

# Entwurf



## KONZEPTE DER PROGRAMMIERUNG

Institut für Informatik

Jonas Cleve, Katharina Klost, Wolfgang Mulzer, Max Willert

Wintersemester 2024/25

## Preface

This is an evolving draft of the lecture notes for „Konzepte der Programmierung“, a class that is currently under development at the department of Computer Science of Freie Universität Berlin.

This is the second regular iteration of the class, taught in the winter semester 2024/25 by Wolfgang Mulzer. The first iteration was taught by Max Willert during the winter semester 2023/24.

Before the first regular iteration of the class, the „ProInformatik“<sup>1</sup> framework was used to test the material and to develop the class.

Man people have contributed to these notes. In particular, we would like to thank ....

### Purpose and target audience

This class is intended to be a first year introduction into basic notions of computer science. We will cover the major programming paradigms and features of modern programming languages, as well as fundamental ideas concerning algorithms and data structures.

This is a truly interdisciplinary class. It is attended by students from Computer Science, Bioinformatics, Computational Sciences, Business Informatics, Data Science, Mathematics, and beyond. This reflects the broad reach of modern computer science and shows the vitality of the field.

The class is taught in German, but the lecture notes are written in English. The reason is that we want to support those students who do not have German as their native language (and to familiarize students with the English terminology that is common in the field).

Since these lecture notes and the whole class are still under heavy development, we appreciate any feedback and ideas for improvements.

---

<sup>1</sup><https://pro.inf.fu-berlin.de/>

## Inhaltsverzeichnis

<b>Preface</b>	<b>i</b>
<b>I Introduction and Motivation</b>	<b>1</b>
<b>1 What is Computer Science?</b>	<b>2</b>
1.1 Overview . . . . .	2
1.2 Terminology . . . . .	5
1.3 Some historical notes . . . . .	5
1.4 Computer Science today . . . . .	7
1.5 Artificial Intelligence and Machine Learning . . . . .	7
<b>II Imperative Programming</b>	<b>10</b>
<b>2 Basics of Imperative Programming</b>	<b>11</b>
2.1 Von-Neumann-Architecture . . . . .	11
2.2 Programming languages . . . . .	13
2.2.1 Assembly languages . . . . .	13
2.2.2 High-level languages . . . . .	14
2.3 Expressions . . . . .	14
2.4 Imperative Programming . . . . .	16
2.4.1 Assignments . . . . .	18
2.4.2 Input and output . . . . .	19
2.4.3 Control flow . . . . .	20
2.4.4 Two programs . . . . .	24
<b>3 Data Types and Variables</b>	<b>29</b>
3.1 Primitive data types . . . . .	29
3.1.1 NoneType . . . . .	29
3.1.2 Boolean . . . . .	30
3.1.3 Integer . . . . .	30
3.1.4 Float . . . . .	32
3.1.5 Complex . . . . .	34
3.2 Composite data types . . . . .	34
3.2.1 Characters and Strings . . . . .	34
3.2.2 Lists . . . . .	35

3.2.3	Tuples . . . . .	40
3.2.4	Dictionaries . . . . .	40
3.3	Data representation . . . . .	42
3.3.1	Truth values . . . . .	43
3.3.2	Unsigned integers . . . . .	43
3.3.3	Signed integers . . . . .	44
3.3.4	Floating-point numbers . . . . .	47
3.3.5	Characters . . . . .	49
3.4	The five facets of a variable . . . . .	49
3.4.1	Content . . . . .	49
3.4.2	Location . . . . .	50
3.4.3	Size . . . . .	50
3.4.4	Data type . . . . .	50
3.4.5	Scope . . . . .	50
<b>4</b>	<b>Subroutines and Functions</b>	<b>52</b>
4.1	Fundamentals of subroutines . . . . .	52
4.1.1	Functions . . . . .	54
4.1.2	Scope . . . . .	55
4.1.3	Nested functions and closures . . . . .	57
4.2	Parameter passing strategies . . . . .	59
4.2.1	Call by value . . . . .	60
4.2.2	Call by reference . . . . .	61
4.2.3	Call by name . . . . .	63
4.3	Recursion . . . . .	67
4.4	The five steps for implementing a function . . . . .	70
4.4.1	Step 1: Give a function signature . . . . .	70
4.4.2	Step 2: Provide a specification . . . . .	71
4.4.3	Step 3: Design interesting test cases . . . . .	73
4.4.4	Step 4: Implement the function . . . . .	74
4.4.5	Step 5: Add helpful comments . . . . .	74
<b>5</b>	<b>Case Study: Autocomplete</b>	<b>76</b>
<b>III</b>	<b>Algorithms</b>	<b>78</b>
<b>6</b>	<b>Algorithmic Problems</b>	<b>79</b>
6.1	Finding the greatest common divisor . . . . .	80
6.2	Searching . . . . .	83
6.2.1	Linear search . . . . .	84
6.2.2	Binary search . . . . .	84

<b>7</b>	<b>Sorting</b>	<b>88</b>
7.1	The algorithmic problem of sorting . . . . .	88
7.2	Selection sort . . . . .	90
7.3	Insertion sort . . . . .	91
7.4	Merge sort . . . . .	92
7.5	Quicksort . . . . .	95
7.6	Stability . . . . .	100■
7.7	Memory usage . . . . .	101■
7.7.1	The hidden space of recursion . . . . .	102■
<b>8</b>	<b>Running Time</b>	<b>104■</b>
8.1	Fundamentals of running time . . . . .	104■
8.1.1	Linear search . . . . .	106■
8.1.2	Cartesian product . . . . .	107■
8.1.3	Binary search . . . . .	108■
8.2	O-notation . . . . .	109■
8.2.1	Typical asymptotic running times . . . . .	110■
8.3	Worst-case analysis of sorting algorithms . . . . .	111■
8.3.1	Selection sort . . . . .	111■
8.3.2	Insertion sort . . . . .	112■
8.3.3	Merge sort . . . . .	112■
8.3.4	Quicksort . . . . .	116■
8.4	Complexity of algorithmic problems . . . . .	118■
8.4.1	Decision trees . . . . .	120■
8.4.2	A lower bound for comparison-based sorting . . . . .	121■
<b>9</b>	<b>Correctness</b>	<b>124■</b>
9.1	Fundamentals of correctness . . . . .	124■
9.2	Correctness of simple linear programs . . . . .	126■
9.3	Correctness of programs with loops . . . . .	127■
9.4	Correctness of programs with recursion . . . . .	134■
9.5	Correctness of sorting algorithms . . . . .	139■
9.5.1	Selection sort . . . . .	139■
9.5.2	Quicksort . . . . .	142■
<b>IV</b>	<b>Functional Programming</b>	<b>147■</b>
<b>10</b>	<b>Functional Programming in Scala</b>	<b>148■</b>
10.1	Foundations of functional programming . . . . .	148■
10.2	Introduction to Scala . . . . .	149■
10.3	Data types in Scala . . . . .	153■
10.3.1	Primitive data types . . . . .	154■
10.3.2	Simple composite data types . . . . .	157■

10.4 Syntactic structures for defining functions . . . . .	159■
10.4.1 Pattern matching . . . . .	160■
10.4.2 Guards . . . . .	162■
10.5 Tail recursion . . . . .	162■
10.5.1 Computing the factorial function . . . . .	163■
10.5.2 Computing the Fibonacci numbers . . . . .	165■
10.5.3 Why tail recursion? . . . . .	166■
<b>11 Lists and Higher-Order Functions</b>	<b>168■</b>
11.1 Lists in Scala . . . . .	168■
11.2 Programming with lists . . . . .	170■
11.2.1 Polymorphism . . . . .	174■
11.2.2 Reversing a list . . . . .	178■
11.3 Higher-order functions . . . . .	180■
11.3.1 Higher-order functions for lists . . . . .	181■
11.4 Functional implementations of sorting algorithms . . . . .	186■
11.4.1 Insertion sort . . . . .	187■
11.4.2 Quicksort . . . . .	188■
<b>12 Functional Data Types</b>	<b>189■</b>
12.1 Type classes . . . . .	189■
12.1.1 The type class Ordering . . . . .	190■
12.1.2 The type class Numeric . . . . .	192■
12.2 Algebraic data types . . . . .	193■
12.2.1 Parametrized algebraic data types . . . . .	194■
12.2.2 Polymorphic algebraic data types . . . . .	195■
12.2.3 Recursive algebraic data types . . . . .	196■
12.2.4 Lists . . . . .	198■
12.3 Instances of type classes . . . . .	199■
12.4 Arithmetic expressions . . . . .	200■
<b>V Object-Oriented Programming and Data Abstraction</b>	<b>203■</b>
<b>13 Object-Oriented Programming</b>	<b>204■</b>
13.1 Imperative programming in Scala . . . . .	204■
13.2 Introduction to OOP . . . . .	211■
13.3 Encapsulation and information hiding . . . . .	213■
13.4 Classes . . . . .	216■
13.5 Sharing state between instances of a class . . . . .	223■
13.6 Inheritance . . . . .	225■
<b>14 Queues and Stacks</b>	<b>227■</b>
14.1 Traits . . . . .	227■

14.2	Abstract Data Types . . . . .	229■
14.3	Stacks . . . . .	230■
14.3.1	LinkedNodes implementation . . . . .	231■
14.3.2	Array implementation . . . . .	232■
14.3.3	Discussion . . . . .	234■
14.3.4	Arrays of dynamic size . . . . .	235■
14.4	Queues . . . . .	237■
14.4.1	LinkedNodes implementation . . . . .	238■
14.4.2	Array implementation . . . . .	239■
<b>15</b>	<b>Priority Queue</b>	<b>242■</b>
15.1	ADT PrioQueue . . . . .	242■
15.1.1	Sorting with priority queues . . . . .	244■
15.1.2	LinkedNodes Implementation . . . . .	244■
15.2	Array Implementation . . . . .	246■
15.2.1	Binary Trees . . . . .	246■
15.2.2	Binary Heaps . . . . .	247■
15.2.3	Header . . . . .	248■
15.2.4	Insertion . . . . .	248■
15.2.5	Deletion . . . . .	249■
15.3	Sorting with a PrioQueue . . . . .	250■
15.3.1	Insertionsort and Selectionsort . . . . .	250■
15.3.2	Heapsort . . . . .	251■
15.3.3	In-place Heapsort . . . . .	251■
<b>16</b>	<b>Dictionaries and Binary Search Trees</b>	<b>253■</b>
16.1	Abstract Data Type . . . . .	253■
16.1.1	Implementation with linked nodes . . . . .	255■
16.1.2	Implementation with arrays . . . . .	255■
16.1.3	Conclusion . . . . .	256■
16.2	Binary Search Trees . . . . .	256■
16.2.1	Definition . . . . .	256■
16.2.2	Searching . . . . .	257■
16.2.3	Insertion . . . . .	259■
16.2.4	Removing . . . . .	260■
16.2.5	Traversing a tree . . . . .	262■

# Entwurf

## I

### **Introduction and Motivation**



### What is Computer Science?

This class aims to provide a first overview of some major ideas in *Computer Science*. However, before we start, let us first try to explain what “Computer Science” actually is.

This question does not have a clean and short answer. Historically, Computer Science has evolved from several other—older—disciplines. As such, it encompasses a broad range of viewpoints, approaches, and subdisciplines. In addition, Computer Science is constantly evolving: new application areas appear and are included into the field, while old subfields become less popular and lose in prominence.

#### 1.1 Overview

The subdisciplines of Computer Science are quite diverse. Thus, there are many different views on how Computer Science fits into the broader scheme of scientific research. Here are some prominent opinions:

- **Computer Science is a science.** Computer Science studies the phenomenon of *computation and efficiency*, captured in the abstract notion of an *algorithm*. One can argue that computation is a physical process that occurs in many parts of the natural world. Efficiency is necessary for this computation to take place. Therefore, we can consider Computer Science a (*natural*) *science*. It tries to explore a phenomenon that is present everywhere in the natural world around us. The goal is to build a comprehensive theoretical and experimental framework that helps in understanding this phenomenon.
- **Computer Science is an engineering discipline.** Computer Science is concerned with constructing actual artifacts that help humans to fulfill their everyday tasks. Examples of such artifacts include smartphones, laptops, smartwatches, routers, as well as the countless applications and software systems that come with them. As such, Computer Science can be considered an *engineering discipline*. The purpose is to develop tools and methods for constructing these artifacts. Furthermore, we would like to collect experiences and best practices that make the development process easier.

- **Computer Science is a subfield of Mathematics.** The more theoretical aspects of Computer Science touch many fields of *Mathematics*, with a constant exchange of ideas in both directions. Questions and techniques from Computer Science permeate into the more traditional fields of Mathematics, while theoretical Computer Scientists continuously incorporate new tools from other mathematical areas into their field.
- **Computer Science is interdisciplinary.** Computer Science is in close contact with many other disciplines. For example, some aspects of artificial intelligence touch on foundational questions in philosophy, such as the mind-body-problem and the question of consciousness. Moreover, some artifacts of Computer Science, such as the World Wide Web or social networks, pose interesting research questions for the social sciences, the humanities, economics, or the law. Another prominent example is the field of *bioinformatics*. Here, the goal is to answer questions from the biosciences and medicine using tools from Mathematics and Computer Science.

These viewpoints are not mutually exclusive. Many people in Computer Science subscribe to several of these opinions, and most competent Computer Scientists agree that all of them have merit. In fact, some subfields of Computer Science straddle many of these aspects simultaneously. For example *cryptology* combines aspects of theory, engineering, usability, and policy; *computer graphics* can be done in both a very theoretical and a very applied fashion; and *bioinformatics* touches theory, engineering, and practical questions.

It seems impossible to list all topics that belong to Computer Science in this short introduction. However, to give an idea of some concrete areas of Computer Science, we provide a non-exhaustive list of example topics, some of which are also represented at our department.

*Algorithms.* This is one of the core disciplines of Computer Science. Algorithmic thinking belongs to the education of every competent Computer Scientist. The goal is to develop and analyze fast and correct algorithms for computational problems. In this class and throughout your studies, you will see countless algorithms for typical tasks such as searching, sorting, classification, optimization, etc.

*Design and construction of hardware systems.* This is another core discipline of Computer Science. The goal is to construct actual computers (such as servers, desktop computers, or smartphones), but also to design networking infrastructure, robots, autonomous systems, and much more. The area is also concerned with developing software that operates close to the hardware, such as operating systems or device drivers.

*Artificial intelligence and machine learning.* Nowadays, this is one of the most well-known areas of Computer Science. Even most non-Computer Scientists will have

heard about artificial intelligence, since it has generated a lot of publicity in recent years (as of 2024). This publicity is due to stunning developments in the field that have caught the imagination of many observers. In fact, the theoretical foundations for these developments have been known for decades. However, only now, with the availability of vast amounts of training data for the computational models of AI, the recent breakthroughs have become possible. We will say more about AI at the end of this chapter, in Section 1.5.

*Data compression.* This area deals with efficient ways to remove redundancies from data for efficient storage and processing. For example, one major goal is to stream high definition video content while using as little bandwidth as possible, or to make audio and video files small without losing any relevant information

*Data encryption, privacy, and security.* Data encryption is vital to the way we communicate online. For example, when we browse the web, the communication between our computer and the web server is encrypted (and the browser will show scary warning messages if encryption is not provided). Even if an eavesdropper had a transcript of the complete encrypted communication, they should not be able to deduce anything about the actual contents. More generally, computer security asks how an intruder could influence a given system to behave in an unintended way, and is looking for ways to prevent this. This can be done at many levels, e.g., by developing a comprehensive theory of secure systems, by developing tools that make it easy to create a secure system, or by studying the psychological factors that lead users to being negligent.

*Computational complexity theory.* This subfield of theoretical computer science tries to explore which resources (e.g., time or memory) are necessary to solve certain computational problems. One goal is to explain why some computational problems seem to be more difficult than others. Another goal is to show that known solutions cannot be improved. For example, we will see later in this class that for the sorting problem, the known algorithms are “best possible”, in a certain sense.

*Human-computer interaction.* Here, we study the different ways in which humans interact with computers. Different systems and applications require different kinds of user interfaces (e.g., visual, auditory, or haptic), and the user experience is crucial for the productivity and satisfaction of the people who use the artifacts of Computer Science.

*Data visualization.* This area deals with designing graphical or visual representations of (large amounts of) data that are easy to understand and analyze. In many application areas of Computer Science, such as the computational sciences, data science, or the digital humanities, these methods are instrumental in gaining new insights from the huge amounts of data that are available.

*Software development.* Here, we study the everyday task of designing and writing “good” software. This is a highly non-trivial problem, and the area encompasses a wide range of questions. For example, the goal is to find the right tools and processes for developing software, or to understand the interaction of developers in a team and effects of different ways to organize them.

*Limits of computability.* It turns out that not all natural algorithmic problems can actually be solved by a computer. This branch of theoretical computer science explores these fundamental limits of computation, and ways to overcome them.

*Impacts of computing technology on society.* Since computing technology has a large impact on our everyday lives, it is of great importance to understand what this means for our society and our political system. For example, this subfield studies the effects of surveillance technology, social networks, or autonomous vehicles, and what companies and governments can do to address them.

## 1.2 Terminology

In German, the field is called *Informatik*. This term was coined in the 1950s and became established in the 1960s. It is obtained by combining the word *Information* with the extension *-ik*. There are, however, different interpretations of what *-ik* stands for: it could come from *Mathematik*, from *Automatik*, or it could just be devoid of any meaning. An older, related discipline is called *Kybernetik*.

In English, the discipline is called *Computer Science*, emphasizing the role of computer systems as the main topic of study. However, some people suggest that using the term “computing science” might be more appropriate, since the field is more about the process of computation than about the devices that perform computation. As mentioned above, there is also some discussion about whether the term “science” is appropriate, since in English, this refers to a *natural* science.

## 1.3 Some historical notes

Historically, there are two main lines that were important to the development of today’s Computer Science: mathematical logic on the one hand, and engineering, industrial and military applications on the other hand.

The former line of development is concerned with finding a “general method” to solve (mathematical) problems. This question has a long history in the field of Mathematics. For example, in approximately 300 BC, Euclid published a highly influential textbook, the *Elements*. This textbook aims to be a comprehensive and systematic overview of the mathematical knowledge in Greece at Euclid’s time. In particular, it contains procedures for the construction of geometric objects such as equilateral triangles, right angles, or parallel lines from given inputs. Even before that, mathematicians studied the problem of “squaring the circle”: given a circle, is it

possible to construct a square with the same area, using only a compass and a ruler. Around 800 AD, Muḥammad ibn Mūsā al-Khwārizmī<sup>1</sup> described a systematic way to solve linear and quadratic equations. Adam Riese, around 1500 AD, popularized the use of the Arabic positional number system in Europe, as opposed to Roman system, that was dominant in Western culture until then. Adam Riese presented general methods for adding, subtracting, and multiplying those numbers, methods that are taught at elementary schools to this day.

All these developments show one thing: given the right understanding and a useful formalism, it is possible to *mechanically* solve problems that earlier required a great deal of creativity. At the turn of the 20th century, mathematicians wondered whether this holds for *all* mathematical problems. Concretely, in the 1920s, the mathematician David Hilbert formulated *Hilbert's program*. His vision was to provide secure foundations for all of mathematics on the basis of formal logic. All mathematical knowledge should be presented using a single formal language, the language of logic. Furthermore, this language should allow for *mechanical* procedures to verify and obtain *all* true mathematical statements.

Initially, David Hilbert and his collaborators made great progress towards this goal, providing a complete axiomatic characterization of geometry. However, then the progress stalled. In the 1930s, the Austrian mathematician Kurt Gödel showed that Hilbert's project is doomed to fail: logic and mechanical procedures will never be enough to capture all true mathematical statements. In the course of his program, Hilbert also posed the *Entscheidungsproblem*: "Does there exist a *general method* for recognizing all true (or at least all provable) mathematical statements?" In the 1930s, following Gödel's work, Alonzo Church and Alan Turing independently presented mathematical definitions that capture the notion of a "general method" to solve problems. Using their respective definitions, they showed that Hilbert's *Entscheidungsproblem* does not have a general solution. Even though the main results were negative, showing the impossibility of Hilbert's original plan, this line of research proved to be hugely influential and highly productive. In the following decades, it has led to the fields of computability theory, complexity theory, artificial intelligence, programming languages, algorithms, and many more.

In the other, more applied line of development, the goal is to construct mechanical and electronic devices that automatize large, repetitive (computational) tasks, so that these devices can also adapt easily to changing circumstances. One early example from the 18th century is the *Jacquard loom*.<sup>2</sup> This device could be "programmed" to produce different patterns, using *punch cards*. Other examples of such devices are the first mechanical calculators from the 17th century, developed by Blaise Pascal and Gottfried Wilhelm Leibniz. In the early 19th century, Charles Babbage tried to build the first general purpose computational devices. Even though this attempt failed due to the technical limitations of the time, the plans were quite concrete. In fact, Ada Lovelace already wrote first general programs for these designs. Before and during

---

<sup>1</sup>His last name is the origin of the term *algorithm*.

<sup>2</sup>A machine for weaving cloth.

the Second World War, the development of the first large mechanical and electronic computers began in earnest (e.g., Konrad Zuse's computers, ABC, Mark I, ENIAC, EDVAC, etc.). Some of these designs were classified (e.g., the Colossus computer that was used in England for code breaking). After the war, a fast and explosive development of computing devices sets in, in particular after the invention of the transistor and the microchip. Computers become ever faster and smaller, and new applications of computing devices were explored. In the 1980s and 1990s, networking, the internet, and the World Wide Web become widespread. Nowadays, this line of research lives on in the fields of computer engineering, computer architecture, robotics, telematics, and many more.

## 1.4 Computer Science today

As discussed above, the subareas of Computer Science are fluid and always evolving. Traditionally, Computer Science can be broadly categorized into three main areas:

*Theoretical Computer Science.* Deals with the mathematical foundations of computer science: algorithms, complexity theory, data structures, cryptography, semantics of programming languages, database theory, etc.

*Computer systems.* Deals with the hardware aspects of computer science: computer engineering, operating systems, computer networks, embedded systems, etc.

*Applied Computer Science.* Deals with concrete software and the process of developing it: Computer graphics, software engineering, artificial intelligence and machine learning, compiler construction, data visualization, etc.

In addition to these traditional areas, new subfields appear that emphasize the impact and connections of Computer Science to other scientific fields (e.g., computational sciences, data science, bioinformatics, digital humanities, etc.). Furthermore, the growing impact of Computer Science on society and our everyday lives also leads to many new concerns that call for new methods (e.g., legal aspects of Computer Science, Computer Science and society, network analysis, etc.).

## 1.5 Artificial Intelligence and Machine Learning

Nowadays, one subfield of Computer Science receives a particular amount of attention: in recent years, we have seen stunning developments in machine learning and artificial intelligence. This may lead to a new era in computing and to a dramatic transformation of our daily lives.<sup>3</sup> Here are some of the most important recent milestones that illustrate the new quality of AI technology :

---

<sup>3</sup>Or maybe not; at this point, it is hard to tell.

- In 2012, *DeepFace* allowed for highly accurate facial recognition on Facebook data. The system could automatically tag people in uploaded pictures, with nearly the same accuracy as a human. Nowadays, *FaceNet* is even better at this task than humans.
- In 2016, *AlphaGo* beat a human champion at Go and was quickly surpassed by *AlphaGo Zero*, a system which was trained by only playing games against itself, without any existing strategies or game plays it could learn from. At the same time, machine translation became much more reliable.
- In 2023, *ChatGPT* was able to have a natural conversation with a human being and to generate convincing texts in response to simple prompts.

