{-20} \end{pmatrix},$$

which, however, does not imply that the solution is necessarily a good one because the condition number of the matrix is very poor:

$$\kappa \begin{pmatrix} 10^{-20} & 2 \\ 10^{-20} & 10^{-20} \end{pmatrix} = \left\| \begin{pmatrix} 10^{-20} & 2 \\ 10^{-20} & 10^{-20} \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} \frac{-1}{2-10^{-20}} & \frac{2\cdot 10^{20}}{2-10^{-20}} \\ \frac{1}{2-10^{-20}} & \frac{-1}{2-10^{-20}} \end{pmatrix} \right\| \approx 2 \cdot 10^{20}.$$

Multiplying the second row with 10^{20} improves the condition number a great deal:

$$\kappa \begin{pmatrix} 10^{-20} & 2 \\ 1 & 1 \end{pmatrix} = \left\| \begin{pmatrix} 10^{-20} & 2 \\ 1 & 1 \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} \frac{-1}{2-10^{-20}} & \frac{2}{2-10^{-20}} \\ \frac{1}{2-10^{-20}} & \frac{-10^{-20}}{2-10^{-20}} \end{pmatrix} \right\| \approx 2$$

(using the maximum absolute row sum norm in both cases). This, however, results in a large residual vector:

$$\begin{pmatrix} 10^{-20} & 2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{1}{2} \end{pmatrix}.$$

A good condition does, however, help when using backward stable methods. If we in general wish to solve an instance I of a numerical computational problem and if we know that the approximate solution $\tilde{x}$ found by a particular algorithm is the correct solution for an instance $\tilde{I}$ and that $\frac{\|I-\tilde{I}\|}{\|I\|}$ is small (backward stability), then the relative error $\frac{\|\tilde{x}-x\|}{\|x\|}$ is given to a first approximation by $\kappa(I)\frac{\|I-\tilde{I}\|}{\|I\|}$. This assumes a (clearly not always given) linear behavior and otherwise is approximately correct for sufficiently small errors only. Here one also speaks of linearized error theory.

Theorem 11.23 provides an example. Here, however, the matrix is perturbed, not the right-hand side. Then the condition of the instance is not (as in Proposition 11.29) simply the condition of the matrix. We require the following more general statement.

Theorem 11.31 *Let $A, \tilde{A} \in \mathbb{R}^{n\times n}$ be given with A nonsingular and $\|\tilde{A}-A\|\,\|A^{-1}\| < 1$ (with respect to a fixed vector norm and the induced matrix norm). Then $\tilde{A}$ is nonsingular.*

In addition, let $b, \tilde{b} \in \mathbb{R}^n$ be given with $b \neq 0$. Then the following holds for $x := A^{-1}b$ and $\tilde{x} := \tilde{A}^{-1}\tilde{b}$:

$$\frac{\|\tilde{x}-x\|}{\|x\|} \le \frac{\kappa(A)}{1-\|\tilde{A}-A\|\,\|A^{-1}\|}\left(\frac{\|\tilde{b}-b\|}{\|b\|}+\frac{\|\tilde{A}-A\|}{\|A\|}\right).$$

Proof The following holds for all $y \in \mathbb{R}^n$:

$$\begin{aligned}\|A^{-1}\|\cdot\|\tilde{A}y\| &\ge \|A^{-1}\tilde{A}y\| \\ &= \|y + A^{-1}(\tilde{A}-A)y\| \\ &\ge \|y\| - \|A^{-1}(\tilde{A}-A)y\| \\ &\ge \|y\| - \|A^{-1}\|\cdot\|\tilde{A}-A\|\cdot\|y\| \\ &= \left(1-\|\tilde{A}-A\|\,\|A^{-1}\|\right)\|y\|\,.\end{aligned} \tag{11.5}$$

By the assumption the right-hand side is positive for all $y \neq 0$, therefore $\|\tilde{A}y\| > 0$ for all $y \neq 0$ and hence $\tilde{A}$ is nonsingular.

Setting $y = \tilde{x} - x$ in (11.5) and using

$$\begin{aligned}\|\tilde{A}(\tilde{x}-x)\| &= \|\tilde{b}-\tilde{A}x\| \\ &= \|(\tilde{b}-b)-(\tilde{A}-A)x\| \\ &\le \|\tilde{b}-b\| + \|\tilde{A}-A\|\cdot\|x\| \\ &= \|A\|\left(\frac{\|\tilde{b}-b\|}{\|b\|}\cdot\frac{\|Ax\|}{\|A\|}+\frac{\|\tilde{A}-A\|}{\|A\|}\cdot\|x\|\right) \\ &\le \|A\|\left(\frac{\|\tilde{b}-b\|}{\|b\|}+\frac{\|\tilde{A}-A\|}{\|A\|}\right)\|x\|\,,\end{aligned}$$

we obtain

$$\|\tilde{x}-x\| \le \frac{\|A^{-1}\|}{1-\|\tilde{A}-A\|\,\|A^{-1}\|}\,\|A\|\left(\frac{\|\tilde{b}-b\|}{\|b\|}+\frac{\|\tilde{A}-A\|}{\|A\|}\right)\|x\|\,,$$

as desired. □

Theorems 11.23 and 11.31 together provide a very good estimate of the relative error in the solution which we obtain in the second phase of the Gaussian Elimination Method using machine numbers, at least for a matrix with a small condition number.

We have already seen in Example 11.30 that one can often improve the condition of a matrix by multiplying a row with a (nonzero) number. One can similarly multiply columns with nonzero constants. In general one chooses nonsingular diagonal matrices D_1 and D_2 such that D_1AD_2 has better condition than A; then one solves the system $(D_1AD_2)y = D_1b$ instead of $Ax = b$ and afterwards sets $x := D_2y$. This is called preconditioning.

The matrices D_1 and D_2 can in fact be any nonsingular matrices, but unfortunately it is not even clear when using diagonal matrices, how best to choose D_1 and D_2. Normally one tries to achieve that the sums of the absolute values of the entries of all the rows and columns of D_1AD_2 are similar in size; this is called equilibration. If one restricts preconditioning to using only multiplication on the left by a diagonal matrix (i.e. $D_2 = I$), then equal absolute value row sums do in fact yield the best possible condition with respect to the ℓ_∞ norm:

Theorem 11.32 *Let $A = (\alpha_{ij}) \in \mathbb{R}^{n\times n}$ be a nonsingular matrix with $\sum_{j=1}^{n} |\alpha_{ij}| = \sum_{j=1}^{n} |\alpha_{1j}|$ for $i = 1, \dots, n$. Then for every nonsingular diagonal matrix D we have, with respect to the ℓ_∞ norm*

$$\kappa(DA) \geq \kappa(A)\,.$$

Proof Let $\delta_1, \dots, \delta_n$ be the diagonal entries of D. Then

$$\|DA\|_\infty = \max_{1\leq i\leq n} |\delta_i| \sum_{j=1}^{n} |\alpha_{ij}| = \|A\|_\infty \max_{1\leq i\leq n} |\delta_i|\,.$$

Here we have used the fact that the row sums are all equal.

The diagonal entries of the inverse of D are $\frac{1}{\delta_1}, \dots, \frac{1}{\delta_n}$. Let $A^{-1} = (\alpha'_{ij})_{i,j=1,\dots,n}$. Then we have that:

$$\begin{aligned}\|(DA)^{-1}\|_\infty &= \|A^{-1}D^{-1}\|_\infty \\ &= \max_{1\leq i\leq n} \sum_{j=1}^{n} \frac{|\alpha'_{ij}|}{|\delta_j|} \geq \frac{\max_{1\leq i\leq n} \sum_{j=1}^{n} |\alpha'_{ij}|}{\max_{1\leq k\leq n} |\delta_k|} = \frac{\|A^{-1}\|_\infty}{\max_{1\leq k\leq n} |\delta_k|}\,.\end{aligned}$$

Multiplying these two expressions yields:

$$\kappa(DA) = \|DA\|_\infty \|A^{-1}D^{-1}\|_\infty \geq \|A\|_\infty \|A^{-1}\|_\infty = \kappa(A)\,. \qquad \square$$

In summary, Gaussian Elimination (where necessary with preconditioning and/or post-iteration) is a useful method for solving general systems of linear equations and related problems. If, however, one requires an exact solution, then one must compute accurately using rational numbers (Sect. 11.3). For systems with special properties there are other, often better methods available, which we cannot, however, treat in detail here.

Bibliography

[1] Agrawal M, Kayal N, Saxena N: PRIMES is in P. Annals of Mathematics 2004;160:781–93.

[2] Applegate DL, Bixby RE, Chvátal V, Cook WJ. The Traveling Salesman Problem. A Computational Study. Princeton University Press; 2006

[3] Bellman R. On a routing problem. Quarterly of Applied Mathematics 1958;16:87–90.