Despite all these successes, it is quite difficult to give a definite definition of the field “artificial intelligence”. There are many differing opinions, and the precise notions vary. In general, the goal of the field is to perform tasks with a computer that are easy for human beings, such as playing a game, driving a car, having a conversation, or reading a newspaper. The specific objectives reach from very narrow problems like recognizing handwritten digits to very wide-reaching tasks like writing a computer program that achieves “consciousness”, like a human being. These later objectives are often discussed under varying terms, such as “strong artificial intelligence”, “artificial general intelligence”, etc.

More concretely, it may be more enlightening to look at the specific problems that are studied under the umbrella of artificial intelligence, and at the methods that are used to address them. Broadly, the problems of artificial intelligence fall into three categories:

- **Classification:** Given a piece of data and a set of *categories*, decide which category this data belongs to. Typical examples are recognizing handwritten characters, detecting fraudulent financial transactions, classifying pictures on the internet, helping in medical diagnoses; and many more.
- **Generation:** Given a short piece of information, or “prompt”, generate an artifact that looks like it was made by a human, such as a piece of text, a picture, a video, or music. This is the subarea that has received a lot of press recently, with examples like ChatGPT (text), DALL-E (pictures), etc.
- **Planning:** Given a set of constraints and rules, try to find a sequence of actions to accomplish a certain task. Typical examples are playing a game, parking an autonomous vehicle, or solving a puzzle.

To address these problems, the field of artificial intelligence has developed a large variety of methods. Broadly, these fall into two categories:

- **Symbolic methods:** Here, the idea is to elicit all the knowledge about a certain problem that exists in the world and among experts, and to put this knowledge

into a form that can be processed by a computer. The computer program should try to simulate the reasoning process of a human when solving its task.

A classic example of symbolic methods can be found in the victory of the chess computer IBM DeepBlue over the chess grandmaster Gary Kasparov in the 1990s. There, IBM spent a lot of effort to elicit knowledge from the chess literature and from living grandmasters and to process into huge lookup tables that could help DeepBlue to evaluate positions and to find the next move. Special hardware was constructed to speed up the game playing algorithms, and the chess grandmasters were consulted to fine-tune the evaluation functions. Eventually, DeepBlue was able to narrowly defeat Kasparov, a milestone in modern artificial intelligence.

- **Machine learning:** The methods of machine learning are data-driven. The idea is to “train” a general model (such as a neural net) with large amounts of data, the more the better. The model is supposed to extract the relevant information directly from the data, without much assistance from humans. Unlike symbolic methods, data-driven methods lead to models that are difficult or impossible to understand for human beings. It can be very hard to see how a model comes to a result, even though we may be able to judge the quality of the result quite easily. These methods also have been around for a long time, but only in the last decade, with the availability of vast amounts of digital data, they really started taking off. The recent successes of artificial intelligence are mostly based on machine learning methods.

Throughout this class, we will discuss the problems and methods that are studied in artificial intelligence. We will provide simple examples and exercises that illustrate their use. This is part of a larger effort to integrate more topics from artificial intelligence in our basic curriculum.



# Entwurf

## II

### **Imperative Programming**

### Basics of Imperative Programming

#### Lernziele

**KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

**KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigm.



#### 2.1 Von-Neumann-Architecture

Before we talk about the fundamentals of programming languages, we first take a brief look at the underlying hardware. This helps us understand the choices and principles that underlie the design of modern programming languages.

In modern usage, a *computer* is a physical device that executes one or many *programs*. A program is a sequence of *instructions* that are usually executed sequentially, one after the other. A computer is a *universal* machine. Unlike, say, a washing machine or a toaster, a computer is not built to fulfill a certain task. Instead, the specific task that a computer fulfills is determined by the program that it currently executes. This task may change over time. For example, first, a computer can solve a mathematical equation, then it can play a movie, then it can find a fast public-transport connection to the university, and then it can be used to write a homework solution.

In the early days of computing, there were many suggestions of how to construct a computer. One of the most influential solutions is named after the mathematician John von Neumann. It is called the *von-Neumann architecture*<sup>1</sup>. In the von-Neumann architecture, the pieces of a computer are structured in the following four components:

1. **Main Memory/Random Access Memory (RAM).** The main memory stores both the programs and the data that the programs operate on. It consists of *memory cells*. Each memory cell stores a binary *data word* with a fixed number of bits,

<sup>1</sup>This architecture was first described in 1945.

which is the same for all data words. The memory cells are numbered consecutively, starting from 0. The number of a memory cell is called its *address*. The main memory can read or write any given memory cell using its address, in arbitrary order. (The word “random” in “random access memory” is used in the sense of “arbitrary”).

2. **Central Processing Unit (CPU).** The central processing unit controls the working of the computer and executes the program. It consists of several parts: the *Control Unit* coordinates the other components of the computer and is in charge of running the program; the *Arithmetic Logical Unit* (ALU) contains the circuitry to perform arithmetic (e.g., addition, subtraction, multiplication, division) and logical operations (e.g., bitwise AND, bitwise OR, shift left, shift right, etc); the *Program Counter* is a memory cell that stores the memory address of the next instruction in the current program; and the *clock generator* generates an electronic signal that synchronizes all the components of the computer.
3. **Input/Output (I/O).** Consists of physical devices that connect the computer to the outside world, e.g.; keyboard, mouse, USB port, screen, network connector, sound card, and many more.
4. **Bus system.** The bus system consists of a set of electronic connections that transports data, addresses, and the clock signal between the other three components.

Now, given this architecture, we can describe the basic operation of a computer. As soon as a computer is turned on, it starts to execute instructions from main memory. This is done by repeating the following *machine command cycle* indefinitely, until the computer is turned off:

- (i) **FETCH** (fetch the next instruction from main memory): the control unit uses the bus to send the contents of the program counter to the main memory. The main memory reads the memory cell with this address, and it uses the bus to send the contents back to the control unit. This represents the next instruction that needs to be executed.
- (ii) **DECODE** (interpret the next instruction): the control unit looks up the code for the instruction in a table, and it fetches additional operands from main memory, if necessary.
- (iii) **EXECUTE**: The control unit coordinates the other components in order to perform the desired instruction, e.g., perform a calculation, a logical operation, or a comparison in the ALU; transfer data in the main memory; perform an input/output operation; or more. After this, the control unit updates the program counter to the next instruction. The next instruction is typically the instruction after the current one (sequential execution); but for certain instructions, it may be specified by an operand (jump) or by the result of comparison (conditional branch).

(iv) **REPEAT**: Begin the next cycle, for the next instruction.

The programs that are executed are given in *machine language*. This consists of very simple instructions, such as arithmetic and logic operations, comparisons, branches, data transfer, or input and output operations. The possible instructions and their precise functionality depend on the concrete hardware and can vary widely between systems and architectures. They are encoded numerically, as a sequence of numbers. For example, 10 12 05 13 F2 0D might be a valid machine language instruction. The first computers were programmed directly using this simple machine language. It was very cumbersome and error-prone to learn, use and maintain such programs.

## 2.2 Programming languages

Programming languages are artificial languages that are designed to simplify the programming task for humans. We distinguish these languages by their degree of abstraction.

### 2.2.1 Assembly languages

Assembly languages provide a simple textual representation of the numerical machine language of a given computer. In particular, assembly languages introduce *mnemonics*: short names for machine language instructions, such as ADD, SUB, MUL, CMP, BNZ, IN, OUT, and many more. Furthermore, they provide a simple syntax for operators, memory references, and control structures. Here is a short example an assembly language program for an x86-processor:

```
MOV  EBX, EAX
SUB  EAX, 25
JMP  foo
```



The first instruction copies the contents of the memory cell EAX into the memory cell EAX. The second instruction subtracts the number 25 from the contents of the memory cell EAX, and it writes the result back into EAX. The third instruction changes the program counter, so that it points to the memory location that is given by the label foo (this is called a *jump*).

To program in assembly language, we write the source code into a simple text file. Then, we run a special program, the *assembler*, that directly translates this text file into the numerical machine language. Some assemblers provide additional functionality to simplify the coding task (e.g., a macro facility to reuse snippets of code; a way to name memory locations with the help of labels; resolution of addresses, etc). Using an assembler simplifies the coding task, since assemblers provide a more human readable form of the machine language and automate the boring task of translating a program into its numerical representation. However, assembly language is still very specific

to the underlying hardware, and it forces us to program in very small, unstructured steps.

### 2.2.2 High-level languages

High-level languages put the focus on the human who performs the programming task, and not on machine that executes the instructions. The goal is to have a formal artificial language that allows humans to solve algorithmic problems easily and efficiently. The language should be close to how humans actually speak. At the same time, it should have enough formal structure so that it can be processed by a computer.

High-level languages have a *higher level of abstraction* than assembly languages. They provide more structure and are easier to understand. Furthermore, they provide language features that help humans to avoid and detect programming errors early in the development process. Finally, high-level languages depend less on the concrete hardware: programs that are written in a high-level language can be used on computers with differing hardware and architectures.

There are many different high-level languages. Throughout your studies, you will encounter several of them. In this class, we will focus on *Python 3* and on *Scala 3*, because we believe that these two languages are useful in showcasing the different approaches and features that are common in most modern programming languages.



## 2.3 Expressions

Almost every high-level programming language has a notion of an *expression*. An expression is a syntactic construct that has two crucial properties: a *type* and a *value*. The *type* of an expression determines the range of possible values and the set of the possible operations. Every programming language has its own specific collection of types. Commonly, there are types that represent integral numbers, fractional numbers, truth values, characters, text, and more complicated data items. We will look at types in more detail later in this class. The *value* of an expression is the specific result that is obtained by performing the computations that the expression describes. A basic task in every programming language is to determine the value of a given expression, i.e., to *evaluate* the expression. Here are some different kinds of expressions, with corresponding examples in Python.

- *Constant expressions*. A constant expression (also called a *literal*) directly represents a single value. For example:

```
# A constant expression with type integer and value 0
0
# A constant expression with type fractional number and value 1.5
1.5
```



```
# A constant expression with type text (string) and value "Informatik"
"Informatik"
# A constant expression with type truth value and value True
True
```

- *Arithmetic expressions.* Arithmetic expressions consist of numbers and basic numerical operations that are familiar from elementary school. Typically, arithmetic operations also allow for parenthesis and follow the known rules of precedence. For example:

```
# An arithmetic expression with type integer and value -3.
1 + 2 * (5 - 7)
# An arithmetic expression with type fractional number and value 0.8333333.
1.5 - 2/3
```

- *Boolean expressions.* Boolean expressions consist of comparison operators, truth values, and logical operators. Their type is always truth value.

```
# A Boolean expression that is a predicate with value True.
3 <= 5
# A Boolean expression that is a predicate with value False.
7 == 2
# A Boolean expression consisting of logical operators, with value True.
True and not False
# A Boolean expression consisting of logical operators, with value True.
True or False
# A Boolean expression with comparisons and logical operators; it is False.
3 <= 5 and 7 == 2
```


- *Conditional expressions.* Conditional expressions consist of three parts: a Boolean expression (the *condition*) and two arbitrary expressions (the *yes-expression* and the *no-expression*). To evaluate a conditional expression, we first evaluate the condition. If the result is True, we evaluate the yes-expression; if the result is False, we evaluate the no-expression.

In Python, the concrete syntax is “*yes-expression if condition else no-expression*”. For example:

```
# A conditional expression with type integer and value 10.
0 if 3 >= 5 else 10
# A conditional expression with type integer and value 20.
20 if 12*5 == 60 else 10
```

In Python, the types of the yes-expression and the no-expression may be different. For example:


```
# A conditional expression with type integer and value 10.  
True if 3 >= 5 else 10  
# A conditional expression with type truth value and value False.  
True if 12*5 == 60 else 10
```



In particular, the type of a conditional expression in Python can be determined only after the condition has been evaluated. This is a result of the fact that Python is *dynamically typed*. A programming language may also be *statically typed*. This means in particular that the type of every expression must be known in advance, before any evaluation takes place. In a statically typed language (e.g., in Scala), we would insist that the yes-expression and the no-expression have the same type. We will talk more about the difference between statically and dynamically typed languages later in this class.

- *Function calls*. Python has built-in functionality that is represented as *functions*. A function can receive zero, one, or more *arguments*, and it returns a *result* that depends on the parameters. Functions can be used in expressions. The type of such an expression depends on the function that is used. The available pre-defined functions are described in the Python documentation. It is also possible to define our own functions, and this is actually a main part of programming in Python. We will talk much more about this later in this class.

```
# A function that determines the number of characters in a text.  
# It takes an expression of type text as an argument, and returns  
# an integer. The value is 5.  
len("Hallo")  
# A function that determines the absolute value of an integer.  
# It takes an expression of type integer as an argument, and it returns  
# an integer. The value is 1.  
abs(-1)  
# A function that rounds a fractional number down to the next integer.  
# It takes an expression of type fractional number as an argument, and it  
# return an integer. The value is 2.  
int(2.7)
```



Different kinds of expressions can be combined and nested in arbitrary ways, as long as they fit together, as determined by their types.

## 2.4 Imperative Programming

There are several different approaches (or philosophies) when designing a high-level programming language. These approaches are also called *programming paradigms*. The main paradigms are as follows:

- **Imperative programming languages**. Imperative programming languages have the same structure as low-level languages, but at a higher level of abstraction:

an imperative program is a sequence of *instructions* that change the *state* of the computer. Most popular languages are imperative, e.g., C, C++, Java, C#, Pascal, Python, Rust, and many others.

- **Declarative programming languages.** Declarative programming languages try to hide the underlying hardware completely. Instead, they use another (mathematical) abstraction as the guiding idea for writing a program. Depending on this abstraction, there are different types of declarative programming languages, e.g., *logic programming languages* (based on mathematical logic; for example, PROLOG); *functional programming languages* (based on mathematical functions; for example, Haskell, ML, or Scala); or *query languages* (based on mathematical relations; for example SQL). Typically, a declarative program describes *what* should be computed, but the details of *how* to do this are left to the machine.
- **Multi-paradigm programming languages.** Older languages such as Haskell, C, Pascal, or PROLOG often follow only a single programming paradigm. However, many recent languages combine ideas of different paradigms. For example, the Python and Scala are functional, imperative, and object-oriented programming languages.

Now, we will focus on *imperative programming* (the functional paradigm will be covered later in this class). To illustrate the main ideas of imperative programming, we will use *Python*. Python is a relatively recent language that has become popular in the last few years. It is often called a *scripting language*: it focuses on quick solutions and it tries to avoid many constructs from other languages that are aimed at enforcing structure and discipline in larger software projects. There is a vast collection of libraries and tools for Python programming that makes the language very useful in the context of scientific and machine learning applications.

The central notion of an imperative program is the *state* of the machine. The state machine consists of the contents of all the memory cells; the program counter; and the configurations of all the input/output devices. An imperative program consists of a sequence of *instructions* that are executed in sequence, one after the other. Each instruction changes the state of the machine. (the *effect* of the instruction). The goal is to write a sequence of instructions that transform a given *input state* into a desired *output state*.

The instructions of a given imperative programming language are characterized by their precise effect state. This details differ from programming language to programming language, but typically, instructions fall into three categories:

- **Assignments.** Assignment instructions affect the state of the main memory, by creating, changing, or erasing data items.
- **Control flow.** Control flow affect the state of the program counter, by controlling the order in which the instructions of the program are executed.
- **Input and output.** Input and output instructions affect the state of the input/output devices, by reading from or writing to devices in the outside world.



We will now look at the different kinds of instructions in more detail, and we will see how they look like in Python.

### 2.4.1 Assignments

Assignment instructions in Python (and also in other languages) are of the form:

```
name = expression
```



Some languages use `:=` instead of `=`, and some languages need a semicolon at the end of the line. To execute an assignment instruction, we first evaluate `expression`. Then, we *assign* the resulting value to a *variable* that is identified by `name`. It is not clear what exactly this means, and in fact, the precise interpretation of this process varies from programming language to programming language.

In Python, the meaning is as follows: after evaluating `expression`, Python creates a new *data object* and stores it in the memory.<sup>2</sup> This data object contains the type and the value that result from `expression`. After this, the variable `name` is *associated* with the data object. This means that after the assignment instruction, we can use `name` in expressions to obtain the value of the data object that `name` is associated with. Thus, in Python, a *variable* can be thought of as a label that can be used to identify different data objects.

It is important to note that the type is a property of the data object and not of the variable name. This means that throughout a Python program, the same variable name can be associated with different data objects of completely different types. It is up to the programmer to keep track of what kind of data objects `name` is currently associated with. We say that Python is *dynamically typed*.

In contrast, other programming languages (e.g., Scala, Java, C, Haskell) are *statically typed*. Here, we must declare a type for any variable name that we would like to use in our program. Thus, the type is a property of `name`, and it cannot change throughout the execution of the program. In this setting, a variable typically represents a fixed memory location that can store only data objects of this type.

A noteworthy case is an assignment of the following form:

```
name1 = name2
```



Here, we assign the variable `name2` to the variable `name1`. There are two possible ways how such an instruction can be interpreted:

1. **the assignment instruction has reference semantics:** `name1` becomes *synonymous* to `name2`. That is, after the assignment instruction is executed, the identifiers `name1` and `name2` refer to *the same* data object in memory. This is the

---

<sup>2</sup>This is only partially correct. For performance reasons, some implementations of Python reuse existing data objects in order to save time and memory. For example, for truth values, there are only two fixed data objects that represent **True** and **False**. For integers, there are fixed objects for the numbers in the range from `-5` to `256`.

interpretation that is used in Python. We call this interpretation *reference semantics* of the assignment statement; or

2. **the assignment instruction has value semantics:** we make a copy of the data object that `name2` refers to, and we associate this copy with `name1`. After the assignment instruction, `name1` and `name2` refer to *two different* data objects that have *the same* type and value. This is the interpretation that is used in programming languages like C, C++, or Pascal. We call this interpretation *value semantics* of the assignment statement.

Most data objects in Python are *immutable*: their value cannot be changed. For such data objects, the reference semantics of the assignment statement in Python does not make a difference. However, some data objects in Python are *mutable*, and their value can be changed by later instructions. In this case, the change is visible under all identifiers that are associated with the same data object. We will see several examples later on.

### 2.4.2 Input and output

Input/Output instructions allow the program to communicate with the outside world (e.g. the user, the file system, the network, a screen, a speaker). The precise details of input/output instructions vary widely from programming language to programming language. They also depend on the precise nature of the outside world. In some languages (e.g., in Python), instructions for communicating with the user look quite different from instructions for communicating with the file system. Other programming languages (e.g., Java or C++) try to handle all kinds of input/output in a uniform way, irrespective of whether it may be appropriate or not.

For now, we consider two input/output instructions in Python that we will use frequently: the instructions for outputting a text onto the screen and for reading an input from the keyboard.

- `print(arg1, arg2, ..., argk)` receives an arbitrary number of arguments. Each argument is an expression. The arguments are evaluated, and each result is converted to a string. The strings are concatenated, separated by spaces. The resulting string is then displayed on the screen. In the process, `print` also interprets certain *control sequences* that may appear in the argument strings, e.g., `\n` creates a new line, or `\t` performs a tab-operation.

```
# shows "Konzepte der 42 ." on the screen
print("Konzepte", "der", 6*7, ".")
```



Alternatively, we can call `print` with a single string argument that we prepare ourselves. This gives more control over the layout of the string. However, we must convert non-string data objects to a string manually, typically by using the function `str(·)`.

```
# shows "Konzepte der 42." on the screen
print("Konzepte " + "der " + str(6*7) + ".")
```



- `input(string)` outputs string onto the screen and reads an input from the keyboard. The input is terminated by the return key. The input is stored as a string object and can be assigned to a variable.

If we want to receive as input a data object of another type (e.g., a number), we must convert the input string manually to this type (e.g., by using the functions `int(·)` or `float(·)`).

**Attention:** When dealing with input/output operations, we must always expect errors. For example, the user input might contain mistakes (we would like to receive a number, but we do not get one), hardware might fail, the network may be off-line, someone may have deleted the file that we would like to access, etc. We must be ready to handle such errors in our code. We will talk about how this is done later.



### 2.4.3 Control flow

Using expressions, assignments, and input/output instructions, we can already write interesting imperative programs, e.g., for performing simple computational tasks or for implementing simple numerical algorithms. However, to obtain more powerful programs, we need instructions that allow us to adapt our program to the input data. These instructions affect the *control flow* of the imperative program. That is, they change the order in which the instructions of the program are executed. Control flow instructions modify the program counter in the CPU. There are several types of control flow instructions.

**Jumps.** Jumps are the most direct way to affect the control flow in an imperative program.

To use a jump instruction, we write a *label* in front of the instruction that we would like to jump to. At another point in the program, we write `goto` or `jump`, followed by the name of the label. Then, when the `goto`-instruction is executed, the imperative program continues with the instruction after the given label.

Even though jumps seem to be simple, they can lead to unstructured code that is hard to understand, hard to maintain, and hard to debug. Thus, most modern imperative languages follow a *structured programming* approach and do not allow full-fledged `goto`-instructions. Jumps are allowed only in very limited ways, e.g., in the context of exceptions (see below) or in the form of `break` or `continue` instructions in the context of loops (see below).

**Attention:** Although jumps are technically possible in many languages, it is bad programming style to use them. Hence, do not use any jumps, unless you *really* know what you are doing.



**Conditional execution.** In conditional execution, we can indicate that a certain sequence of instructions (called a *block*) should be executed only if a certain condition is met. We can test for several conditions, and we can also provide a block for the case that no condition is met.

In Python, conditional execution is handled through the **if**-construct. It has the following form:

```
if condition1:
    Block1
elif condition2:
    Block2
elif condition3:
    Block3
...
else:
    Blockn
```



First, `condition1` is tested. It is a Boolean expression that is evaluated. If the value is **True**, Python executes `Block1`, and afterwards continues with the instruction after the **if-elif-else**-construct. If the value is `False`, Python evaluates `condition2`, which is again a Boolean expression. If the value is **True**, Python executes `Block2` and afterwards continues with the instruction after the **if**-construct. This continues until the first condition that evaluates to **True**. If no condition evaluates to **True**, then `Blockn`, i.e., the block after the final **else**-, is executed.

There can be an arbitrary number of **elif**s (also none). The **else**:-part can be omitted, in which case the execution continues after the **if**-construct, if no condition is **True**.

A block can consist of one or more instructions. In Python, a block is marked by *indentation*, i.e., the whitespaces that precede the instructions.

**Loops.** Loops allow us to execute a sequence of instructions (a *block*) several times. There are two kinds of loops:

- Counting loops/**for**-loops. **for**-loops allow us to iterate over all elements of a given domain or range. In a **for**-loop, there is typically an *index variable* that successively goes through all the possible values of the range, in order.

In Python, a **for**-loop looks as follows:

```
for variable in range:
    Block
```



Here, `range` can be anything that contains a set of elements that we can iterate over. We will see some examples below. While executing the loop, `variable` is associated successively with every element of the range, and `Block` is executed for each such assignment.

One common scenario is that `range` consists of *integers*. In Python, such a range is written as `range(a,b,s)`. This represents the (ordered) sequence of numbers that is obtained by starting with `a` and repeatedly adding `s`. The sequence stops when we meet or pass `b` (this last number is *not* included). The *step size* `s` may be negative, which means that we count downwards. If `s` is positive and  $b \leq a$ , or if `s` is negative and  $b \geq a$ , then the range is *empty*. If we omit `s`, then it defaults to 1. In this case, `range(a,b)` consists of all numbers from `a` (included) to `b` (excluded) (empty, if  $b \leq a$ ). We can also use `range(a)`, which represents all numbers from 0 to  $a - 1$ .

```
# 0 2 4 6 and 8 are shown on the screen
for x in range(0,10,2):
    print(x)
```



Another example of a range is a string. In this case, the variable is associated to all the characters that appear in the string, in order.

```
# I n f o r m a t i k are shown on the screen
for x in "Informatik":
    print(x)
```



- Conditional loops/**while**-loops. Conditional loops consist of a *condition* and a *block*. The number of times the block is executed is controlled by the condition. There are several variants of conditional loops, depending on whether the condition is necessary for continuation (**while**-loop) or for termination (until-loop) and whether we first test the condition and then execute the block or whether we first execute the block and then test the condition.

In Python, there is only one conditional loop. In this variant, the condition is necessary for continuation, and the condition is tested before the block is executed. The syntax is as follows:

```
while condition:
    Block
```



`condition` is a Boolean expression. It is evaluated first. If the value is **False**, the execution continues after the **while**-construct. If the value is **True**, `Block` is executed, and afterwards, `condition` is tested again. This continues until `condition` is evaluated to **False**.

```
# 0 2 4 6 and 8 are shown on the screen
x = 0
while x < 10:
    print(x)
    x = x + 2
# 0 2 4 6 8 and 10 are shown on the screen
```



```
x = 0
while x <= 10:
    print(x)
    x = x + 2
```



**Dealing with errors.** Whenever we write a serious imperative program, we need to be prepared to deal with errors. Even if our own code is perfect (which is unlikely), we will need to rely on input/output operations, and these input/output operations can lead to problems. For example, a user input might not have the form that we expect, we might try to access a file that does no longer exist, we might try to use a storage device that is not physically present, we might try to use a network connection that is currently down, etc.

Thus, our code needs to have a capability to handle errors. There are several ways how this can be done in imperative programs.

- **Abort the program.** Whenever an error occurs, the program is terminated. We may output an error message to the user, and we return to the operating system. All data is lost, the program needs to be restarted, in the hope that the error does not happen again.

This is the default behavior in Python, and in many other programming languages. Even though it is quite easy for the programmer, it is not very useful for people who use the program.

- **Introduce a special value that indicates that an operation could not be executed as expected.** For example, some number types may have a special value, NaN (not a number), that can indicate that a division through 0 has occurred. If a language decides to use this feature, one would need to check after a division operation whether the resulting value is NaN. If so, the program needs to react appropriately.

Similarly, one might implement the `int(input(...))` operation in such a way that it returns **None** if the string input cannot be interpreted as an integer number. Again, the program would need to test explicitly for this **None** value to handle the error.

This behavior is not implemented in Python, but we can find it in other languages, most prominently in C. In C, most operations can result in a special value `-1` which indicates that an error has occurred. There is a variable `errno` that can be tested by the program to obtain more information on the precise nature of the error.

- **Use a special error handling mechanism (SIGNAL, interrupt).** Somewhere in our program, we define a special *error handler*. This error handler is called whenever an error occurs. The task of the error handler is to determine the cause of the error and to try to fix it. After that, the program is resumed at the

instruction where the error has occurred. The program tries to repeat the critical operation. If the error happens again, the error handler is called again, etc.

This mechanism is implemented in many operating systems (e.g., in order to deal with a missing storage device<sup>3</sup>), but it is not the default behavior in Python.

- **Use Exceptions.** Exceptions offer a structured way to deal with errors. Exceptions are a language feature that tries to separate the actual code and the error handling mechanism. They are similar to a global error handler, but work at a more fine-grained level. Exceptions are present in many modern programming languages. In particular, they are the default error-handling mechanism in Python.

In Python, exceptions use the following general syntax:

```
try:
    block
except:
    error handler
```



We put the critical operations into a **try**-block. If the operations are executed without problems, the execution continues after the exception-construct (the error handler is not executed). If an operation in the **try**-block causes an error, the **try**-block is aborted immediately. In this case, the execution continues with the error handler. After the error handler is finished, the program continues after the exception construct, and it is up to the program to deal with the error (e.g., the error handler might provide a default value and the program might continue, or the exception-construct might be in a loop that is repeated until the operation succeeds).

In fact, the exception-handling mechanism in Python is much more sophisticated, than what we describe here: We can test for specific errors, we may define our own errors, we may nest **try**-blocks, and there are additional blocks that may be added to the exception construct (e.g., a **finally**-block that is always executed after the other blocks). We refer the interested reader to the Python documentation for more details.

This concludes our tour of the basic instruction types in imperative programming languages. Now we have all the tools to implement more serious imperative programs. We conclude the chapter with two small examples.

### 2.4.4 Two programs

**First example: guessing a number.** Our first example is a small number guessing game. The program chooses a secret number between 1 and 100. The user is supposed

---

<sup>3</sup>Such an error handler is the source of the legendary MS-DOS error message `Drive is not ready: Abort, Retry, Ignore`, a classic example of a bad user-interface that has puzzled users for decades.



to guess this number. The program provides hints to help the user with this task. The Python-code is as follows:

```
from random import randint
number = randint(1, 100)
print("Hello, dear player. Welcome to our little game.")
print("I will think of a secret number between 1 and 100.")
print("It is your task to guess this number.")
guess = -1
while guess != number:
    guess = -1
    while guess == -1:
        try:
            x = input("Please enter a number between 1 and 100: ")
            guess = int(x)
            if guess < 1 or guess > 100:
                raise exception
        except:
            guess = -1
            print("This was not a valid number! Please try again.")
    if number > guess:
        print("My number is bigger.")
    elif number < guess:
        print("My number is smaller")
print("Congratulations! You have won!")
```

In order to choose a secret number, the program uses a random-number generator. For this, we *import* the function `randint` from the module `random`. As mentioned, Python has a large collection of functions that provide a lot of functionality. These functions are organized into *modules* that can be *imported* into a program to make the functions available. We will learn more about this in later classes.

In this case, the function `randint(a, b)` allows us to generate a random integer number between *a* and *b* (including *a* and *b*).<sup>4</sup> We use it to pick a secret number between 1 and 100. This number is stored in the variable `number`.

Next, the program shows a few instructions on the screen, using the `print`-function. After that, the main game begins. The game takes place in a `while`-loop that iterates until the player has guessed the secret number. In each iteration of the main `while`-loop, the program obtains a guess from the player. This guess is stored in the variable `guess`. The `while`-loop terminates as soon as the guess is correct. We want the `while`-loop to be executed at least once. To achieve this, we set the variable `guess` to `-1` before the `while`-loop is executed for the first time. This ensures that the loop-condition

---

<sup>4</sup>Actually, this number will not be truly random, but only *pseudo-random*. For our purposes, this does not make any difference.



`guess != number` is fulfilled initially (since we know that `number` must lie between 1 and 100), and the loop-body will certainly be executed at least once.<sup>5</sup>

In each iteration of the main **while**-loop, the program asks the user to guess a new number. For this, we use an inner **while**-loop. Before the inner **while**-loop is executed, we set `guess` to `-1` to indicate that we do not yet have a valid guess. The inner **while**-loop executes until `guess` is no longer `-1`. Within the **while**-loop, we use the `input()`-function to receive a string `x` that is supposed to represent a number between 1 and 100, and the `int()`-function to convert the string `x` to the number `guess`. Since it may happen that `x` does not represent a valid number between 1 and 100, we use a **try-except**-block to handle the potential error. If the call `int(x)` is not successful, the **except**-block is executed, and the user sees an error-message. Within the **except**-block, we also set `guess` to `-1` to indicate that the guess has not been valid. If the call `int(x)` is not successful, it may still be the case that the input does not lie between 1 and 100. We check this directly with an **if**-instruction. If `guess` does not lie in the desired range, we manually create an error, using the instruction `raise` exception. This will cause the execution of the **except**-block, with an error message and with setting `guess` to `-1`.

If `guess` is valid, the inner **while**-loop terminates. After that, we use an **if-elif**-instruction to provide a hint about the secret number, indicating whether the secret number is larger or smaller than the guess.

Finally, when the guess is correct, the main **while**-loop terminates at last and the program outputs a success message.

**Second example: computing the square-root.** Our second program deals with a well-known mathematical problem: given a number  $a > 0$ , we would like to compute the *square-root*  $\sqrt{a}$  of  $a$ . For example, given the number  $a = 4$ , we would like to compute  $\sqrt{4} = 2$ .

Of course, in general, this will be impossible. From *Diskrete Strukturen für Informatik*, we know that there are numbers whose square roots are *irrational*. For example, for  $a = 2$ , we know that  $\sqrt{2} = 1.4142\dots$  cannot be represented as a fraction. This means in particular that we need an *infinite number of digits* to represent such a square root. Thus, the best we can do is to compute an *approximation* of  $\sqrt{a}$ : a number that we can represent with the finite precision of a computer, but that is “close enough” to the actual square root.

To make this precise, we need a mathematical definition of “close enough”. There are several ways to do this, but we use the following simple notion: we pick a number  $\varepsilon > 0$ , our *desired precision*. We say that a given number  $a$  is “close enough” to  $\sqrt{a}$  if we have

$$|r^2 - a| \leq \varepsilon. \quad (2.1)$$

---

<sup>5</sup>In other programming languages, we could use a conditional loop where the condition is tested at the end of an iteration, such as a `do-while`-loop in Java or a `repeat-until`-loop in Pascal. However, Python does not offer this kind of conditional loop.

This means, we would like the *square* of  $r$  to be within an  $\varepsilon$ -distance to  $a$ . This condition is easy to check for a given  $r$ , and it seems like a reasonable definition for being “close enough” to  $\sqrt{a}$ .

Next, we need a way to obtain a good approximation for  $\sqrt{a}$ . The mathematical literature contains a simple iterative method for this. This method proceeds in steps and provides increasingly better approximations of  $\sqrt{a}$ . It is as follows: we start with a simple fixed approximation  $x_0 > 0$  for  $\sqrt{a}$ . For concreteness, we pick  $x_0 = 1$ . Now, we observe the following: if  $x_0$  is smaller than  $\sqrt{a}$ , then  $a/x_0$  will be larger than  $\sqrt{a}$ . If  $x_0$  is larger than  $\sqrt{a}$ , then  $a/x_0$  will be smaller than  $\sqrt{a}$ . In any case, no matter how good an approximation  $x_0$  is, we can always conclude that  $\sqrt{a}$  needs to lie between  $x_0$  and  $a/x_0$ . Thus, in particular, the mean value  $x_1 = (x_0 + a/x_0)/2$  of  $x_0$  and  $a/x_0$  will be a better approximation for  $\sqrt{a}$  than  $x_0$ . The same argument applies to  $x_1$ , and we can do the same step to get an even better approximation  $x_2$ , and so on.

Thus, the following sequence provides increasingly better approximations for  $\sqrt{a}$ :

$$x_0 = 1; \text{ and } x_n = \frac{1}{2} \left( x_{n-1} + \frac{a}{x_{n-1}} \right), \text{ for } n \geq 1. \quad (2.2)$$

This sequence of approximations can be implemented in a **while**-loop that iterates until the current approximation is “close enough”. The following Python-program implements this idea.

```
a = -1
while a == -1:
    try:
        x = input("Please enter a positive number: ")
        a = float(x)
        if a <= 0:
            raise exception
    except:
        a = -1
        print("This was not a valid number! Please try again.")
eps = 0.0000001
r = 1
while abs(r*r - a) > eps:
    r = (r + a/r)/2
print("The square root of " + str(a) + " is " + str(r) + ".")
```

First, the program prompts the user to obtain a positive number  $a$ . We use the same mechanism as in the number-guessing game. We set  $a$  to  $-1$  to indicate that no valid number has been obtained yet. Then, we use a **while**-loop to obtain a valid input, using the functions `input()` and `float()` to obtain the number. Again, exceptions are used to handle possible input errors, including the case that the input could be a negative number.

Once the **while**-loop terminates, we begin with the mathematical part of the program. The program contains a constant `eps` that represents our desired precision. The variable `r` stores our current approximation for  $\sqrt{a}$ . Now, the main computation

takes place in a **while**-loop that directly implements the mathematical procedure. In particular the loop-condition is a direct Python-implementation of (2.1), and the loop-body is a direct implementation of (2.2).

Notice that we can use a single variable  $r$  to represent the whole sequence  $x_n$ , since we only need the last element of the sequence. Notice also the different semantics of the  $=$  operator in (2.2) and in the Python program. In Python,  $=$  is an *assignment-operator*: we first evaluate the right-hand side, and then we assign the resulting value to variable on the left-hand side. In particular, it is perfectly fine to use the variable  $r$  on both the left- and the right-hand side of  $=$ . This just means that the new value of  $r$  is obtained by an expression that uses the old value of  $r$ . In (2.2),  $=$  is a *mathematical definition*. This means that we define a new quantity in terms of an old quantity. In Mathematics, we then expect that this new quantity has a new name.

After the main **while**-loop terminates, the program outputs the resulting approximation for  $\sqrt{a}$  on the screen.

### Data Types and Variables

#### Lernziele

**KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

**KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und -paradigmen.



Recall that a data type defines the possible values that a data object can have, as well as the possible operations that may be carried out on these values. Every programming language has its own collection of data types. The *type system* is one of the distinctive features of a programming language. We will now look at a concrete example, by going through some features of the type system of Python. Later in this class, we will also encounter another example of a type system when we learn about the programming language Scala.

#### 3.1 Primitive data types

Primitive data types have data objects that consist of a single atomic value, such as a number, a character, or a truth value. We will now look at the primitive data types that Python has to offer.

##### 3.1.1 NoneType

The data type `NoneType` allows only one possible value: **None**. This value is used to represent the *absence* of a result or a value. For example, we might want to write a program that is looking for a desired entry in a database. To indicate, that the desired entry is not present, the program might also return the value **None**.

In other languages, similar notions exist, but the value is called typically called differently, e.g., `nil` or `null`. There are no particular operations for this `NoneType`.

```
>>> x = None
>>> print(x)    # nothing is printed on the screen
>>>
```



### 3.1.2 Boolean

The data type `bool` represents Boolean truth values. These truth values behave like the truth values that we know from our discussion of propositional logic in *Diskrete Strukturen für Informatik*. There are two possible values for a data object of type `bool`: **True** or **False**. The data type supports the typical operations from propositional logic. In particular, there is **and** (where a **and** b is **True** if and only if both a and b are **True**), **or** (where a **or** b is **True** if and only if at least one of a or b is **True**), and **not** (where **not** a is **True** if and only if a is **False**).

```
>>> True and False
False
>>> True and False or True
True
>>> not True or (False and True)
False
```



### 3.1.3 Integer

The data type `int` represents signed integer numbers (whole numbers that can be positive, negative, or zero). Examples are 0, 1, -1, 2, 122 42, -1234. Some operations on integer numbers are:

- **Addition, subtraction, multiplication:** These operations behave exactly as we might expect:  $a + b$  computes the sum of the integers  $a$  and  $b$ ,  $a - b$  computes the difference of the integers  $a$  and  $b$ , and  $a * b$  computes the product of the integers  $a$  and  $b$ .

```
>>> 5 + 10
15
>>> 4 - 20
-16
>>> 4*0
0
>>> 12*(-2)+10
-14
```



- **Division:** In Python, there are *two* division operators for integers: *fractional division* (`//`) and *integer division* (`/`).

The result of a fractional division is a fractional number, as we would expect with the usual division from elementary school:

```
>>> 4/3
1.3333333333333333
>>> 5/2
2.5
>>> -5/2
-2.5
>>> -5/-2
2.5
>>> 10/5
2.0
```



As such, the result of a fractional division is not a data object of type `int`, but of type `float` (the data type in Python for fractional numbers, see below).

The result of an integer division is always an integer number. If necessary, the result is rounded to obtain an integer number. There is some variance between programming languages in how exactly this rounding is implemented. In particular, there are different possibilities what happens if one or both operands are negative. In Python, the result is always rounded down to the next *smaller* integer (in particular, this also applies to negative numbers).

```
>>> 4//3
1
>>> 5//2
2
>>> -5//2
-3
>>> -5//-2
-3
>>> -5//-2
2
>>> 10//5
2
```



- **Modulo:** To complement integer division, there is also a *modulo* operator (%) that computes the *remainder* of an integer division. Again, the details of the modulo operator may vary between programming languages, in particular, if negative operators are involved. In Python, `a % b` is implemented so that the result always has the same sign as `b` and an absolute value that is smaller than the absolute value of `b`.

```
>>> 5%3
2
>>> -5%3
1
>>> 5%-3
-1
>>> -5%-3
-2
```



```
>>> 10%2
0
```



- **Absolute value:** The function `abs(·)` returns the absolute value of an integer.

```
>>> abs(10)
10
>>> abs(-10)
10
>>> abs(0)
0
```



Many programming languages (not Python) also provide *unsigned integers*, where all possible values are at least 0.

**Integer issues.** In Python, integer numbers can be arbitrarily large. This is actually quite uncommon. In Section 2.1, we saw that computers store data in the form of data words of a fixed size. The arithmetic-logic-unit which carries out the actual computations is designed for this fixed size. Thus, in order to support integers of arbitrary size, Python cannot rely directly on the underlying hardware, but it needs to implement additional logic that makes operations on integer numbers slow.

For this reason, other languages like Scala, C++, or Java do not have arbitrarily large integer numbers. Their integer types are modeled on the underlying hardware and have a finite precision (typically 32 or 64 bits). Typically, this finite amount of bits is used to represent a number in the range between  $-2^{31}$  and  $2^{31} - 1$  or between  $-2^{63}$  and  $2^{63} - 1$ . The advantage is that now operations are much faster. The main disadvantage is that whenever the result of an arithmetic operation is not within range, a so-called *overflow error* occurs. In this case, the result may not be what we expect. Here is an example in Scala:

```
>>> val a: Int = 2147483647 // = 2^31 - 1 (within the range)
>>> val b: Int = a + 1      // = 2^31 (not within the range)
>>> println(b)             // OVERFLOW ERROR
-2147483648
```



Later in this chapter, we will discuss the exact representation of integer numbers in languages like Scala or C++. This will shed additional light on what happens during an overflow-error.

### 3.1.4 Float

The data type `float` represents fractional numbers. More precisely, data objects of type `float` are *floating-point numbers*, a *subset* of the rational numbers, represented in a special binary format (called IEEE 754). Some operations on floating-point numbers are:

- **Basic arithmetic operations.** Floating-point numbers support addition (+), subtraction (-), multiplication (\*), and (/) division.

```
>>> -5.0+2.2
-2.8
>>> -7*1.1
-7.7000000000000001
>>> 6/4
1.5
>>> 4/3
1.3333333333333333
```



The examples already show that we need to be a bit careful with floating-point numbers. Floats are only an *approximation* of rational numbers, with finite precision. We must always expect numerical inaccuracies and rounding errors.

- **Integer conversion.** The function `int(·)` converts a floating-point number into an integer, by removing the digits after the decimal point.

```
>>> int(1.1)
1
>>> int(1.7)
1
>>> int(-2.3)
-2
```



Many languages distinguish between the types **Float** (32 bits) and **Double** (64 bits). These two types behave essentially in the same way, but **Double** offers larger numerical precision.

**Floating-point issues.** We have a fixed number of bits (typically 32 or 64) to represent a floating-point number. Therefore, we can have only  $2^{32}$  or  $2^{64}$  different numbers. On the other hand, there are infinitely many different rational numbers. Thus, there are infinitely many rational numbers that cannot be represented by floating-point numbers. Moreover, floating-points numbers are represented as fractional numbers in the *binary* system. In particular, the decimal number 0.1 is a periodic fraction in the binary system (we have  $0.1_{10} = 0.0001100011\dots_2$ ). As such, it cannot be represented exactly in floating-point arithmetic.

This means: when working with floating-point numbers we must be aware that there will always be rounding errors and numerical inaccuracies. In particular, highly familiar mathematical laws which we take for granted, like the associative law, do not hold in the floating point world.

```
>>> 0.1*3
0.30000000000000004 # Suddenly some (very small) error appears.
>>> 1+0.0000000000000001
1.0 # Adding this very small number is "ignored", thus it does not help ...
```





[Draft] • (None)@(None) • [(None)]

- **Computing the length:** The function `len(·)` returns the *length* of a string `a`, i.e., it returns the number of characters in `a`.

```
>>> len("Hallo")
5
>>> len("Welt")
4
>>> len("")
0
```



- **Computing the position in the character table:** The function `ord(·)` returns the *ordinal number* of a character `c`, i.e., the position of `c` in the UNICODE-table. The function produces an error if its input is not a string of length 1 (a character).

```
>>> ord(" ")
32
>>> ord("A")
65
>>> ord("a")
97
>>> ord("Hallo")
TypeError: ord() expected a character, but string of length 5 found
```



- **Looking up a character in the character table:** The function `chr(·)` is the inverse function of `ord(·)`. It takes an integer `c` and returns the character that is at position `c` of the UNICODE table.

```
>>> chr(32)
' '
>>> chr(65)
'A'
>>> chr(200)
'È'
```



In Python, a character is the same as a string of length 1. Many other languages distinguish between strings of length 1 and characters. For example, in Scala there are the two data different types **String** and **Char**.

Python strings are *immutable*. This means that the contents of a string cannot be changed. In particular, the concatenation operator (+) creates a *new* string that contains the concatenation of its operands.

### 3.2.2 Lists

A *list* in Python represents a sequence of (an arbitrary number of) data objects. These data objects are called the *elements* of the list. In Python, the elements of a list can be of distinct types. The order of the elements matters, and the same data element can appear repeatedly in a list. For example,

```
a = [1, True, "Hello", 5.5, 1]
```



defines a list `a` that consists of 5 elements: the integer number 1, the truth value `True`, the String `"Hello"`, the floating-point number 5.5, and the integer number 1, in this order. Notice that the integer number 1 appear twice in `a`.

Lists are a central data structure in Python, and almost every Python program contains one more more lists.

**Index notation.** We can access individual elements a list using the *index notation* `a[i]`, starting with index 0. For example,

```
>>> a[1]
True
>>> a[4]
1
```



In Python, it is also possible to use *negative* index values. This means that the position is counted from the back of the list (where `a[-1]` denotes the *last* element of `a`). For example,

```
>>> a[-2]
5.5
```



We will get an error if we try to access an index that does not exist, e.g.,

```
>>> a[10]
IndexError: list index out of range
>>> a[-7]
IndexError: list index out of range
```



We can determine the number of elements in a list with `len(a)`.

```
>>> len(a)
5
```



The possible index values go from 0 to `len(a) - 1` and from `-1` to `-len(a)`. A specialty of Python are *slices*, a dedicated syntax to extract sublists from a given list. The notation is `a[i:j:s]`. The interpretation is very similar to the function `range(a, b, s)` that we saw in Section 2.4.3. More precisely, `a[i:j:s]` generates a sublist of `a` that starts at index `i` (included) and increases the index by increments of `s`, until the first index value that is larger or equal to `j` is reached (excluded).

```
>> a[1:4:2]
[True, 5.5]
>>> a[1:3:2]
[True]
```



If we omit *i*, it is taken to be 0, the first index in the list.

```
>>> a[:3:2]
[1, 'Hello']
>>> a[:2:2]
[1]
```



If we omit *j*, it is taken to be the largest index in the list.

```
>> a[1::2]
[True, 5.5]
>>> a[2::2]
['Hello', 1]
```



If we omit *s*, it is taken to be 1.

```
>>> a[1:4]
[True, 'Hello', 5.5]
>>> a[0:2]
[1, True]
```



We can also use a *negative* value for *s*. This means that we go through the list in reverse order.

```
>>> a[4:2:-1]
[1, 5.5]
>>> a[4:2:-2]
[1]
>>> a[4:1:-2]
[1, 'Hello']
```



Python lists are *mutable*. This means we can change entries of a list by using the index notation.

```
>>> a[1]
True
>>> a[1] = 7
>>> a
[1, 7, 'Hello', 5.5, 1]
```



It is important to note that the assignment statement `a[1] = 7` changes the *existing* list `a`. This means that this change is also visible for other variables that are associated with the same list (this is the first time where the *reference semantics* of the assignment statement in Python (see Section 2.4.1) becomes relevant). For example,

```
>>> b = a
>>> a[1] = 7
>>> b
```



```
[1, 7, "Hello", 5.5, 1]    # a and b refer to the same data object.  
                           # The change via a is also visible through b.
```



**Lists of lists.** A list can contain other lists. For example, we can have the following list

```
c = [[7,2,3], [1,2], [4,6], [3,4,5,6]]
```



The list `c` consists of 4 lists: the first list has 3 elements, the second list has 2 elements, the third list has 1 element, and the fourth list has four elements. We can use a double index notation to access the elements in the sublists. For example,

```
>>> c[1][0]  
1          # The first element of the second list.  
>>> c[3][2]  
5          # The third element of the fourth list.  
>>> c[2]  
[4, 6]     # The whole third list.
```



We call `c` a *two-dimensional list*. Of course, there are also three-dimensional and higher-dimensional lists. Almost every programming language supports a concept of two- or higher-dimensional lists, but it may be more restrictive than in Python.