[4] Berge C. Two theorems in graph theory. Proceedings of the National Academy of Sciences of the United States of America 1957;43:842–44.

[5] C++ Standard. ISO/IEC 14882:2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[6] Church A. An unsolvable problem of elementary number theory. American Journal of Mathematics 1936;58:345–63.

[7] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3. ed. MIT Press; 2009.

[8] Dantzig GB, Fulkerson DR. On the max-flow min-cut theorem of networks. In: Kuhn HW, Tucker AW, eds. Linear Inequalities and Related Systems. Princeton University Press, Princeton; 1956:215–21.

[9] Dijkstra EW. A note on two problems in connexion with graphs. Numerische Mathematik 1959;1:269–71.

[10] Edmonds J. Systems of distinct representatives and linear algebra. Journal of Research of the National Bureau of Standards 1967;B71:241–5.

[11] Edmonds J, Karp RM. Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM 1972;19:248–64.

[12] Folkerts M. Die älteste lateinische Schrift über das indische Rechnen nach al-H̱āwarizmī. München: Verlag der Bayerischen Akademie der Wissenschaften; 1997.

[13] Ford LR. Network flow theory. Paper P-923, The Rand Corporation, Santa Monica 1956.

[14] Ford LR, Fulkerson, DR. Maximal flow through a network. Canadian Journal of Mathematics 1956;8:399–404.

[15] Ford LR, Fulkerson, DR. A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. Canadian Journal of Mathematics 1957;9:210–18.

[16] Frobenius G. Über zerlegbare Determinanten. Sitzungsberichte der Königlich Preussischen Akademie der Wissenschaften 1917;XVIII:274–77.

[17] Fürer M. Faster integer multiplication. SIAM Journal on Computing 2009;39:979–1005.

[18] Held M, Karp RM. A dynamic programming approach to sequencing problems. Journal of the SIAM 1962;10:196–210.

[19] IEEE Standard for Floating-Point Arithmetic. IEEE Computer Society, IEEE Std 754-2008. DOI 10.1109/IEEESTD.2008.4610935

[20] Jarník V. O jistém problému minimálním. Práce Moravské Přírodovědecké Společnosti 1930;6:57–63.

[21] Karatsuba A, Ofman Y. Multiplication of multidigit numbers on automata. Soviet Physics 1963;Doklady 7:595–6.

[22] Knuth DE. The Art of Computer Programming; vols. 1–4A. 3. ed. Addison-Wesley; 2011.
[23] König D. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. Mathematische Annalen 1916;77:453–65.
[24] Kruskal JB. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the AMS 1956;7:48–50.
[25] Lippman SB, Lajoie J, Moo BE. C++ Primer. 5. ed. Addison Wesley 2013
[26] Moore EF. The shortest path through a maze. Proceedings of an International Symposium on the Theory of Switching; Part II. Harvard University Press; 1959. S. 285–92.
[27] Oliveira e Silva T. Empirical verification of the 3x+1 and related conjectures. In: Lagarias JC, ed. The Ultimate Challenge: The 3x+1 Problem. American Mathematical Society 2010: 189–207.
[28] Petersen J. Die Theorie der regulären Graphs. Acta Mathematica 1891;15,193–220.
[29] Prim RC. Shortest connection networks and some generalizations. The Bell System Technical Journal 1957;36:1389–401.
[30] Schönhage A, Strassen V. Schnelle Multiplikation großer Zahlen. Computing 1971;7:281–92.
[31] Stroustrup B. The C++ Programming Language. 4. ed. Addison-Wesley; 2013.
[32] Stroustrup B. Programming: Principles and Practice Using C++. 2. ed. Addison Wesley 2014
[33] Tarjan RE. Data Structures and Network Algorithms. SIAM; 1983.
[34] Treiber D. Zur Reihe der Primzahlreziproken. Elemente der Mathematik 1995;50:164–6.
[35] Turing AM. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 1937;(2)42:230–65 and 43:544–6.
[36] Vogel K, ed. Mohammed ibn Musa Alchwarizmi's Algorismus. Das früheste Lehrbuch zum Rechnen mit indischen Ziffern. Aalen: Zeller; 1963.
[37] Wilkinson JH. Error analysis of direct methods of matrix inversion. Journal of the ACM 1961;8:281–330.

Index

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6