**Attention:** Since Python is dynamically typed, it is possible that a list can have elements of different types. This is not common in other languages. Many languages, like Scala, Java, or C++, allow only lists in which all elements are of the same type. In particular, the list `a` from above would not be possible.



**Operations on lists.** Lists in Python support many operations. Here we mention a few popular examples.

- **Concatenation:** The *concatenation* (+) of two lists `a` and `b` is obtained by first writing all elements of list `a`, followed by all elements of list `b`. The operation is completely analogous to the concatenation of strings from Section 3.2.1.

```
>>> [4,5,6] + [1,2,3]  
[4,5,6,1,2,3]
```



- **Computing the length:** The function `len(.)` returns the *length* of a list `a`, i.e., it returns the number of elements in `a`. Again, this is completely analogous to strings (Section 3.2.1).

```
>>> len([1,2,3])  
3
```



```
>>> len([1,True,5.5, False, False])
5
>>> len([])
0
```



- **Repeating a list:** Given a list `a`, we can use the multiplication operator `j * a` to create a new list that is obtained by concatenating `a` `j` times with itself.

```
>>> 3 * [1,2,3]
[1,2,3,1,2,3,1,2,3]
```



- **Appending to list:** The operation `append` adds an element to an existing list:

```
>>> a = [1,2,3]
>>> b = a
>>> a.append(10)
>>> b
[1,2,3,10]    # append modifies an existing list.
```



After this, both `a` and `b` are still associated with the same, modified, list `[1,2,3,10]`. This should be contrasted with the concatenation operator `+`. Concatenation does not modify an existing list, but it creates a new list. For example, consider the following code:

```
>>> a = [1,2,3]
>>> b = a
>>> a = a + [10]
>>> b
[1,2,3]    # Concatenation creates a new list.
```



After this, `b` still refers to the old list `[1,2,3]`, while `a` refers to the new list `[1,2,3,10]` that was created by the concatenation operator.

When using an operation on lists, we must be aware whether this operation creates a new list or modifies an existing list. This information can typically be found in the Python documentation.

**Lists and Loops.** Finally, let us mention that lists can be used as ranges in **for**-loops (see Section 2.4.3). That is, we can write **for** `i` **in** `a`: to iterate over the element that are contained in the list `a`, in the order in which they appear in `a`. For example:

```
>>> for x in [1, True, "Hello", 5.5, 1]:
...     print(x)
1
True
Hello
5.5
1
```



In fact, **for**-loops and lists are commonly used together in programs, because lists can contain an arbitrary number of elements and **for**-loops offer the best way to consider all these elements one by one.

### 3.2.3 Tuples

Tuples in Python are very similar to lists. The syntax for tuples is almost the same, except that tuples are written with round parentheses instead of square brackets.

```
tup = (1, True, "Hello", 5.5, 1) # Defines a tuple.
```



The index operator for tuples uses square brackets, just like for lists.

```
>>> tup[1]
True # The second element of tup.
```



The big difference between lists and tuples in Python is that tuples are *immutable*, whereas lists are *mutable*. That is, we can make changes to the elements of an existing list, whereas we cannot make changes to the elements of an existing tuple.

```
>>> tup[1] = 100
TypeError: 'tuple' object does not support item assignment
```



Tuples support similar operations as lists, except for those operations that modify an existing list (e.g., concatenation is also available for tuples, but the append-operator is not).

```
>>> (4,5,6) + (1,2,3)
(4, 5, 6, 1, 2, 3)
>>> len((1,2,3))
3
>>> len((1, True, 5.5, False, False))
5
>>> len(())
0
>>> 3 * (1,2,3)
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> a = (1,2,3)
>>> a.append(10)
AttributeError: 'tuple' object has no attribute 'append'
```



### 3.2.4 Dictionaries

Dictionaries are a particularly convenient feature of Python. Dictionaries store *key-value* pairs, where a *key* can appear at most once and can be used to look up a value. A key-value pair is called an *item*. For example, we can write

```
dict = {1: "one", 2: "two", 3: "three"}
```



to create a dictionary with three items. The first item has key 1 and value "one", the second item has key 2 and value "two", and the third item has key 3 and value "three".

We can retrieve the value for a given key using the index notation.

```
>>> dict[2]
'two'                                     # Obtain the value for key 2.
```



If we try to access the value for a key that does not exist, we get an error.

```
>>> dict[10]
KeyError: 10                          # There is no item with key 10.
```



Dictionaries are *mutable*. There are several ways to modify the items that are stored in an existing dictionary. For example, the assignment

```
dict[key] = expression
```



has two possible effects: if key already exists in `dict`, the value is changed to the value of expression.

```
>>> dict[2] = "dos"
>>> dict[2]
'dos'                                     # The value for key 2 has been changed to 'dos'.
>>> dict
{1: 'one', 2: 'dos', 3: 'three'}         # A key appears at most once.
```



If key does not exist in `dict`, we create a new item with key key.

```
>>> dict[4] = "four"
>>> dict[4]
'four'                                     # There is a new item with key 4.
>>> dict
{1: 'one', 2: 'dos', 3: 'three', 4: 'four'}
```



To remove an item with key key, we write `del dict[key]`.

```
>>> del dict[4]
>>> dict[4]
KeyError: 4                          # The item with key 4 has been removed.
>>> dict
{1: 'one', 2: 'dos', 3: 'three'}
```





Every Python object can be used as a value in a dictionary, but not every Python object can be used as a key. For examples, lists and dictionaries cannot be used as keys, but all simple Python types (integers, floats, strings) are possible.

We can use dictionaries in **for**-loops. If we use the dictionary itself as a range, we iterate over the keys of the dictionary (the same happens if we use `dict.keys()`). To iterate over all values, we must use `dict.values()`. To iterate over all items (tuples that represent the key-value pairs in the dictionary), we use `dict.items()`.



```
>>> dict = {1: "one", 2: "two", 3: "three"}
>>> dict.keys()
[1, 2, 3]
>>> dict.values()
["one", "two", "three"]
>>> dict.items()
[(1, "one"), (2, "two"), (3, "three")]
...     print(k)
1
2
3
>>> for k in dict.keys():
...     print(k)
1
2
3
>>> for v in dict.values():
...     print(v)
one
two
three
>> for (k, v) in dict.items():
...     print("key: " + str(k) + ", value: " + str(v))
key: 1, value: one
key: 2, value: two
key: 3, value: three
```

### 3.3 Data representation

In Section 3.1, we saw that Python (and other programming languages) support a wide range of primitive data types. On the other hand, in Section 2.1, we saw that modern computers represent data only as a sequence of data words with a fixed number of bits (typically 32 or 64). How does this go together?

Of course, the answer is that *internally*, all data is represented simply as *bitstrings* (sequences of bits). The only thing that makes these bitstrings into truth values, (signed or unsigned) integers, fractional numbers, characters, etc. is how we *interpret* them.

Over time, computer scientists have come up with many ways of going between general data items and bitstrings. We say that the general data items are *encoded* as bit

strings. We will now discuss some common encodings for a few of the data types that we have encountered in Section 3.1.

### 3.3.1 Truth values

The easiest way to encode a truth value as a bitstring is by using a single bit: 0 represents **False**, and 1 represents **True**. However, since the RAM in a modern computer is structured in *data words*, and not in individual bits, it is also common to interpret the bit string 000...000 that consists of 32 or 64 zero bits as **False**, and any other bitstring as **True**.

### 3.3.2 Unsigned integers

Let us first recall how we learned to represent numbers in elementary school. For this, we use the *decimal system*. There are ten *digits*: 0,1,2,3,4,5,6,7,8,9; and we write a number as a sequence of digits. For example, 100, 42, or 66343221 are representations of numbers.

The meaning of a digit depends on the *position* of the digit in the number.<sup>2</sup> The last digit represents multiples of 1, the second to last digit represents multiples of 10, the third to last digit represents multiples of 100, and so on, for increasing powers of 10. For example, we have

$$73282 = 7 \cdot 10000 + 3 \cdot 1000 + 2 \cdot 100 + 8 \cdot 10 + 2 \cdot 1,$$

or, to make the powers of 10 more explicit,

$$73282 = 7 \cdot 10^5 + 3 \cdot 10^4 + 2 \cdot 10^3 + 8 \cdot 10^2 + 2 \cdot 10^1.$$

Note that the digit 2 appears twice in 73282, with two different values: 200 and 2.

In general, an  $n$ -digit number  $d_{n-1}d_{n-2}\dots d_0$  in the decimal system (where each  $d_i$  is a digit from 0 to 9) is interpreted as

$$d_{n-1}d_{n-2}\dots d_0 = d_{n-1} \cdot 10^{n-1} + d_{n-2} \cdot 10^{n-2} + \dots + d_0 \cdot 10^0,$$

or, more succinctly,

$$d_{n-1}d_{n-2}\dots d_0 = \sum_{i=0}^{n-1} d_i \cdot 10^i.$$

Here, we chose the indices for the digits to match the powers of 10 (from right to left, starting with 0).

Of course, there is nothing special about using powers of ten (except for the fact that most humans have ten fingers). Indeed, instead of working with ten digits, we could do the following:

---

<sup>2</sup>This is why the decimal system (the Hindu-Arabic system) is also called a *positional* system. In contrast the Roman system is called an *additive* system.

1. pick any integer  $b \geq 2$  (the *base*);
2. invent symbols for  $b$  different digits; and
3. interpret to positions of the digits as powers of  $b$ .

If we do this, we work in a *positional system with basis  $b$* .

In bitstrings, we have only two digits: 0 and 1. Thus, it is natural to interpret a bitstring as a positional number with basis 2. More precisely, let  $x = b_{n-1}b_{n-2}\dots b_0$  be a string of  $n$  bits (indexed from right to left, starting with zero), where each  $b_i$  is either 0 or 1. Then, we can interpret  $x$  as the number

$$b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_0 \cdot 2^0 = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

For example, we can interpret the bitstring 10011 as

$$\begin{aligned} 10011_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 16 + 0 + 0 + 2 + 1 \\ &= 19_{10}, \end{aligned}$$

where we use the subscripts 2 and 10 to indicate the different bases.

Now, suppose that our computer offers data words of 64 bits. Then, we can use strings of 64 bits to represent an unsigned integer with a single data word. Thus, we can represent all numbers from 0 to the unsigned integer that corresponds to the bitstring 1111...1 that consists of 64 1-bits. This is

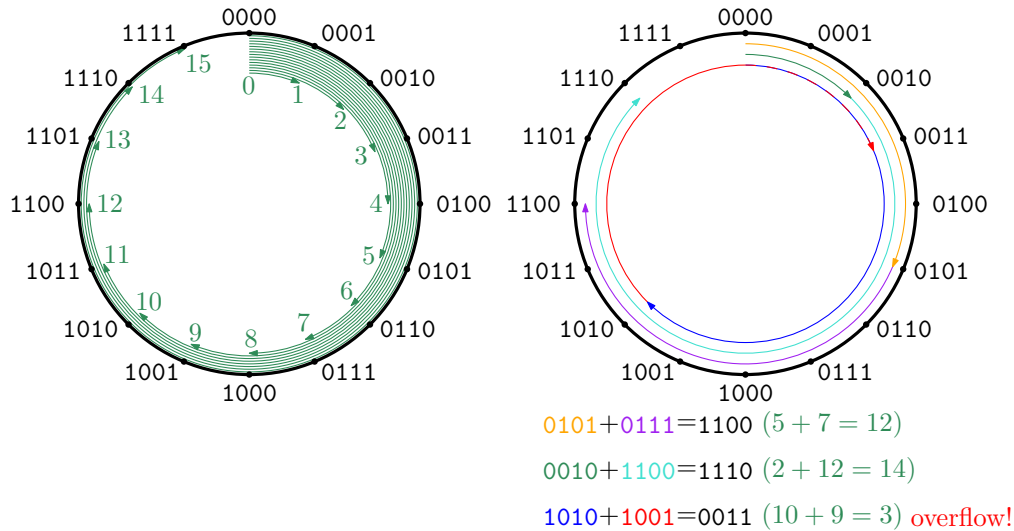
$$\sum_{i=0}^{64-1} 1 \cdot 2^i = \sum_{i=0}^{63} 2^i = 2^{64} - 1,$$

using the formula for the geometric series from *Diskrete Strukturen für Informatik*. Thus, with a single 64-bit data word, we can represent the unsigned integers from 0 to  $2^{64} - 1$ .

If an addition or a multiplication results in a number that needs more than the available number of bits, the upper bits are simply cut off. This is called an *overflow* error. For example, adding 10 to  $2^{64} - 1$  would not result in  $2^{64} + 9$ , as we might expect, but in 9. The bit in position 64 is lost.

### 3.3.3 Signed integers

Now we would like to extend our encoding of integer numbers from Section 3.3.2 to include *negative* numbers. There are several ways to do this. Here, we describe the *two's complement representation*. This is the most common encoding for negative numbers in modern computers.



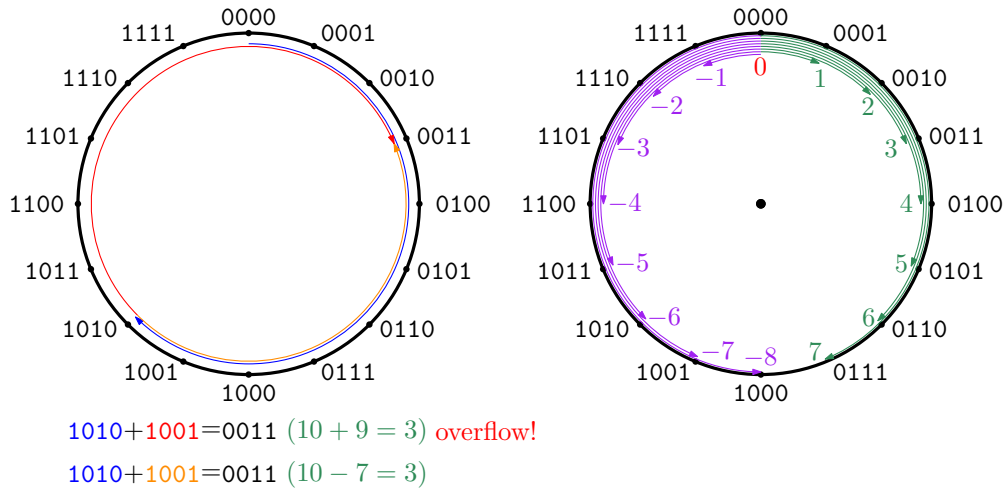
**Abbildung 3.1:** We interpret the 4-bit strings as the unsigned integers from 0 to 15. We can imagine the bitstrings as positions on a circle (left). Addition of bitstrings corresponds to moving along the circle: adding  $x$  to  $y$  means that we take  $\langle y \rangle$  clockwise steps from the position for  $x$ , possibly incurring an overflow (right).

The idea of the two's complement is as follows: suppose that we work with a fixed word length. For concreteness, let us pick a word length of 64 bits. In Section 3.3.2, we saw how to interpret 64-bit strings as unsigned integers between 0 and  $2^{64} - 1$ . Furthermore, we used this interpretation to define an *addition operation* on 64-bit strings. This addition operation *approximates* the addition operation on actual unsigned integers. However, it is *not the same*. In particular, the addition on 64-bit strings cuts off the result at 64 bits. This can lead to overflows. If we go beyond  $2^{64} - 1$ , we do not reach  $2^{64}$ , but we come back to 0.

Therefore, we can imagine that the addition for our 64-bit strings does not happen on a *line*, but on a *circle*. More precisely, we can imagine that we have a circle with  $2^{64}$  positions, one for each 64-bit string. Adding two 64-bit strings  $x$  and  $y$  means that we start at the position for the bitstring  $x$ , and that we take  $\langle y \rangle$  clockwise steps, arriving at the bitstring that represents the sum of  $x$  and  $y$ . Here,  $\langle y \rangle$  denotes the *interpretation* of  $y$  as an unsigned integer from Section 3.3.2. See Figure 3.1 for an example with 4 bits.

Now, the main observation is this: since our circle has  $2^{64}$  positions, taking  $\langle y \rangle$  *clockwise* steps from position  $x$  is the same as taking  $2^{64} - \langle y \rangle$  *counter-clockwise* steps from position  $x$ . In each case, we arrive at the same place. Thus, there are two possible ways to interpret the bitstring addition of  $x$  and  $y$  on the circle: either, we *add*  $\langle y \rangle$  to  $x$ , or we *subtract*  $2^{64} - \langle y \rangle$  from  $x$ . Both interpretations are compatible with the bitstring addition from Section 3.3.2.

As a consequence, there are now two different ways to interpret a bitstring  $x$ : either as the *positive* number  $\langle x \rangle$ , or as the *negative* number  $-(2^{64} - \langle x \rangle)$ . For example, we



**Abbildung 3.2:** We can interpret the addition of two bitstrings  $x$  and  $y$  in two different ways: either we take  $\langle y \rangle$  *clockwise* steps from  $x$ , or we take  $16 - \langle y \rangle$  *counterclockwise* steps to the right from  $x$ . In the first case, we *add*  $\langle y \rangle$  to  $x$ , in the latter case, we *subtract*  $(16 - \langle y \rangle)$  from  $x$  (left). We are now free to interpret each bitstring  $x \neq 0000$  as either the positive integer  $\langle x \rangle$  or as the negative integer  $-(16 - \langle x \rangle)$ . A natural way to do this is in such a way that the leftmost bit of  $x$  represents the sign of the corresponding number (right).

could interpret the bitstring  $00 \dots 01$  either as 1 or as  $-2^{64} + 1$ , the bitstring  $00 \dots 011$  either as 2 or as  $-2^{64} + 3$ , the bitstring  $11 \dots 10$  either as  $2^{64} - 2$  or as  $-2$ , and the bitstring  $11 \dots 1$  either as  $2^{64} - 1$  or as  $-1$ . In principle, we are free to choose an arbitrary such interpretation for every bitstring. However, to get a useful addition operation, we should choose this interpretation in such a way that we get contiguous block of integers.

For this, we pick a *cut-off* bitstring  $z$ , and we interpret all bitstrings that come before and including  $z$  as nonnegative numbers, and all bitstrings that come after  $z$  as negative numbers. In this way, we can represent the integers from  $-(2^{64} - \langle z \rangle - 1)$  to  $\langle z \rangle$ . The standard choice for the cut-off is  $z = 01 \dots 1$ . This leads to the range of integers from  $-2^{63}$  to  $2^{63} - 1$ . See Figure 3.2 for an illustration with 4 bits.

There are two reasons for this choice of the cut-off bitstring: (a) it almost balances the number of possible positive and possible negative numbers; and (b) to determine the sign of a number, it suffices to check the highest bit. More precisely, a number is negative if and only if the highest bit is 1. Note that the bitstrings and the addition operation did not change when going from the unsigned integer interpretation to the two's complement interpretation. The only thing that changes is how we *interpret* the bitstrings as numbers. This is a major advantage of the two's complement: our hardware needs to implement only a single addition operation on bitstrings. This operation can be used both for signed and for unsigned addition. Note also that in the

two's complement interpretation, overflows also occur, albeit at a different place: if we go beyond  $2^{63} - 1$ , we arrive at  $-2^{63}$ , and if we go beyond  $-2^{63}$ , we arrive at  $2^{63} - 1$ . This is exactly the behavior that we observed for Scala-integers in Section 3.1.3. Let us mention that everything we have said so far also holds for *multiplication*: it suffices to implement a single multiplication operation, for both unsigned and signed integers.<sup>3</sup>

In the two's complement, there is an easy way to implement a *negation operation*: given a bitstring  $x$ , find the bitstring  $x'$  such that  $x$  and  $x'$  represent numbers with the same absolute value and opposing signs. The procedure is as follows:

1. We invert all the bits of  $x$ , flipping all 0s to 1s and all 1s to 0s.
2. We take one clockwise step from the resulting bitstring.

This procedure works for all bitstrings that are different from  $00\dots 0$  and  $10\dots 0$ . For  $00\dots 0$  and  $10\dots 0$ , the result is again  $00\dots 0$  and  $10\dots 0$ , respectively. For  $00\dots 0$ , this is consistent with our interpretation, because we have  $-0 = 0$ . For  $10\dots 0$ , this is a shortcoming of the two's complement interpretation: there is no positive partner for the negative number  $-2^{63}$ .

### 3.3.4 Floating-point numbers

The most common way to interpret bitstrings as fractional numbers in a computer is the *floating-point representation*. Before we can describe how this works, let us recall how fractional numbers are represented in the decimal system. As described in Section 3.3.2, a number is represented in the decimal system as a sequence of digits, where the position of a digit determines its meaning. In order to extend this representation for fractional numbers, we add one more feature: the *decimal point* “.”. The positions to the *left* of the decimal points are interpreted as in Section 3.3.2, as *increasing* powers of ten, starting from  $10^0$ . The positions to the *right* of the decimal point are interpreted as *decreasing* powers of ten, starting from  $10^{-1}$ . For example, we have

$$82.565 = 8 \cdot 10 + 2 \cdot 1. + 5 \cdot 0.1 + 6 \cdot 0.01 + 5 \cdot 0.001,$$

or, more succinctly,

$$82.565 = 8 \cdot 10^1 + 2 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2} + 5 \cdot 10^{-3}.$$

It is straightforward to generalize this representation to the binary system, using powers of two instead of powers of ten. For example, we have

$$101.1101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$$

---

<sup>3</sup>For the mathematically inclined reader, let us say that the deeper reason for this phenomenon lies in the following fact: the addition and multiplication operations on bitstrings that we obtain by cutting off at 64 bits exactly correspond to addition and multiplication in the *ring of integers* modulo  $2^{64}$ . The elements of this ring are the congruence classes modulo  $2^{64}$ , and we are free to choose a representative for each congruence class. Depending on this choice, we either get the unsigned integers or the two's complement integers.

$$\begin{aligned}
 &= 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot 0.5 + 1 \cdot 0.25 + 0 \cdot 0.125 + 1 \cdot 0.0625 \\
 &= 5.8125_{10}.
 \end{aligned}$$

Now, how can we interpret a 64-bit string as a fractional binary number? A simple way would be to choose a fixed position in the bitstring for the decimal point<sup>4</sup>, and to interpret all the bits to the left of this position as positive powers of two, and all the bits to the right of this position as negative powers of two. For example, if we decide that the decimal point comes after the third bit, we would interpret the bitstring 01101010 as the fractional binary number  $11.0101_2$ , and the bitstring 10100011 as the fractional binary number  $101.00011_2$ . This is called the *fixed-point representation* of fractional numbers.

A disadvantage of the fixed-point representation is that we must commit to a position of the decimal point. This restricts the range of numbers that we can represent. If we know that all the numbers that appear in our program are of approximately the same magnitude, this is fine, and fixed-point representation is a good choice. However, if we do not know in advance which magnitudes to expect, it is better to have a more flexible representation.

In the floating-point representation, we partition the bitstring into three parts: (i) the *sign-bit*  $s$ ; (ii) the *exponent*  $p$ ; and (iii) the *significand*  $m$  (also called the *mantissa*). The sign bit is a single bit that determines whether the number is positive ( $s = 0$ ) or negative ( $s = 1$ ). The exponent is interpreted as a signed integer, and it determines the position of the decimal point. The significand  $m$  is a  $k$ -bit string  $m_1 m_2 \dots m_k$ , where the bits  $m_1, m_2, \dots, m_k$  correspond to negative powers of 2. More precisely, the floating-point interpretation of the bitstring is

$$(-1)^s \cdot (1.m_1 m_2 \dots m_k)_2 \cdot 2^p.$$

This interpretation directly implies that the number is positive for  $s = 0$  and negative for  $s = 1$ . The main part  $(1.m_1 m_2 \dots m_k)_2$  is the binary fractional number that has 1 to the left of the decimal point and the bits of the significand to the right of the decimal point. By fixing the bit in front of the decimal point to 1, we can save one bit in our floating-point representation. We can always use  $p$  to shift the decimal point such that our number is of this form.<sup>5</sup> The exponent  $p$  determines the position of the bitstring: if  $p$  is positive, the decimal point is moved  $p$  positions to the right. If  $p$  is negative, the decimal point is moved  $-p$  positions to the left. In this way, we always have  $k + 1$  meaningful bits to represent our fractional number, and these bits can appear at varying (*floating*) positions relative to the decimal point.<sup>6</sup>

Nowadays, there is an established standard (called IEEE 754) that specifies exactly how bitstrings can be interpreted as floating point numbers. For example, the IEEE-754-standard describes for 32- and 64-bit strings where the sign bit, the bits for the

<sup>4</sup>We will use the term “decimal point” also for fractional numbers in the binary system.

<sup>5</sup>Except if we try to represent the number 0. To fix this issue, the number 0 is represented by a special bitstring. We omit the details here.

<sup>6</sup>You may already have encountered this kind of representation in the decimal system. There, it is called the *scientific notation* for a decimal number.



exponent, and the bits for the significant are located and how to interpret the bits for the exponent as a signed integer. The details for 32- and 64-bit strings are very similar, but 64-bit strings have more bits for the exponent and the significand (this explains the difference between the **Single** and **Double** data types mentioned in Section 3.1.4). The IEEE 754 standard also defines special encodings for the numbers 0,  $\infty$ ,  $-\infty$ , and NaN (not a number), and it makes rules for the exact implementation of floating-point operations. This has come as a big relief to programmers, because before the advent of IEEE 754, there was a huge variety between different hardware types and programming languages as to how floating-point arithmetic was implemented.

### 3.3.5 Characters

At this point in our discussion, the interpretation of a bitstring as a character is quite straightforward. We define a fixed table that contains a list of possible characters, and we number the symbols in an arbitrary way, starting from 0. Then, given a bitstring  $x$ , we determine the unsigned integer  $\langle x \rangle$  that is represented by  $x$ , and we interpret  $x$  as the character that is situated at position  $\langle x \rangle$  in our character table.

Of course, if we proceed like this, we are completely free in choosing our character table. Indeed, over the years, many possibilities have been suggested. By now, there is an established standard for this character table, called UNICODE. However, competing standards are still used, and this can sometimes lead to unexpected behavior in our programs.

There is also a variant of UNICODE, called UTF-8, that uses bitstrings of varying length to encode the positions in the UNICODE table. The idea is to save space by using fewer bits to represent more commonly used characters. We will not go into the details here.

## 3.4 The five facets of a variable

We conclude this chapter with a small recap of what we have seen so far. In most imperative programming languages a variable is characterized by at least five different properties—the five *facets* of a variable. Let us see what they are.

### 3.4.1 Content

The *content* of a variable is the specific data object with which the variable is currently associated. More precisely, the content is a bitstring that represents the current data object. For example, after the assignment operation

```
x = 42
```



the content of the variable  $x$  in Python is the bitstring 101010.



### 3.4.2 Location

The *location* (or *address*) of a variable is the position of the variable content in memory. More precisely, if the data object consists of more than one memory cell, the location of the variable is the address of the first memory cell where this data object is stored. In Python (and in Scala), it is not possible to determine the exact location of a variable. However, other programming languages that are closer to the hardware, such as C, C++, or Rust, are able to determine and manipulate the location of a variable.

Nevertheless, in Python every object has a unique *identifier* that can be obtained by the function `id(·)`. In practice, this identifier often corresponds to the data object's memory location, but this behavior is not guaranteed.

### 3.4.3 Size

Every data object has a *size*—the number of bits that are used to represent the data object. In Python (and in Scala), it is very difficult to determine the exact size of a data object. Hence, we will ignore size for this lecture. However, languages that are closer to the hardware (e.g., C, C++, and Rust) can easily determine the size of data objects.

### 3.4.4 Data type

A *data type* is a collection of possible *values of the same kind*. The data type of a data object tells us what possible values the data object can have and which *operations* can be applied to these values. Whenever a variable in Python is associated with a data object, the type of this variable corresponds to the type of the associated object. In Python, it is possible to get the determine the type of the data object that is currently associated with a variable by using the function `type(·)`.

**Attention:** Recall that Python is *dynamically typed*. This means in particular that the data type and the location of a variable can change over time. The following example demonstrates this behavior:

```
>>> a = 0
>>> type(a)
<class int>
>>> a = True
>>> type(a)
<class bool>
```



In statically typed languages like Java, Scala, C++, or C, this does not happen.

### 3.4.5 Scope

The *scope* of a variable is the range of the program in which the variable can be used. The scope begins with the *declaration* of the variable. In Python, this means that the

scope begins with the first assignment *to* the variable, e.g., with a statement of the form `x = expression`. After this, the variable `x` can be used in other expressions within in the program. In Section 4.1.2, we will say much more about the notion of a scope when we discuss functions and the difference between *global* and *local* variables.

**Attention:** Since Python is a highly dynamic language, the following program will start but might produce a runtime error.

```
n = int(input("Please enter a number: "))
if n <= 10:
    x = 12
print(x)
```



Whenever the user inputs a number that is larger than 10, the assignment `x = 12` is not executed and the variable `x` is not declared. In this case, `print(x)` causes a **NameError**.



### Subroutines and Functions

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und -paradigmen.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.



#### 4.1 Fundamentals of subroutines

Many imperative programming languages offer a way to define *subroutines* and *functions*. This allows for much more structured and sophisticated programs at a much larger scale. At a very basic level, a *subroutine* lets us collect a sequence of instructions into a unit that has a name. To execute this sequence of instructions in our program, we can *call* or *invoke* the subroutine by using its name. In Python, this is done as follows:

```
def name():  
    block
```



This defines a subroutine `name` that consists of the sequence of instructions `block`. To invoke this subroutine, we just write `name()` in our program, like this:

```
>>> def greet():  
...     print("Hello, nice to meet you.")  
...     print("How are you doing?")  
...
```



```
>>> x = 1
>>> greet()
Hello, nice to meet you.
How are you doing?
```



A subroutine may receive *parameters*. Parameters are variables that pass information to the subroutine when it is called and that can be used inside the subroutine. The parameters must be *declared* in the definition of the subroutine, and the subroutine call must conform exactly to this declaration. For example:

```
>>> def greet(firstname, lastname):
...     print("Hello " + firstname + " " + lastname + ", nice to meet you.")
...     print("How are you doing?")
...
>>> greet("Helmut", "Rote")
Hello Helmut Rote, nice to meet you.
How are you doing?
>>> greet("Günter", "Alt")
Hello Günter Alt, nice to meet you.
How are you doing?
```



In this example, the greet-subroutine has two parameters: `firstname` and `lastname`. These parameters define the first and the last name of the person that should be greeted. Each time we call the subroutine `greet`, we can assign new values to these parameters. In this way, we can customize the behavior of the subroutine for each invocation.

When using the subroutine, we must use exactly two parameters.

```
>>> greet("Max")
TypeError: greet() missing 1 required positional argument: 'lastname'
>>> greet("Katharina", "Willert", "Klost")
TypeError: greet() takes 2 positional arguments but 3 were given
```



In Python, it is possible to define default values for parameters. In this case, we can omit the corresponding parameter in the function call. In this case, the parameter will be initialized with the given default value. For example:

```
>>> def greet(firstname, lastname = "Wang"):
...     print("Hello " + firstname + " " + lastname + ", nice to meet you.")
...     print("How are you doing?")
...
>>> greet("Helmut", "Rote")
Hello Helmut Rote, nice to meet you.
How are you doing?
>>> greet("Günter")
Hello Günter Wang, nice to meet you.
How are you doing?
```



Now, the second function call is valid. The last name of Günter defaults to Wang.

### 4.1.1 Functions

A subroutine may also have a *return value*. In this case, the subroutine is called a *function*. The value is returned using the **return**-statement. The **return**-statement also finishes the execution of the function, even if there are instructions that come after it. For example,

```
def square(x):  
    return x*x  
  
def greet2(timeofday):  
    print("Good " + timeofday + ".")  
    name = input("What is your name? ")  
    return name
```

A function may be used in an expression. When the expression is evaluated, the function is called and the return value is used as the value of the function in the expression. For example,

```
>>> z = square(5) + square(10)  
>>> z  
125  
>>> name = "Person " + greet2("morning")  
Good morning.  
What is your name? Max  
>>> name  
'Person Max'
```

In Python, *every* subroutine is a function and may be used in an expression (in contrast, other programming languages such as Pascal, C, or Java provide a special syntax to distinguish between subroutines/procedures and functions). If there is no **return** statement, the function value is **None**, the only value of `NoneType` (see Section 3.1.1). We can also use a return statement without a value. This is interpreted as returning **None**.

**Attention:** There is an crucial difference between the function `print(·)` and the **return**-statement: The function `print(·)` simply prints something on the screen. It has return value **None**. In contrast, the **return**-statement is used to pass the value to the invoking instance. This difference is highly relevant for the exact *specification* of a functions. For example, the following function prints 42 on the screen but returns 0.

```
>>> def func():
...     print(42)
...     return 0
...
>>> a = func() + 3
42
>>> print(a)
3
```



### 4.1.2 Scope

To make a program more structure, certain identifiers are valid only in a given part of the code, and they cannot be used in another part of the program. The part of the program where a given identifier  $x$  can be used is called the *scope* of  $x$ . We already encountered this notion in Section 3.4.5.

Now, every function in Python defines a *local scope*. A variable name that is created inside a function  $f$  is not visible outside of  $f$ . For example:

```
>>> def f():
...     x = 1
...
>>> f()
>>> print(x)
NameError: name 'x' is not defined
```



This code produces an error, because the variable  $x$  is defined only in the local scope of the function  $f$ , and it cannot be used outside of  $f$ .

Scopes can be *nested*. For example, the scope of the main program is called the *global scope*, and variable names that are in the global scope also appear in the local scope of functions that are defined inside the main program:

```
>>> x = 10
>>> def f():
...     print(x)
...
>>> f()
10
>>> print(x)
10
```



It may happen that we define a variable in a local scope of a function that has the same name as a variable in the outer scope. This defines two *different* variables *with the same name*. In the inner scope, we can access only the local variable, and once we leave the inner scope, the local variable disappears, and name again refers to the variable in the global scope. We say that the variable in the inner scope *shadows* the variable in the outer scope:

```
>>> x = 10
>>> def f():
...     x = 20
...     print(x)
...
>>> print(x)
10
>>> f()
20
>>> print(x)
10
```

Since Python creates new variables through assignment statements, there is a possible ambiguity: whenever we *assign* to a new variable *x* in the local scope of a function *f*, Python assumes that we would like to use *x* as a local variable for *f*. Now, if the same variable name *x* also appears in the global scope, and if this variable name is used in *f* before the first assignment to *x*, we get an error. Python assumes that we wanted to use *x* as a local variable, but that we forgot to initialize it on time.

```
>>> x = 10
>>> def f():
...     print(x)
...     x = 20
...
>>> f()
UnboundLocalError: cannot access local variable 'x'
where it is not associated with a value
```

We can use the **global** keyword to instruct Python not to create a new local variable but to use the variable from the global scope:

```
>>> x = 10
>>> def f():
...     global x
...     print(x)
...     x = 20
...     print(x)
...
>>> print(x)
10
>>> f()
10
20
>>> print(x)
20
```

**Attention:** Even though Python allows for global variables, it is considered bad programming practice to use them. The reason is that global variables can lead to changes



of the state that we may not expect. If we call a function *f* and forget that *f* changes a global variable, this may have unintended consequences:

```
>>> x = 1
>>> def square(y):
...     global x
...     x = "Be there or be square."
...     return y*y
...
>>> z = square(5)
>>> print(x + z)           # Oops.
TypeError: can only concatenate str (not "int") to str
```

We say that a function that changes the global state has *side-effects*. In general, since side-effects can easily lead to nasty programming errors that are difficult to find, we would like to avoid them as much as possible. Thus, we should use global variables *very* sparingly, if at all.

### 4.1.3 Nested functions and closures

In Python, a function may define its own subfunctions. This is called a *nested function*. The scope of the nested function include the local scope of the function that contains it.

```
>>> def func(x):
...     # y is a local variable of func.
...     y = 2
...     # g is a nested function of func.
...     # In particular, g can access the local
...     # variable y of func.
...     def g(z):
...         return z + y
...     # We can use g as a normal function inside func.
...     a = g(x)
...     a = x + g(5)
...     return a
...
>>> func(10)
17
# The function g lies in the local scope of func. We cannot
# use g in the global scope
>>> g(1)
NameError: name 'nested' is not defined
```

Functions in Python are “first-class citizens”. In particular, a function in Python can return a nested function that can then be used as a function outside.



```
>> def func():
...     # greeter is a nested function of func.
...     def greeter():
...         print("Hello.")
...         # The function greeter is returned by func.
...         return greeter
...
# We can use the result of func just like any other function.
>>> g = func()
>>> g()
Hello.
>>> g()
Hello.
```

We can also return a nested function that uses local variables from the surrounding function. In this case, the local variables from the surrounding function *survive* the end of the call to the surrounding function. They are still available when calling the nested function. This language feature is called a *closure*.

```
>>> def func(name):
...     # greeter is a nested function of func that uses the local variable name.
...     def greeter():
...         print("Hello " + name + "!")
...         # The function greeter is returned by func.
...         return greeter
...
# In this call to func, the value of the local variable name is "Max"
>>> hiMax = func("Max")
# In this call to func, the value of the local variable name is "Wolf"
>>> hiWolf = func("Wolf")
# When calling hiMax, the local variable with name "Max" is still available.
>>> hiMax()
Hello Max!
>>> hiMax()
Hello Max!
# When calling hiWolf, the local variable with name "Wolf" is still available.
>>> hiWolf()
Hello Wolf!
```

The local variables that are captured in a closure can also be modified.

```
>>> def func():
...     # a is a local variable of func.
...     a = 0
...     def inc():
...         # nonlocal indicates that a should refer to a variable from the
...         # enclosing (but not the global) scope.
...         nonlocal a
...         a = a + 1
```

```
...     print("My a is " + str(a) + ".")
...     return inc
...
# We obtain two inc-functions. Each of them has its own variable a that
# survives from the scope of the surrounding function.
>>> inc1 = func()
>>> inc2 = func()
>>> inc1()
My a is 1.
>>> inc1()
My a is 2.
>>> inc1()
My a is 3.
>>> inc2()
My a is 1.
>>> inc1()
My a is 4.
>>> inc2()
My a is 2.
```

## 4.2 Parameter passing strategies

We now discuss what happens exactly when a function is called. We saw that every function in our program must be *defined* somewhere. This definition provides the name of the function, the function body, and the parameters that the function receives. Once a function is defined, we can *call* it in our program, as often as we like. For each function call, we give an individual list of parameters that are used for this particular function call.

To distinguish between these two roles of the parameters, we call the parameters in the function definition the *formal parameters* of the function, and the parameters that are provided at a function call the *actual parameters* (for this particular function call).

The formal parameters are identifiers that can be used as local variable names in the function body. The actual parameters can be names of variables in the calling scope, or, more generally, they can be expressions that need to be evaluated.

```
# The formal parameters of func are a and b.
# They can be used as local variables inside of func.
>>> def func(a, b):
...     b = 10*b + a
...     a = 2*b
...     return a + b
...
>>> x = 2
>>> y = 10
# The actual parameters for this call to func are x and y.
>>> func(x, y)
306
```

```
# The actual parameters for this call to func are y and the expression -42+7.
>>> func(y, -42+4)
-1110
# The actual parameters for this call to func are -2*x and -12*y.
>>> func(-2*x, -12*y)
-3612
```

Now, when a function is called, we need to establish a connection between the formal parameters of the function and the actual parameters of the function call. We say that the actual parameters must be *bound* to the formal parameters.

There are many ways how this can be done. The details differ from programming language to programming language. We call this process the *parameter passing strategy* of the programming language. In the following, we will give three examples of common parameter passing strategies and discuss some of the relevant issues.

**Attention:** There is a wide variety of parameter passing strategies, and the differences can be subtle. Even worse, the terminology is not consistent. It can happen that the same name is used for slightly different parameter passing strategies. Thus, when encountering a new programming language, it is a good idea to learn about the details of the parameter passing strategy for this language, and to try a few examples is something seems unclear.



### 4.2.1 Call by value

*Call by value* is perhaps the simplest way to bind the actual parameters to the formal parameters: we can imagine that before the function body is executed, there is an *assignment statement* for each formal parameter that associates this formal parameter with its corresponding actual parameter. Thus, the relation between the actual parameters and the formal parameters is the same as after an assignment statement.

```
>>> def func(a, b):
...     b = 10*b + a
...     a = 2*b
...     return a + b
...
>>> x = 2
>>> y = 10
# Has the same effect as first executing
# the assignments a = x and b = y, followed
# by executing the function body for these
# values of a and b.
>>> func(x, y)
306
# Has the same effect as first executing
# the assignments a = y and b = -42+4, followed
# by executing the function body for these
```



```
# values of a and b.
>>> func(x, y)
>>> func(y, -42+4)
-1110
# Has the same effect as first executing
# the assignments a = -2*x and b = -12*y, followed
# by executing the function body for these
# values of a and b.
>>> func(-2*x, -12*y)
```

Most modern programming languages, like Python, C, or C++, use call by value as the default parameter passing strategy. The reason for this lies in the simplicity of the call-by-value-semantics: since it behaves in the same way as an assignment statement, we do not need to get used to new behavior for functions. We can just reuse what we learned for assignments.

**Attention:** The relation between the actual and the formal parameters is really the same as after an assignment statement. In particular, if our programming language uses *reference semantics for assignment statements* (see Section 2.4.1), and if we pass a mutable data object to a function, we can see changes to this data object outside the function. This is the case in Python or in Java. For example:

```
# The function f has the formal parameter a.
# a is a list, and f appends the element 100 to a.
# Since lists in Python are mutable, this changes
# the existing list that a refers to.
>>> def f(a):
...     a.append(100)
...
>>> b = [1]
# This call to f has the actual parameter b.
# The function call has the same effect as first executing
# the assignment a = b, followed by executing
# the function body for this value of a.
# In particular, a and b refer to the same list, and the
# change to a in f is visible from b.
>>> f(b)
>>> b
[1, 100]
```

### 4.2.2 Call by reference

In *call by reference*, only variable names from the calling scope are allowed as actual parameters. Then, the formal parameter variable and the actual parameter variable become *synonymous*. All changes to the formal parameter variable inside the function body are visible afterwards from the actual parameter variable. In particular, if the function changes a formal parameter variable through assignment statements, these

changes also affect the actual parameter variable after the function call. Python does not support call by reference, but here is an example from C++:

```
#include <stdio.h>

// The function func has one formal parameter x.
// The modifier & indicates that x uses call by reference.
// Assignments to x in the function body also affect the
// formal parameters.
void func(int &x) {
    printf("This is func!\n");
    x = 5;
}

// The function func2 has one formal parameter x.
// There is no modifier, so x uses call by reference.
// Assignments to x in the function body do not affect
// the formal parameters.
void func2(int x) {
    printf("This is func2!\n");
    x = 5;
}

int main() {
    int y = 10;
    printf("y = %d.\n", y);
    func(y);
    printf("y = %d.\n", y);
    y = 10;
    printf("y = %d.\n", y);
    func2(y);
    printf("y = %d.\n", y);
}
```

This program creates the following output.

```
y = 10.
This is func!
y = 5.
y = 10.
This is func2!
y = 10.
```

An advantage of call by reference is that we can write subroutines that can interact in more subtler ways with their calling environments than by just producing a return value. For example, using call by reference in C++, we can implement a swap function that switches its two parameters like this:

```
#include <stdio.h>

// The function swap has two formal parameters a and b.
// The modifier & indicate that a and b use call by reference.
// The function uses assignments to switch the values of a and b.
// Since we use call by reference, this also affects the
// actual parameters.
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 1;
    int y = 2;
    printf("Before swap: x = %d, y = %d.\n", x, y);
    swap(x, y);
    printf("After swap: x = %d, y = %d.\n", x, y);
}
```

This program creates the following output.

```
Before swap: x = 1, y = 2.
After swap: x = 2, y = 1.
```

In Python, which uses call by value, this would not work:

```
>>> def swap(a, b):
...     temp = a
...     a = b
...     b = temp
...
>>> x = 1
>>> y = 2
>>> x, y
(1, 2)
# The assignments inside swap do not affect
# the variables x and y.
>>> swap(x, y)
>>> x, y
(1, 2)
```

In addition to C++, call by reference is also supported by, e.g., Visual Basic or Pascal.

### 4.2.3 Call by name

In *call by name*, we do not evaluate the actual parameters before executing the function body. Instead, we can imagine that before the function is called, the programming language performs a *search-and-replace* procedure on the function body that replaces

every occurrence of a formal parameter by the corresponding actual parameter, at a textual level. This is particularly relevant if an actual parameter is an expression. Then, this expression is not evaluated when the function is called. Instead, the expression is evaluated every time when the corresponding formal parameter is used. This may be never, once, or multiple times. These evaluations may possibly have different results.

Python does not support call by name, but here is an example from Scala.

```
// The function g outputs a message on the screen and
// returns the value 10.
scala> def g(): Int =
|   println("Hello, this is g.")
|   return 10
|
def g(): Int

// The function f has two formal parameters, a and b.
// The modifier => indicates that a and b use call by name.
// Every time, when a or b is encountered in the function
// body, the expression for the corresponding actual parameter
// is evaluated from scratch.
scala> def func(a: => Int, b: => Int): Int =
|   var c: Int = a + b
|   c = c + 1
|   var d: Int = a
|   return c + d
|
def func(a: => Int, b: => Int): Int

// This call to func has the actual parameters
// g() and g() + 10. These actual parameters
// are expressions that involve calls to the
// function g. Every time, a appears in the function
// body, Scala evaluates the expression g().
// Every time, b appears in the function body,
// Scala evaluates the expression g() + 10.
// In both cases, g() is called.
scala> func(g(), g() + 10)
Hello, this is g.
Hello, this is g.
Hello, this is g.
val res1: Int = 41
```

Here is a second example that shows how different evaluations of the actual parameter can have different results.

```
// a is a global variable, initially set to 0.
scala> var a: Int = 0
var a: Int = 0

// g is a function that increases the global variable a by 1,
```

```
// and then returns the value of a.
scala> def g(): Int =
  |   a = a + 1
  |   return a
  |
def g(): Int

// func has one formal parameter x. The modifier => indicates
// that x should be passed with call by name.
scala> def func(x: => Int): Int =
  |   var y: Int = x + x
  |   var z: Int = x * x
  |   return y + z
  |
def func(x: => Int): Int

// A function call to func with actual parameter g().
// Every time the formal parameter x is used in the function
// body, the expression g() is evaluated from scratch.
// Thus, the first time x is used, it has value 1;
// the second time, it has value 2; the third time, it has
// value 3; and the fourth time it has value 4.
// Consequently, for this call to func, we get y = 1 + 2 = 3, z = 3*5 = 12,
// and a result value of y + z = 3 + 12 = 15.
scala> func(g())
val res3: Int = 15
```

In Python, which uses call by value, the result would be completely different.

```
# a is a global variable, initially set to 0.
>>> a = 0
# The function g increases the global variable by 1, and then
# it returns the value of a.
>>> def g():
...     global a
...     a = a + 1
...     return a
...
# func has one formal parameter x.
>>> def func(x):
...     y = x + x
...     z = x * x
...     return y + z
...
# A function call to func with actual parameter g().
# Since Python uses call by value, this has the same effect
# as first performing the assignment statement x = g(), followed
# by executing the function body for this value of x.
# In particular, g() is called only once, and x has the value
# 1 throughout func.
# Thus, in func, we get y = 1 + 1 = 2, z = 1 * 1 = 1,
```



```
# and a result value of y + z = 2 + 1 = 3.
>>> func(g())
3
```



If we use call by name, it can happen that some actual parameters are never evaluated. This can be useful sometimes, because we can use expressions as actual parameters that are sometimes undefined. Here is an example in Scala:

```
// The function func has two formal parameters a and b.
// The formal parameter a is passed with call by value,
// the formal parameter b is passed with call by name.
scala> def func(a: Int, b: => Int): Int =
|     if a > 0 then
|         return b
|     else
|         return -1
|
def func(a: Int, b: => Int): Int

// A call to func with actual parameters 1 and 10/2.
// This has the same effect as replacing every occurrence of b in the
// function body by the expression 10/2, followed by
// executing the assignment statement a = 1,
// followed by executing the function body of func for this value of a.
// In particular, since a is positive, the first branch of the
// if-statement is executed,. Hence, the expression 10/2 is evaluated
// and the result 5 is returned.
scala> func(1, 10/2)
val res4: Int = 5

// A call to func with actual parameters 0 and 10/0.
// This has the same effect as replacing every occurrence of b in the
// function body by the expression 10/0, followed by
// executing the assignment statement a = 0,
// followed by executing the function body of func for this value of a.
// In particular, since a is zero, the else-branch of the
// if-statement is executed. Hence, the undefined expression 10/0 is
// never evaluated and does not cause any trouble.
// The result is -1.
scala> func(0, 10/0)
val res5: Int = -1
```



Again, this does not work in Python.

```
# The function func has two formal parameters a and b.
>>> def func(a, b):
...     if a > 0:
...         return b
...     else:
...         return -1
```



```
...
# A call to func with the actual parameters 1 and 10/2.
# This has the same effect as first executing the assignment
# statements a = 1 and b = 10/2, followed by executing the
# function body for these values of a and b.
>>> func(1, 10/2)
5.0
# A call to func with the actual parameters 0 and 10/0.
# This has the same effect as first executing the assignment
# statements a = 0 and b = 10/0, followed by executing the
# function body for these values of a and b.
# In particular, since 10/0 is an undefined expression,
# we get an error when Python tries to evaluate the expression
# 10/0 in order to assign it to b.
>>> func(0, 10/0)
ZeroDivisionError: division by zero
```

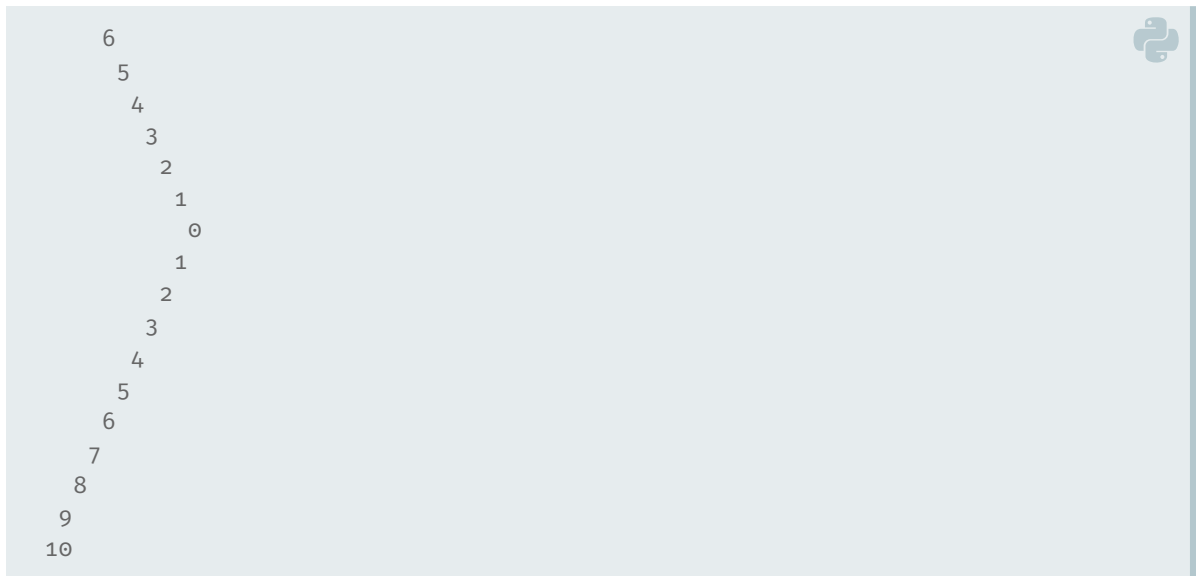
A variant of call by name that is fine-tuned to take advantage of this particular feature is called *call by need*. In call by need, the expression for an actual parameter is not evaluated when the function is called. Instead, the expression is evaluated only when the corresponding formal parameter is encountered for the first time (in particular, if the formal parameter is never used, then the expression is never evaluated). Unlike call by name, however, the result of this evaluation is now stored. When the formal parameter is encountered a second time, we use this stored value and do not evaluate the expression again.

In addition to Scala, the preprocessor macros of C and C++ use call by name. Call by need is a popular choice for *functional* programming languages. For example, in Haskell, call by need is also called *lazy evaluation*.

## 4.3 Recursion

A function may call also itself. This is called *recursion*. When this happens, each invocation of the function has its own local scope and its own variables. For example:

```
>>> def f(x, y):
...     if x <= 0:
...         print(y + str(x))
...     else:
...         print(y + str(x))
...         f(x - 1, y + " ")
...         print(y + str(x))
...
>>> f(10, "")
10
 9
 8
 7
```



In this example, we define a recursive function  $f$  with two parameters  $x$  and  $y$ . Here,  $x$  is an integer, and  $y$  is a string. The function  $f$  first checks whether the parameter  $x$  is nonpositive. If so,  $f$  prints the string  $y$ , followed by the number  $x$ . Then, it terminates immediately. This is called the *base of the recursion*, because for  $x \leq 0$ , the function  $f$  terminates and does not call itself.

Otherwise, if  $x > 0$ , the function  $f$  first outputs the string  $y$  followed by the number  $x$ . Then,  $f$  *calls itself*, but with different parameters. This is called the *recursive call* in  $f$ . As mentioned, this recursive call creates a new instance of  $f$ , with its own actual parameters and its own local variables. In the recursive call, we pass  $x - 1$  as the first actual parameter. This ensures that the value of  $x$  decreases with each recursive call. Thus, we eventually reach the base of the recursion, and  $f$  no longer calls itself. After the recursive call to  $f$  terminates, we again output the string  $y$  followed by the number  $x$ . Note that this outputs the local variables for the current invocation of  $f$ . The example output for  $f(10, "")$  shows the nested structure of the recursive calls and how each recursive call has its own values for  $x$  and  $y$ .

Recursion is a powerful programming technique that is applied in many situations in computer science. We will now see a few examples. We begin with the classic *factorial* function that is familiar from *Diskrete Strukturen für Informatiker*: given a natural number  $n \in \mathbb{N}$ , the factorial  $n!$  is defined as

$$n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot \dots \cdot n.$$

In particular, we have  $0! = 1$  and  $1! = 1$ . The following Python function computes  $n!$  using a **for**-loop.

```
>>> def factorial(n):  
...     val = 1  
...     for i in range(1, n + 1):
```

```
...     val = val * i
...     return val
...
>>> factorial(10)
3628800
>>> factorial(0)
1
```



We say that `factorial(·)` uses an *iterative* approach for compute  $n!$ , because it is based on loops. In contrast, we will now see how to compute the factorial function with a *recursive* approach. For this, we observe that the factorial function has a simple recursive structure: the factorial of 0 is  $0! = 1$ . Next, suppose that we are given  $n \in \mathbb{N}$ ,  $n \geq 1$ , and that we already know how to compute the factorial function  $(n-1)!$ . Then, we can find  $n!$  by multiplying  $(n-1)!$  with  $n$ . More formally, we have that

$$n! = \begin{cases} 1, & \text{if } n = 0, \text{ and} \\ n \cdot (n-1)!, & \text{if } n > 0. \end{cases}$$

The following Python function implements this idea:

```
>>> def factorial_rec(n):
...     if n < 0:
...         return 0      # base of the recursion
...     elif n == 0:
...         return 1      # base of the recursion
...     else:
...         return n * factorial_rec(n - 1) # recursive call
...
>>> factorial_rec(10)
3628800
>>> factorial_rec(0)
1
```



In a recursive function, it is very important that there is at least *base case* for the recursion, i.e., at least one case where the function does *not* call itself. Furthermore, the *recursive calls* from any other case must eventually end up in a base case. If this is not the case, the recursion will not terminate. The cases that lead to a recursive call are also called the *recursion steps* of the function.

Here are two more examples of recursive functions:

```
# Given a list a of integers, compute the sum of
# the elements in a.
# The sum of the elements in the empty list is 0.
# If the list is not empty, we recursively compute
# the sum of the elements in the list that we get
# by removing the first element, and then we add the
# first element to the result.
>>> def sumList(a):
```





```
...     if len(a) == 0:
...         return 0                                # base of the recursion
...     else:
...         return a[0] + sumList(a[1:]) # recursive step
...
>>> sumList([])
0
>>> sumList([1, 2, 1])
4
# Given a list a of integers, compute the maximum of
# the elements in a.
# The maximum of a list with a single element is this element.
# If the list has at least two elements, we remove the first
# element and recursively compute the maximum for this list.
# Then, we compare this maximum to the first element, and we
# report the larger of the two.
>>> def maxList(a):
...     if len(a) == 1:
...         return a[0]                                # base of the recursion
...     else:
...         m = maxList(a[1:])                          # recursive step
...         if a[0] > m:
...             return a[0]
...         else:
...             return m
...
>>> maxList([1])
1
>>> maxList([1, 5, 7, 2, 1])
7
```

## 4.4 The five steps for implementing a function

Functions are used to handle well-defined, recurring tasks in a program, like, e.g., computing the largest number in a list of numbers. The goal is to make programs more readable and more modular. For this, it is important to be precise about the exact behavior of a function and to provide comments with further details about the function.

Hence, when implementing a function, it is good practice to proceed according to the following five steps.

### 4.4.1 Step 1: Give a function signature

The *signature* of a function provides the name of the function, the types of the input parameters, and the type of the return value.

Since Python is dynamically typed, the types of the parameters and of the return value are not fixed, but they can change for each function call. This is demonstrated in the following example.

```
>>> def double(x):  
...     return 2 * x  
...  
>>> double(4)  
8  
>>> double([0, 1])  
[0, 1, 0, 1]
```

However, this is bad programming style. In fact, languages with a static type system (like Scala, Java, or C++) do not allow this. Therefore, we should add comments in Python to state the signature of a function. The signature should be provided before the function is implemented. Here are some examples:

```
# double(x: int): int  
  
# maximum(a: List[int]): int
```

The function `double(·)` is intended to double an integer value. The result is again an integer. The function `maximum(·)` is intended to receive a list of integers. The result will be an integer.


### 4.4.2 Step 2: Provide a specification

The *specification* of a function is a *contract* between the developer of a function and the user of the function. A specification consists of a *precondition* and a *postcondition*. The developer of the function promises the user of the function that *if* the precondition is fulfilled *before* the function is called, then the postcondition will hold *after* the function has finished. The postcondition can be split into a condition on the *effect* of the function (i.e., the way in which the function changes the global state), and a condition on the *result*. The postcondition can be written as a mathematical statement, or as a sentence in *stative passive*<sup>1</sup>. The specification should be fixed *before* the function is implemented. Here are some examples of function signatures and specifications:

```
# double(x: int): int  
# Precondition: None  
# Postcondition:  
#     There is no effect.  
#     result = 2*x  
  
# maximum(a: List[int]): int  
# Precondition: a != []
```

---

<sup>1</sup>The German word is *Zustandspassiv*.



```
# Postcondition:
#   Effect: The largest number in a is printed on the screen.
#   result = max_{i = 0, ..., len(a) - 1} a[i]
def maximum(a):
    res = a[0]
    i = 1
    while i < len(a):
        if a[i] > res:
            res = a[i]
        i = i + 1
    print(res)
    return res


# bar(a: float, b: float): float
# Precondition: b != 0
# Postcondition:
#   Effect: The input numbers a and b are printed on the screen
#   result = a/b
# Result: The quotient of a and b is returned.
def bar(a, b):
    print(a, b)
    return a / b
```

Note the difference between the function `print(·)` and the `return`-statement, and how this affects the specification. The `print(·)`-function changes the state of the machine (output device). Thus, it belongs to the effect of the function. The value of the `return`-statement provides the result of the function.

**Attention:** Whenever possible, a function should have either an effect or a result, but not both. The reason is as follows:

Functions sometimes have hidden effects in addition to returning a value. These effects are called the *side-effects* of the function. Sometimes, unintended side effects may lead to errors that are hard to find in an imperative program. The following implementation of the `minimum(·)`-function has the side effect that it changes the input list:






```
# minimum(a: List[int]): int
# Precondition: a != []
# Postcondition:
#     Effect: for i = 0,..., len(a) - 1, the position a[i] contains
#             the minimum of a[0],..., a[i]
#     result = min_{i = 0, ..., len(a)-1} a[i]
def minimum(a):
    res = a[0]
    i = 1
    while i < len(a):
        if a[i] < res:
            res = a[i]
        a[i] = res
        i = i + 1
    return m
```

### 4.4.3 Step 3: Design interesting test cases

A *test case* for a function consists of an input to the function, together with the expected output. We would like to have test cases that cover the “expected” behavior of the function for typical inputs, as well as tricky “corner cases”. If there are different kinds of inputs for which the function might behave differently, there should be at least one test case for each such kind of input.

We emphasize that even if a function passes all test cases, we usually cannot be certain that the function works correctly on *all* inputs. This certainty can only be obtained with a formal *proof of correctness* (see Chapter 9). However, formal correctness proofs can be very complex, and good test cases are very useful in catching errors. Good test cases can also help in making sure that a change in the function code did not break anything. The test cases should be designed once the specification of the function is completed, but *before* the function is implemented. Here are some examples:



```
# double(x: int): int
# Precondition: None
# Postcondition:
#     There is no effect.
#     result = 2*x
''' Test cases:
double(4) == 8    # positive number
double(0) == 0    # number 0
double(-1) == -2  # negative number
'''

# maximum(a: List[int]): int
# Precondition: a != []
# Postcondition:
#     Effect: The largest number in a is printed on the screen.
```



```
# result = max_{i = 0, ..., len(a) - 1} a[i]
''' Test cases:
maximum([1,2,3,2,1]) == 3
maximum([1,2,3,4]) == 4
maximum([4,3,2,1]) == 4
maximum([1,1,1,1]) == 1
maximum([42]) == 42
'''
```



Observe that `maximum([])` is not a test case, even though it seems to be an important edge case for the implementation. However, the precondition explicitly forbids the case that `a` is empty. Thus, `maximum(·)` does not need to fulfill any guarantee for an empty list, and every result (even an error) will satisfy the specification.

#### 4.4.4 Step 4: Implement the function

Once the signature, the specification, and the test cases have been formulated, we can proceed with the actual implementation. Once we have finished the function definition, we use our test cases to find potential errors in the implementation.

#### 4.4.5 Step 5: Add helpful comments

Finally, we add *comments* in natural language to our code. Comments are supposed to explain the behavior of certain parts in the code and to illustrate the underlying ideas. Comments are important when multiple people work on the same project—they make it easier to understand what other people did (in fact, comments can also help the original author to understand code that has been written some time ago).

Here are two full examples of how to build a function:

```
# sumList(a: List[int]): int
# Precondition: None
# Postcondition:
#   Effect: None
#   Result: result = a[0] + a[1] + ... + a[len(a) - 1]
#   in particular, result = 0, if a = []
''' Test cases:
sumList([3,3,3,3]) == 12
sumList([5]) == 5
sumList([]) == 0
sumList(range(1,101)) == 100*101//2
sumList([9,-9]) == 0
sumList([1,5,2,-4,9]) == 13
'''

def sumList(a):
    # res accumulates the result, it is initially 0
    rep = 0
```






```
# We use a for-loop that iterates through all elements in a
# and accumulates them in x.
for x in list:
    res += x

return res

# maximum(a: List[int]): int
# Precondition: a != []
# Postcondition:
#     Effect: The largest number in a is printed on the screen.
#     result = max_{i = 0, ..., len(a) - 1} a[i]
''' Test cases:
maximum([1,2,3,2,1]) == 3
maximum([1,2,3,4]) == 4
maximum([4,3,2,1]) == 4
maximum([1,1,1,1]) == 1
maximum([42]) == 42
'''
def maximum(a):
    # res contains the maximum
    # of the elements that we have
    # seen so far. Initially, it is a[0].
    res = a[0]
    # we iterate over the remaining elements
    # a[1], ..., a[len(a) - 1]
    i = 1
    while i < len(a):
        # if a[i] is larger than the maximum
        # so far, we update the maximum.
        if a[i] > res:
            res = a[i]
        i = i + 1
    print(res)
    return res
```

### Case Study: Autocomplete



```
def tokenize(text):
    tokens = []
    current_token = []
    allowed_chars = set("abcdefghijklmnopqrstuvwxyz0123456789'")
    for char in text:
        if char in allowed_chars:
            current_token.append(char)
        else:
            if current_token:
                tokens.append(''.join(current_token))
                current_token = []
    if current_token:
        tokens.append(''.join(current_token))
    return tokens

def read_text_file(file_path):
    file = open(file_path, 'r')
    text = file.read().lower()
    words = tokenize(text)
    return words

def build_markov_model(words):
    markov_model = {}
    for i in range(len(words) - 1):
        current_word = words[i]
        next_word = words[i + 1]
        if current_word not in markov_model:
            markov_model[current_word] = {}
        if next_word not in markov_model[current_word]:
            markov_model[current_word][next_word] = 0
        markov_model[current_word][next_word] += 1
    return markov_model

def predict_next_words(markov_model, current_word, num_suggestions):
    if current_word not in markov_model:
        return []
    next_word_dict = markov_model[current_word]
    next_word_list = list(next_word_dict.items())
```



```
def get_frequency(item):
    return item[1]
sorted_next_words = sorted(next_word_list,
                           key=get_frequency, reverse=True)
selected_next_words = sorted_next_words[:num_suggestions]
suggestions = []
for item in selected_next_words:
    suggestions.append(item[0])
return suggestions

def autocomplete(markov_model, num_suggestions):
    user_input = input("Enter a word or "
                       "'quit' to exit: ").strip().lower()
    while user_input != 'quit':
        input_words = tokenize(user_input)
        if not input_words:
            print("Please enter a valid input.")
        else:
            last_word = input_words[-1]
            suggestions = predict_next_words(markov_model,
                                           last_word,
                                           num_suggestions)

            if suggestions:
                print("Suggestions for '" + last_word + "':")
                i = 1
                for suggestion in suggestions:
                    print(str(i) + ". " + suggestion)
                    i = i + 1
            else:
                print("No predictions available for "
                      "the given input.")
        user_input = input("Enter a word or "
                           "'quit' to exit: ").strip().lower()

file_path = 'The_Tale_of_Genji.txt' # Path to your text file

print("This is an autocomplete program based on a "
      "Markov Model. The text is " + file_path + ".")
num_suggestions = int(input("How many recommended "
                             "autocomplete results you want to see? "))

words = read_text_file(file_path)
markov_model = build_markov_model(words)

autocomplete(markov_model, num_suggestions)
```

# Entwurf

## III

### Algorithms

### Algorithmic Problems

#### Lernziele

**KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.

**KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

**KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und -paradigmen.

**KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.



An *algorithmic problem* is a problem that is suitable for processing with computers. The description of an algorithmic problem consists of two parts: a formal description of all the possible *inputs*, and, for each possible input, a formal description of the desired *output*. Here are some examples:

#### Finding the factorial:

Possible inputs: Natural numbers  $n \in \mathbb{N}_0$ .

Desired output for input  $n$ : The factorial  $n!$  of  $n$ .



#### Computing the product:

Possible inputs: Pairs of real numbers  $a, b \in \mathbb{R}$ .

Desired output for input  $a, b$ : The product  $a \cdot b$  of  $a$  and  $b$ .



#### Computing the greatest common divisor:

Possible inputs: Pairs of natural numbers  $a, b \in \mathbb{N}_0$ .

Desired output for input  $a, b$ : The greatest common divisor  $\text{gcd}(a, b)$  of  $a$  and  $b$ . That is, the largest natural number that divides both  $a$  and  $b$  without remainder (we define  $\text{gcd}(0, 0) = 0$ ).





### Single-pair-shortest-path computation:

Input: The public transportation map of Berlin, the name of a start and a target station.

Desired Output for an input: A train ride between the start and the target station.

In Computer Science, our goal is to *solve* algorithmic problems. That is, we would like to develop *algorithms* for algorithmic problems that are *provably correct* (i.e., that provide a desired output for every possible input) and that are *provably efficient* (i.e., that use as little resources as possible). This immediately raises several questions: how do we “prove” that an algorithm is correct and efficient? What does it even mean that an algorithm is “efficient”? Is it possible to find an algorithm for *every* reasonable algorithmic problem? What is an “algorithm”, anyway? In the following chapters, we will focus on the first two questions. The other two questions will be addressed in *Grundlagen der Theoretischen Informatik*, in the third semester.

## 6.1 Finding the greatest common divisor

To start with a simple, but classic, example, we will look at a famous algorithm for computing the *greatest common divisor* (gcd) of two natural numbers  $a, b \in \mathbb{N}_0$ . Our algorithm will be based on a simple recursive strategy that reduces the task of computing the gcd of  $a$  and  $b$  to the task of computing the gcd for two smaller numbers. The main idea is captured in the following lemma:

**Lemma 6.1.** Let  $a, b \in \mathbb{N}_0$  be two natural numbers with  $a \geq b$ , and let  $c \in \mathbb{N}_0$ . Then, we have:

$$c \text{ divides } a \text{ and } c \text{ divides } b \Leftrightarrow c \text{ divides } b \text{ and } c \text{ divides } a - b.$$

*Beweis.* We show both directions of the equivalence. We start with “ $\Rightarrow$ ”: suppose that  $c$  divides both  $a$  and  $b$ . This means that there are natural numbers  $x, y \in \mathbb{N}_0$  such that  $a = c \cdot x$  and  $b = c \cdot y$ . Then, we also have

$$a - b = c \cdot x - c \cdot y = c \cdot (x - y).$$

This means that  $c$  divides  $a - b$ , as claimed.


Now, we show “ $\Leftarrow$ ”: suppose that  $c$  divides both  $b$  and  $a - b$ . This means that there are natural numbers  $x, y \in \mathbb{N}_0$  such that  $b = c \cdot y$  and  $a - b = c \cdot x$ . Then, we also have

$$a = (a - b) + b = c \cdot x + c \cdot y = c \cdot (x + y).$$

This means that  $c$  divides  $a$ , as claimed. □


Lemma 6.1 states that the numbers  $a$  and  $b$  and the numbers  $b$  and  $a - b$  have the same set of common divisors. In particular, this means that the *greatest* common divisor of  $a$  and  $b$  is the same as the greatest common divisor of  $b$  and  $a - b$ . In other

words, we have  $\text{gcd}(a, b) = \text{gcd}(b, a - b)$ . Hence, we can compute the gcd of  $a$  and  $b$  by computing the gcd of  $b$  and  $a - b$ . Now, if  $b > 0$ , then  $a - b$  is smaller than  $a$ , and we can apply recursion to solve the problem. On the other hand, if  $b = 0$ , then, by definition, we have  $\text{gcd}(a, b) = a$ . In this case, we have reached the base of the recursion. We can return the answer directly. This leads to the following Python program:



```
# gcd(a: int, b: int): int
# Precondition: a >= 0, b >= 0
# Postcondition:
#     Effect: None
#     Result: result = gcd(a, b)
''' Test cases
Your task: find some good test cases
'''
def gcd(a, b):
    if a < b:      # gcd is commutative. Make sure that a >= b.
        return gcd(b, a)
    elif b == 0:  # base of the recursion
        return a
    else:        # recursion step; a>=b; apply Lemma 6.1.
        return gcd(b, a - b)
```

If we look closely at how the recursion in `gcd(·)` operates, we see that there is a simple shortcut to speed up the algorithm: when `gcd(·)` is called with natural numbers  $a$  and  $b$  such that  $a \geq b > 0$ , then the recursion will repeatedly subtract  $b$  from  $a$  until it reaches a number that is smaller than  $b$ . This number is exactly  $a \% b$ , because we subtract from  $a$  the largest multiple of  $b$  that leads to a number smaller than  $b$ . Thus, we not only have  $\text{gcd}(a, b) = \text{gcd}(b, a - b)$ , but also  $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$ . We can use this directly in the recursion. The result looks like this:



```
# gcd2(a: int, b: int): int
# Precondition: a >= 0, b >= 0
# Postcondition:
#     Effect: None
#     Result: result = gcd(a, b)
''' Test cases
Your task: find some good test cases
'''
def gcd2(a, b):
    if a < b:      # gcd is commutative. Make sure that a >= b.
        return gcd2(b, a)
    elif b == 0:  # base of the recursion
        return a
    else:        # recursion step; a>=b; apply Lemma 6.1 with shortcut.
        return gcd2(b, a % b)
```

This algorithm is called *Euclid's algorithm*. It appears in Euclid's book *Elements*, which was written more than 2000 years ago.



The algorithm  $\text{gcd2}(\cdot)$  is quite fast. The main reason is captured in the following lemma:

**Lemma 6.2.** *Let  $a, b \in \mathbb{N}_0$  be two natural numbers with  $a \geq b > 1$ . Then, we have:*

$$a \% b < a/2.$$

*Beweis.* We distinguish two cases.

- **Case 1:**  $b \leq a/2$ . In this case, we immediately have

$$a \% b < b \leq a/2,$$

because the result of the modulo operation is a natural number between 0 and  $b - 1$ .

- **Case 2:**  $b > a/2$ . In this case, we have  $a < 2b$ , and hence

$$a \% b = a - b < a - a/2 = a/2.$$

In both cases, the claim holds, and the lemma follows.  $\square$

Using Lemma 6.2, we see that in each recursion step, the *product* of the parameters  $a$  and  $b$  decreases by a factor of at least two. Thus, after a logarithmic number of recursive calls, the algorithm must terminate. More precisely, we have the following theorem:

**Satz 6.3.** *Let  $a, b \in \mathbb{N}_0$  with  $a \geq b$ . Then, the call  $\text{gcd2}(a, b)$  terminates after at most  $\log_2(ab) + 2$  invocations of  $\text{gcd2}(\cdot)$ .*

*Beweis.* We call an invocation of  $\text{gcd2}(\cdot)$  *unsuccessful* if it leads to a recursive call. Then, the total number of invocations of  $\text{gcd2}(\cdot)$  is one plus the number of unsuccessful invocations of  $\text{gcd2}(\cdot)$ .

Thus, our goal is to bound the number of unsuccessful invocations of  $\text{gcd2}(\cdot)$ . Now, the crucial observation is that Lemma 6.2 implies that in each recursive invocation of  $\text{gcd2}(\cdot)$ , the product of the actual parameters decreases by a factor of at least two. More precisely, in the first invocation of  $\text{gcd2}(\cdot)$ , the product of the actual parameters is at most  $ab$ ; in the second invocation of  $\text{gcd2}(\cdot)$ , the product of the actual parameters is at most  $ab/2$ ; in the third invocation of  $\text{gcd2}(\cdot)$ , the product of the actual parameters is at most  $ab/4$ ; and so on. In general, in the  $k$ th invocation of  $\text{gcd2}(\cdot)$ , the product of the actual parameters is at most  $ab/2^{k-1}$ . An invocation of  $\text{gcd2}(\cdot)$  can be unsuccessful only if the product of the actual parameters is at least 1 (since both actual parameters are natural numbers and since both of them need to be at least 1 for a recursive call to occur). Thus, we can bound the maximum number  $k^*$  of unsuccessful invocations of  $\text{gcd2}(\cdot)$  as follows:

$$1 < \frac{ab}{2^{k^*-1}} \Leftrightarrow 2^{k^*-1} < ab \Leftrightarrow k^* < \log_2(ab) + 1.$$

Thus, we have at most  $\log_2(ab) + 1$  unsuccessful invocations of  $\text{gcd2}(\cdot)$ . It follows that the total number of invocations of  $\text{gcd2}(\cdot)$  is at most  $\log_2(ab) + 2$ , as claimed.  $\square$

Theorem 6.3 implies that the number of recursive calls is proportional to the total number of digits in the binary representation of  $a$  and  $b$ . We say that Euclid's algorithm has *running time*  $O(\log ab)$ . The *O-notation* is used to have a succinct yet precise representation of the running time of an algorithm. We will see more examples in the next sections and discuss the *O-notation* in more detail in Section 8.2. A formal description of the *O-notation* and its properties will be given in later semesters.

## 6.2 Searching

*Searching* is one of the most fundamental algorithmic problems in computer science. Typically, the setting is as follows: we have a big data data base, organized in a certain data structure, and we would like to know if a given value (called the *search key*) is contained in this structure. Often, the data base also contains some associated data with the search key that we would like to retrieve (the *satellite data*). For example, the campus management system of Freie Universität Berlin contains a huge list of student entries. Given the id number of a certain student, we would like to find this student in the data base and to determine whether this student is enrolled in *Konzepte der Programmierung*.

For now, we focus on methods for searching in lists. Later in this class (see Chapter 16), and in the follow-up lecture *Algorithmen und Datenstrukturen*, we shall also consider searching in more complex data structures, like trees and hash-tables. For lists, the algorithmic problem of searching can take to different forms:

### Searching in an unsorted list:

Possible inputs: a list  $a$  with elements of some type  $T$ ; a search key  $k$  of the same type  $T$

Desired output for inputs  $a$  and  $k$ : an index  $i$  such that  $a[i] == k$ , if it exists; and **None**, otherwise



### Searching in a sorted list:

Possible inputs: a *sorted* list  $a$  with elements of some type  $T$ , such that  $T$  allows for a total order; a search key  $k$  of the same type  $T$

Desired output for inputs  $a$  and  $k$ : an index  $i$  such that  $a[i] == k$ , if it exists; and **None**, otherwise



Note that if the search key  $k$  occurs multiple times in  $a$ , our algorithmic problems do not specify which occurrence we would like to find. Note also that searching in an unsorted list (the first algorithmic problem) is more general than searching in a sorted list (the second algorithmic problem). In particular, any algorithm that

solves searching in an unsorted list will also solve searching in a sorted list, but not necessarily the other way round.

### 6.2.1 Linear search

There is a very easy solution for searching in an unsorted list  $a$ : we go through the elements of  $a$  one by one, and we check whether we can find the search key  $k$ . If so, we return the corresponding index. If not, we return **None**. This algorithm is called *linear search*, because it performs a linear scan through the list  $a$ . It can be implemented with a simple **for**-loop:

```
# linear_search(a: List[T], k: T): int      T is an arbitrary type
# Precondition: None
# Postcondition:
#     Effect: None
#     a[result] == k, if k appears in a
#     result == None, otherwise
''' Test cases
Your task: find some good test cases
'''
def linear_search(a, k):
    for i in range(len(a)): # We check every position in a,
        if a[i] == k:      # and stop if we find k.
            return i
    return None             # k is not in a.
```

In the worst case, linear search has to check every element of  $a$ . That is, if  $a$  contains  $n$  elements, then, in the worst case, linear search will need to perform  $n$  comparisons. This is unavoidable. If we know nothing about the structure of  $a$ , we can obtain only a single piece of information with each comparison. Namely, with one comparison  $a[i] == k$ , we can only determine whether a given position  $i$  of  $a$  contains the search key  $k$ , or not. Thus, unless we check every position of  $a$ , we cannot be certain that  $k$  is not present in  $a$ .

We say that linear search has *running time*  $O(n)$ . This is also called a *linear running time*, because the number of steps grows linearly with the input size. Here, we use again the  $O$ -notation that we saw in Section 6.1.

### 6.2.2 Binary search

We turn to searching in a *sorted* list  $a$ . As mentioned above, linear search is still applicable, leading to a solution with running time  $O(n)$ . However, now there is a better way to proceed. The idea is as follows: when we argued that linear search is optimal, we said that with a single comparison  $a[i] == k$ , we can only obtain a single piece of information. However, if we have the *additional* information that  $a$  is sorted, a single comparison can tell us much more. Namely, if  $a[i] < k$ , we can exclude not

only position  $i$  from consideration, but also *all* positions that come *before*  $i$ . Similarly, if  $a[i] > k$ , we can exclude not only  $i$ , but also *all* positions that come *after*  $i$ .

This idea leads to the following strategy: Let  $m$  be the *middle* element of  $a$ . Compare the search key  $k$  with  $m$ . There are three cases:

1. If  $k == m$ , we have found the search key.
2. If  $k < m$ , then every element in the *right half* of  $a$  is *larger* than  $k$ . Hence, we can continue our search in the *left half* of the list.
3. If  $k > m$ , then every element in the *left half* of  $a$  is *smaller* than  $k$ . Hence, we can continue our search in the *left half* of the list.

This algorithm is called *binary search*, since in each step, there are *two* possible sublists where the search can continue. A recursive implementation of binary search is as follows:

```
# binary_search(a: List[U], k: U): int      U is a totally ordered universe
# Precondition: a is sorted in ascending order
# Postcondition:
#   Effect: None
#   a[result] == k, if k appears in a
#   result == None, otherwise
''' Test cases
Your task: find some good test cases
'''
def binary_search(a, k):

    # This function searches k in the sublist a[left:right]
    def bin_search(left, right):
        if left >= right:                # k is not in a.
            return None
        m_pos = left + (right - left) // 2 # index of middle element
        m = a[m_pos]                      # middle element
        if k == m:                        # k has been found
            return m_pos
        elif k < m:                       # continue in left half
            return bin_search(left, m_pos)
        else: # m < k                     # continue in right half
            return bin_search(m_pos + 1, right)

    # invocation of the helper function
    return bin_search(0, len(a))
```

**Attention:** Note that we compute index  $m\_pos$  of the middle element as

```
m_pos = left + (right - left) // 2
```

and not as

```
m_pos = (left + right) // 2
```



The reason for this is that in this way, we avoid a potential *overflow* that might occur if  $\text{left} + \text{right}$  is bigger than the largest number that can be represented by an integer (see Section 3.1.3 for a discussion of overflows). In the first expression, all intermediate results will be smaller than  $\text{right}$ , and an overflow cannot occur. In Python, this precaution is actually not necessary, because integers may be arbitrarily large. In other languages, however, this may be an issue. Thus, it is recommended to use the former expression.

We now analyze the performance of binary search.

**Satz 6.4.** *Suppose we apply binary search to a list  $a$  of  $n$  elements. Then, there are at most  $\log_2(n) + 2$  invocations of  $\text{binSearch}(\cdot)$ .*

*Beweis.* We call an invocation of  $\text{bin\_search}(\cdot)$  *unsuccessful* if it leads to a recursive call. Thus, the total number of invocations of  $\text{bin\_search}(\cdot)$  is one plus the number of unsuccessful invocations of  $\text{bin\_search}(\cdot)$ .

Our goal is to bound the number of unsuccessful invocations of  $\text{bin\_search}(\cdot)$ . The crucial observation is that in each recursive call, we discard at least one half of the remaining sublist (either the left half or the right half, plus the middle element). That is, in the first invocation of  $\text{bin\_search}(\cdot)$ , we have at most  $n$  remaining elements; in the second invocation of  $\text{bin\_search}(\cdot)$ , we have at most  $n/2$  remaining elements; and so on. In general, in the  $k$ th invocation of  $\text{bin\_search}(\cdot)$ , we have at most  $n/2^{k-1}$  remaining elements. An invocation of  $\text{bin\_search}(\cdot)$  can be unsuccessful only if there is at least one remaining element. Thus, the maximum number  $k^*$  of unsuccessful invocations satisfies

$$\frac{n}{2^{k^*-1}} \geq 1 \Leftrightarrow n \geq 2^{k^*-1} \Leftrightarrow k^* \leq \log_2(n) + 1.$$

Thus, we have at most  $\log_2(n) + 1$  unsuccessful invocations of  $\text{bin\_search}(\cdot)$ . It follows that the total number of invocations of  $\text{bin\_search}(\cdot)$  is at most  $\log_2(n) + 2$ , as claimed.  $\square$

We say that binary search has running time  $O(\log n)$ . This is also called a *logarithmic running time*, because the number of steps grows logarithmically with the input size. This is much better than linear time, since the linear function  $n \mapsto n$  grows much faster than the logarithmic function  $n \mapsto \log n$ .

**Attention:** ADVANCED COMMENT: There is an important difference between Euclid's algorithm and binary search. In Section 6.1, we saw that Euclid's algorithm has running time  $O(\log ab)$ , for natural numbers  $a, b \geq 0$ . Just now, we showed that binary search has running time  $O(\log n)$ , for a list of  $n$  elements.



However, the running time of Euclid's algorithm is *not* logarithmic, but *linear*! The reason is that the *input size* for Euclid's algorithm is actually  $\log_2 a + \log_2 b = \log_2 ab$ , and not  $a + b$ . The input is given in binary representation. Thus, the running time of Euclid's algorithm is proportional to the input size and hence linear.

On the other hand, the input size for binary search is  $n$ , the length of the list. Thus, the running time of binary search is proportional to the *logarithm* of the input size.

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP7** Du entwickelst Algorithmen zur Lösung vorgegebener *algorithmischer Listen- und Baumprobleme* und stellst diese mithilfe unterschiedlicher Programmierkonzepte in *Scala und Python* dar.
- KdP8** Du analysierst *Algorithmen*, indem du *Korrektheit* (auf Grundlage der Spezifikation), *Speicherplatz* und *Laufzeit* angibst und begründest.



*Sorting* is a fundamental algorithmic problem that occurs in countless situations in Computer Science. For example, in Section 6.2, we saw that the search problem can be solved much faster if the input list is sorted. Thus, if we have a list of items for which we will need to perform many searches, it makes sense to sort this list in a preprocessing step. There are many more examples of this kind. We will take a detailed look at the sorting problem in this chapter.

### 7.1 The algorithmic problem of sorting

Intuitively, the sorting problem is as follows: we are given a list of comparable elements (i.e., for each pair of elements, we can say which one is larger and which one is smaller). The goal is to rearrange these elements such that they are in increasing (or decreasing) order. Formally, the algorithmic problem can be stated like this:

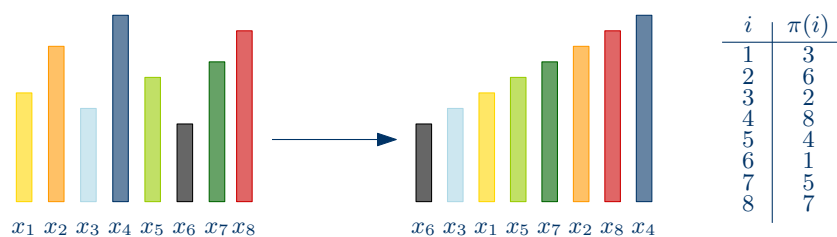
### Sorting:

Possible inputs: sequences  $X = [x_1, \dots, x_n]$  with elements from a totally ordered universe  $U$ .

Desired output for an input  $X$ : a rearranged sequence  $X' = [x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)}]$ , where  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  is a permutation<sup>a</sup> such that  $x_{\pi(1)} \leq x_{\pi(2)} \leq \dots \leq x_{\pi(n)}$ .

<sup>a</sup>A *permutation* is a *bijective* function from  $\{1, \dots, n\}$  to  $\{1, \dots, n\}$ , i.e., a function in which each number from  $\{1, \dots, n\}$  occurs *exactly once* as an image.

By an *ordered universe*, we mean a type that allows for comparisons between its elements (e.g., `int` or `str`, but not `complex`). The permutation  $\pi$  assigns to each input element  $x_i$  the position of  $x_i$  in the sorted order. For example, in the next figure, we have  $\pi(1) = 3$ , because the element  $x_1$  is the third element in the sorted order of the input sequence:



In this class, we will discuss different sorting algorithms—most of them in this chapter. It is important to notice that no single sorting algorithm is the “best” one. Each algorithm has its own advantages and disadvantages, and depending on the situation, one algorithm may be preferable over the other. In this chapter and in the following one, we will discuss some of the aspects that play into choosing a sorting algorithm. But first, we will take a closer look at the following four sorting algorithms:

- (i) selection sort: build a sorted list by successively picking the smallest element from the remaining list.
- (ii) insertion sort: build a sorted list by successively inserting the next element into the right position in the partial list.
- (iii) merge sort: use a divide and conquer strategy, where most of the work happens in the “conquer” step.
- (iv) quicksort: use a divide and conquer strategy, where most of the work happens in the “divide” step.

When implementing the sorting algorithms in Python, we can choose whether we want to have a *procedure* that receives an input list `a` and rearranges the elements of `a` into sorted order (using the fact that the call-by-value semantics of Python relies on the reference semantics for assignment statements, see Section 4.2.1). Then, the sorting function has the *effect* of changing the parameter list.



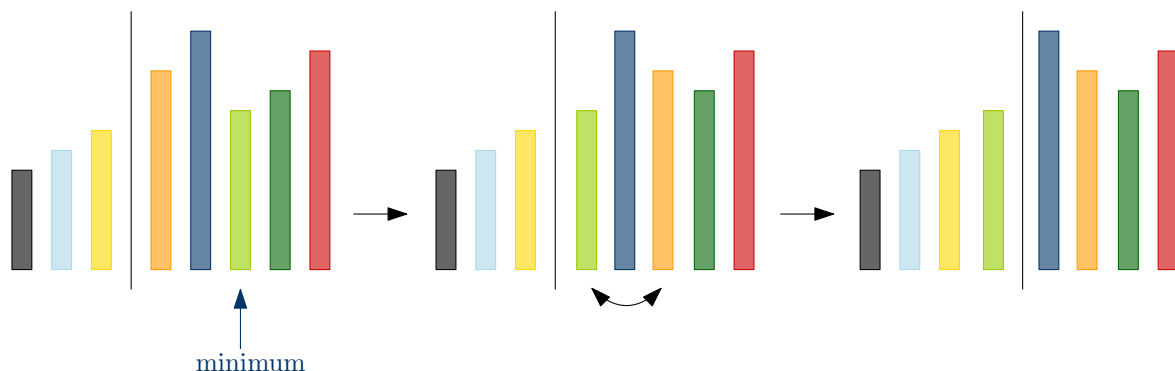
Alternatively, we can have a *function* that returns a *new* list that contains the elements of the input list *a* in sorted order. In this case, the function has a *return value* that is the sorted list.

The second possibility is useful if we do not want to destroy the original unsorted sequence. However, we will follow the first possibility, because it is more space efficient (and more common).

## 7.2 Selection sort

*Selection sort* is a very simple and intuitive sorting algorithm. The idea is as follows: Take all the elements that want to sort in your hand. Then, as long as there are still elements on your hand, find the smallest element on your hand and put it onto a stack on the table. Repeat. Once this procedure is finished, the stack on the table contains all the elements in sorted order.

More technically, suppose we have a list *a* of elements. We subdivide *a* into a sorted part (front) and a remaining unsorted part (back). Then, we repeat the following step, until the elements of *a* are in sorted order: find the smallest element in the unsorted part and *swap* it with the first element in the unsorted part. In the following figure, you can see one step of selection sort:



First, we implement a procedure `((·)swap)` that exchanges two elements in a given list *a*.

```
# swap(a: List[U], i: int, j: int): NoneType; U is an arbitrary type
# Precondition: 0 <= i, j < len(a)
# Postcondition:
#   Effect: a[i] and a[j] switched places in a, the remainder
#           of the list is unchanged
#   Result: None
'''Tests: Your exercise ;-)'''
def swap(a, i, j):
    temp = a[i] # save a[i] in a temporary variable.
    a[i] = a[j] # overwrite a[i] with a[j].
    a[j] = temp # put the stored value of a[i] into a[j]
```



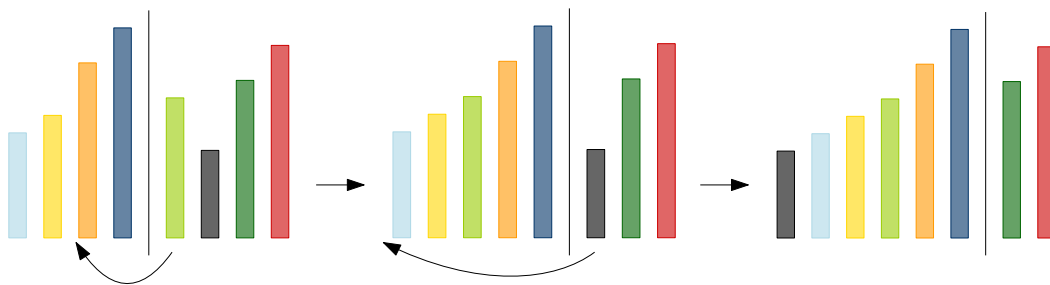
Now we can present the implementation of selection sort.

```
# selection_sort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
'''Tests: Your exercise ;-)'
'''
def selection_sort(a):
    n = len(a)
    for i in range(n):
        # The sorted part is from 0 to i - 1.
        # The unsorted part is from i to n - 1.
        # Find the index of the smallest element.
        # in the unsorted list.
        min_index = i          # candidate for the minimum
        for j in range(i + 1, n):
            if a[j] < a[min_index]:
                min_index = j    # update the candidate
        swap(a, i, min_index)   # put the next minimum into its position.
```

### 7.3 Insertion sort

*Insertion sort* is a simple algorithm that is very similar to selection sort. The idea is as follows: we leave all the elements that we want to sort on the table. We pick up the elements one by one. In our hand, we keep the elements that we have picked up so far in sorted order. Each time that we pick up a new element, we find the correct position in our hand, and we insert the element there.

More technically, we again subdivide the input list *a* in a sorted part (front) and an unsorted part (back). Then, we repeat the following step, until the elements of *a* are in sorted order: we pick the first element in the unsorted part, and we swap it to the left (into the sorted part) until it is in the correct position. In the following figure, you can see two steps of this insertion process:



The implementation of insertion sort is as follows:

```
# insertion_sort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
'''Tests: Your exercise ;-)'
def insertion_sort(a):
    n = len(a)
    for i in range(n):
        # The sorted part is from 0 to i - 1.
        # The unsorted part is from i to n - 1.
        # Insert the element a[i] onto the list
        # a[0], ..., a[i - 1].
        j = i
        # The element is swapped down through the list
        # until it reaches its correct position.
        while j > 0 and a[j - 1] > a[j]:
            swap(a, j, j - 1)
            j = j - 1
```

## 7.4 Merge sort

*Merge sort* is a recursive algorithm that applies the *divide-and-conquer* paradigm. The idea of divide-and-conquer is as follows: given a big problem, we subdivide the big problem into several small subproblems (the number of subproblems can vary; often we have two subproblems). This is called the *divide* step. Then, we use recursion to produce an individual solution for each subproblem. Finally, we take the individual solutions for the subproblems, and we combine them into a solution for the original big problem. This is called the *conquer* step.

For the sorting problem, there are several ways how we can apply the divide-and-conquer paradigm. In merge sort, it is done as follows:

- **Divide:** We partition the input sequence  $a$  into two sequences  $l$  and  $r$  of (almost) equal size. (If the size of  $a$  is odd,  $r$  has one more element than  $l$ .)
- **Recurse:** We use recursion to sort the two sequences  $l$  and  $r$ .
- **Conquer:** We use the two sorted sequences  $l$  and  $r$  to obtain the sorted order for the input list  $a$ .

The main work happens in the conquer step, when we use  $l$  and  $r$  to obtain the sorted list  $a$ . For this, we need a special *merge* procedure that we will explain below. The overall idea is depicted in the following figure:



Let us now explain the merge procedure. We are given two sorted sublists  $l$  and  $r$ , and we would like to combine the two lists into *one* sorted list  $a$ . We can imagine that the elements of  $l$  and  $r$  form two queues, each in sorted order (like, e.g., at an airport or at a nightclub). The elements are waiting to get into the combined list  $a$ . In front of the two queues, there is a guard. In each step, the guard compares the first element of  $l$  and the first element of  $r$ . Then the guard lets the smaller of the two elements proceed to the back of  $a$ . The other element has to wait, and it will be compared again in the next step. The Python-code is as follows:

```
# merge(l: List[U], r: List[U], a: List[U]): NoneType
# U is a totally ordered universe
# Precondition: len(l) + len(r) == len(a); l and r are in sorted order
# Postcondition:
#     Effect: The elements of l and r are in a, in sorted order.
#     Result: None
'''Tests: Your exercise ;-)''''
def merge(l, r, a):
    # i is the index of the first remaining element of l.
    i = 0
    # j is the index of the first remaining element of r.
    j = 0
    # k is the index of first free spot in a.
    k = 0
```



```
# There are remaining elements in both l and r.
while i < len(l) and j < len(r):
    # Compare the two first remaining elements of l and r.
    # Let the smaller element pass to the back of a.
    if l[i] <= r[j]:
        a[k] = l[i]
        i = i + 1
    else:
        a[k] = r[j]
        j = j + 1
    k = k + 1
# Let the remaining elements of l pass, if any.
while i < len(l):
    a[k] = l[i]
    i = i + 1
    k = k + 1
# Let the remaining elements of r pass, if any.
while j < len(r):
    a[k] = r[j]
    j = j + 1
    k = k + 1
```

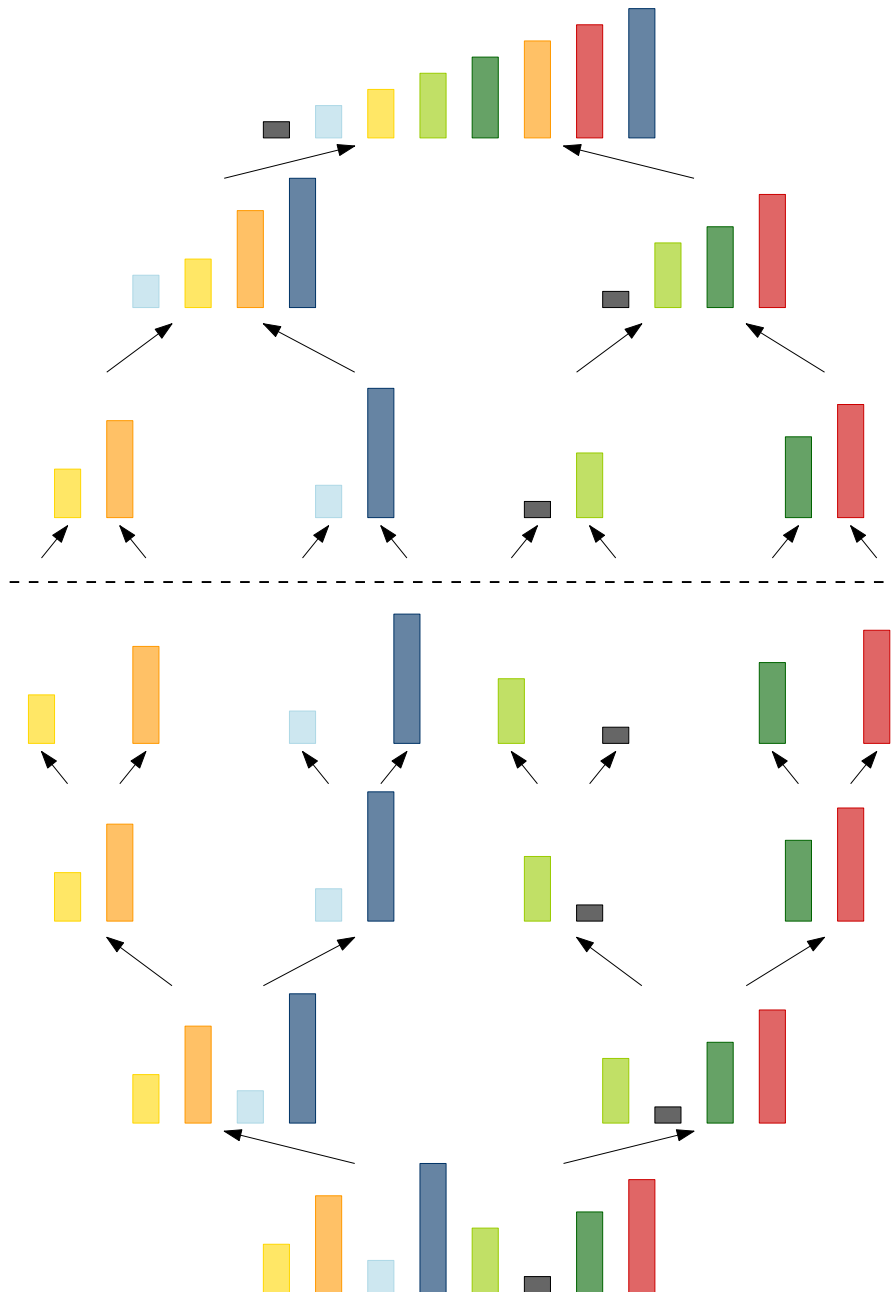


Using `merge(·)`, it is easy to implement the merge sort algorithm as follows:

```
# merge_sort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
'''Tests: Your exercise ;-)'''
def merge_sort(a):
    n = len(a)
    # Recursion base. A list with at most one element is always sorted
    if n <= 1:
        return
    # take a copy of the left and of the right half of a .
    # (This corresponds to the lower part of the figure above.)
    l = a[:n//2]
    r = a[n//2:]
    # sort the two copies recursively ...
    merge_sort(l)
    merge_sort(r)
    # ... and merge them back into the original list a.
    # (This corresponds to the upper part of the figure above.)
    merge(l, r, a)
```



Last but not least, we can now complete the two question marks in the figure from above. Here, merge sort algorithm recursively sorts the two smaller lists. The figure can be completed as follows:



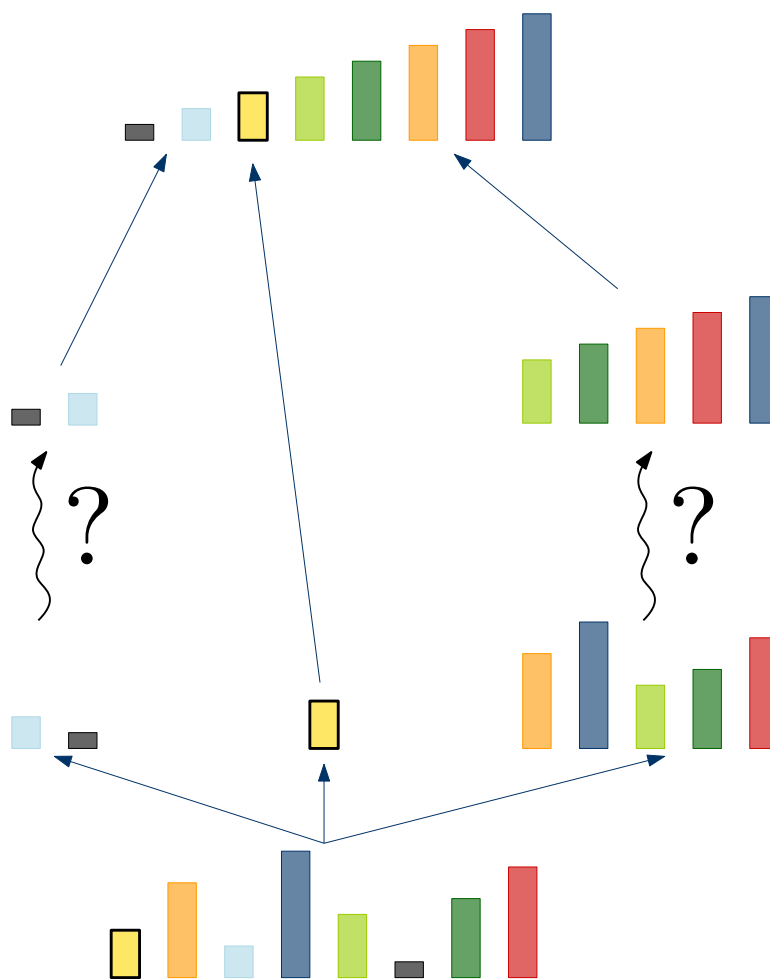
The first three steps correspond to the splitting part. The lists are split until they have size one. After that, the merge process brings the lists together, until a single list remains.

## 7.5 Quicksort

*Quicksort* is based on an alternative application of the divide-and-conquer paradigm. The general approach is as follows:

- **Divide:** We choose an (arbitrary) element of the input list  $a$ . We call this element the *pivot element*. Then, we subdivide  $a$  into three parts: (i) a sequence  $l$  that contains all elements of  $a$  that are smaller than the pivot element (ii) the pivot element itself; and (iii) a sequence  $r$  that contains all elements of  $a$  that are larger than or equal to the pivot element. (except for the pivot element itself);
- **Recurse:** We use recursion to sort the two sequences  $l$  and  $r$ .
- **Conquer:** We concatenate the sorted list  $l$  with the pivot element and the sorted list  $r$ . This gives a sorted order for  $a$ .

The main work happens in the divide step, when we split  $a$  into three parts. For this, we need a special *partition* procedure that we will explain below. The overall idea is depicted in the following figure.



Thus, we have to think about the crucial part of quicksort: how can we subdivide the elements of  $a$  into a left and a right part, according to a pivot element? Merge sort needs two separate lists for the subdivision step and the merge step. If  $a$  is large, this means that merge sort requires a lot of extra memory.

One advantage of quicksort is that the partition step can happen inside `a`. More precisely, the idea is as follows: let `l` and `r` be two indices. Our goal is to partition the sublist `a[l:r]` of `a` (recall that `a[l:r]` consists of the elements from `a[l]` to `a[r - 1]`).

For this, we pick `a[l]` as the pivot element. We want to rearrange the sublist `a[l + 1:r]` such that first we have all elements that are smaller than `a[l]`, followed by all elements that are larger than or equal to `a[l]`. We use an index `i` to go through the elements from `a[l + 1]` to `a[r - 1]`, and we introduce an additional index `m` that separates the two parts of the list. In each iteration, we maintain the following condition:<sup>1</sup>

All the elements from `a[l + 1]` to `a[m]` are smaller than `a[l]`, and all the elements from `a[m + 1]` to `a[i]` are larger than or equal to `a[l]`.

Note that the first or the second sublist can be empty (if `m == l`, the first sublist is empty; if `m == i`, the second sublist is empty). Suppose that the condition holds and that we are looking at the next element `a[i]`. There are two cases:

- `a[i] >= a[l]`: in this case, the condition also holds if we include `a[i]`. There is nothing to do.
- `a[i] < a[l]`: in this case, we need to act. The idea is to move the element `a[i]` to the position `m + 1`. This is the position right after the first sublist. To do this efficiently, we simply swap the element `a[i]` with the element `a[m + 1]`. Then, we increase `m` by one. In this way, the condition is again satisfied: all the elements from `a[l + 1]` to `a[m]` are smaller than `a[l]`, and all the elements from `a[m + 1]` to `a[i]` are larger than or equal to `a[l]`.<sup>2</sup>

In the last step, we swap `a[l]` with `a[m]`, in order to move the pivot element into the correct position.<sup>3</sup> The Python-implementation is as follows:

```
# partition(a: List[U], l: int, r: int): int
# U is a totally ordered universe
# Precondition: 0 <= l < r <= len(a),
# Postcondition:
#     Effect: If m is the return value, then all elements are still in a, and
#             for l <= i < m, we have a[i] < a[m] and
#             for m < i < r, we have a[m] <= a[i].
#     Result: The index m of the pivot element is returned.
'''Tests: Your exercise ;-)''''
def partition(a, l, r):
    # choose a[l] as the pivot
    pivot = a[l]
    # initially, both sublists are empty
    m = l
```

<sup>1</sup>A condition that holds at each iteration of a loop is called a *loop invariant*. We will go into much more detail about loop invariants later in Section 9.3.

<sup>2</sup>Note that if the second sublist is empty, then we swap `a[i]` with itself.

<sup>3</sup>Note that if the first sublist is empty, then we swap `a[l]` with itself.



```
for i in range(l + 1, r):
    if a[i] < pivot:
        swap(a, m + 1, i)
        m = m + 1
    # at this point, all elements from a[l + 1] to a[m] are < pivot,
    # and all elements from a[m + 1] to a[i] are >= pivot
# put the Pivot into the correct position
swap(a, l, m)
return m
```

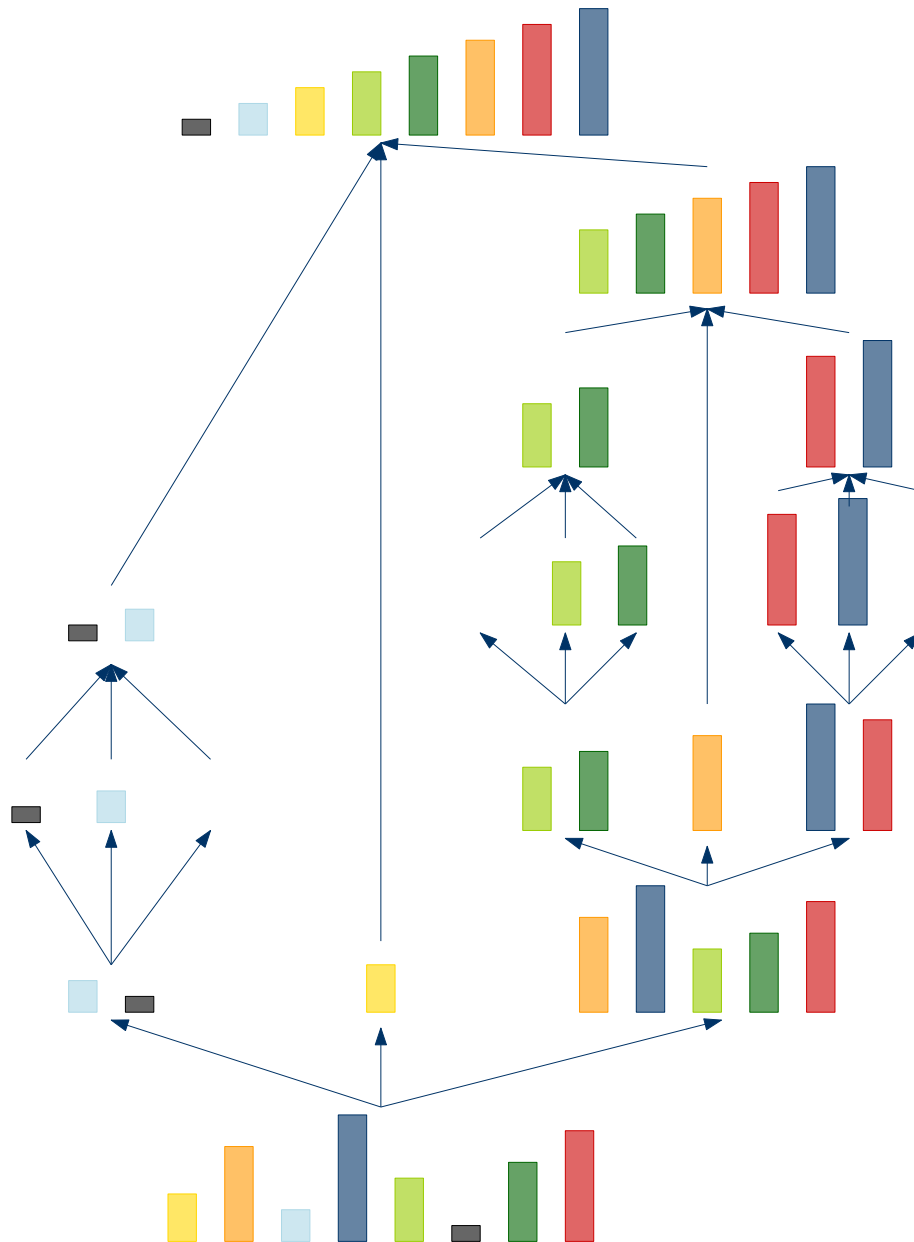


Using `partition(·)`, it is easy to implement the quicksort algorithm as follows:

```
# quicksort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
'''Tests: Your exercise ;-)''''
def quicksort(a):
    # This help function sorts the elements in a between
    # start (inclusive) and end (exclusive)
    def quicksort_help(l, r):
        if r - l <= 1: # Recursion anchor
            return
        # The splitting part.
        # This corresponds to the lower part of the figure above.
        m = partition(a, l, r) # m is the index of the pivot element
        # Sort the two remaining lists recursively.
        quicksort_help(l, m)
        quicksort_help(m + 1, r)
    quicksort_help(0, len(a))
```



Finally, we can complete the figure that illustrates quicksort by replacing the question marks with the correct process:



When comparing the two pictures for quicksort and merge sort, we can see that quicksort is not as symmetric as merge sort. The reason is that in the first step of quicksort, we have chosen a pivot element that is rather small compared to the other elements. This results in a very small and a very large subsequence. This results in the asymmetry. Thus, in quicksort it can happen that the partition step leads to unbalanced lists and to a bad running time. We will talk more about this in Section 8.3.4.

### 7.6 Stability

Now, we briefly discuss the notion of *stability* in sorting algorithms. A sorting algorithm is stable if elements with the same value do not change their order in the sorting permutation. Formally, the algorithmic problem of stable sorting is as follows:

#### Stable sorting:

Possible inputs: sequences  $X = [x_1, \dots, x_n]$  with elements from a totally ordered universe  $U$ .

Desired output for an input  $X$ : a rearranged sequence  $X' = [x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)}]$ , where  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$  is a permutation such that  $x_{\pi(1)} \leq x_{\pi(2)} \leq \dots \leq x_{\pi(n)}$  and such that for all  $i$  from 1 to  $n-1$ , we have that  $x_{\pi(i)} = x_{\pi(i+1)}$  implies  $\pi(i) < \pi(i+1)$ , for all  $i$  from 1 to  $n-1$ .

In other words, in the stable sorting problem, we require that elements with the same value appear in the output sequence in the same order as in the input sequence.

For example, consider the input sequence

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$
3	2	1	2	5	4	3

Then, the output sequence

$x_3$	$x_4$	$x_2$	$x_1$	$x_7$	$x_6$	$x_5$
1	2	2	3	3	4	5

corresponds to the permutation

$\pi(1)$	$\pi(2)$	$\pi(3)$	$\pi(4)$	$\pi(5)$	$\pi(6)$	$\pi(7)$
3	4	2	1	7	6	5

It does *not* satisfy the condition of the stable sorting problem. Indeed, we have  $x_{\pi(2)} = x_{\pi(3)} = 2$ , but  $\pi(2) = 4 > 2 = \pi(3)$ . In other words, the elements  $x_2$  and  $x_4$  of the input sequence have the same value 2, but they have switched their order from the input sequence to the output sequence.

On the other hand, the output sequence

$x_3$	$x_2$	$x_4$	$x_1$	$x_7$	$x_6$	$x_5$
1	2	2	3	3	4	5

corresponds to the permutation

$\pi(1)$	$\pi(2)$	$\pi(3)$	$\pi(4)$	$\pi(5)$	$\pi(6)$	$\pi(7)$
3	2	4	1	7	6	5

It *does* satisfy the condition of the stable sorting problem. Indeed, the input elements  $x_2$  and  $x_4$  have the same value 2, and they have the same order in the input sequence and in the output sequence. Similarly, the input elements  $x_1$  and  $x_7$  both have the

same value 3, and they have the same order in the input sequence and in the output sequence.

Stable sorting is important when sorting *tuples*—for example, such a tuple could contain the first name, the family name, the id-number, the grades, etc. for a given student. A stable sorting algorithm would then keep the correct order—for example, if we first sort all students by their first name, and then by their family name, then all students with the same family name would still be sorted by their first name. Stability is crucial for this to work.

We will now examine whether the four sorting algorithms from this chapter are stable. However, let us emphasize that the stability of a sorting algorithm depends on the exact implementation. Sometimes, even a tiny change in the implementation can affect the stability. Moreover, there is a simple method to turn any given sorting algorithm into a stable version, albeit at the cost of a slightly larger memory footprint and a slightly larger running time.

- **Selection sort.** Selection sort is *not stable*. In each iteration, we swap the minimum to the front of the remaining list. This may change the order of elements with the same value. For example:

$$\text{selection\_sort}([1_1, 1_2, 0]) \mapsto [0, 1_2, 1_1]$$

- **Insertion sort.** Insertion sort is *stable*. When we swap the next element  $a[j]$  down the list, we stop as soon as we encounter the first element that is smaller or equal to  $a[i]$  (that is, the **while**-loop continues only as long as we have  $a[j - 1] > a[j]$ ). Thus, two elements with the same value will never change places, and our implementation of insertion sort is stable. However, if we change the condition in the **while**-loop to  $a[j - 1] \geq a[j]$ , then the implementation would still be correct, but not stable.
- **Merge sort.** Merge sort is *stable*. In  $\text{merge}(\cdot)$ , if the next elements in  $l$  and in  $r$  are equal, we pick the next element from  $l$  for  $a$  (i.e., the condition for the next element is  $l[i] \leq r[j]$ ). In this way, two elements with the same value will never change places. However, if we change the condition to  $l[i] < r[j]$ , then the implementation would still be correct but not stable.
- **Quicksort.** Quicksort is *not stable*. In  $\text{partition}(\cdot)$ , when we execute the instruction  $\text{swap}(a, l, m)$  to switch the pivot element with the element  $a[m]$ , we may change the order of two elements with the same value.

## 7.7 Memory usage

Sorting algorithms (or algorithms in general) typically need extra memory in addition to the input. Of course, one goal for an efficient program must be to keep this extra

memory as small as possible, since memory space is a valuable resource. For sorting algorithms, we distinguish between *in-place* algorithms and *out-of-place* algorithms.

A sorting algorithm is *in-place* if it uses only a constant amount of memory in addition to the input list  $a$ . More precisely, this means that the number of *additional memory cells* does not depend on the size of  $a$ . These additional memory cells can store numbers, e.g., for indices that appear in a loop, or elements from  $a$ , e.g., the temporary variable used in the swap-routine.

Selection sort and insertion sort are in-place. Selection sort uses only four integer variables:  $n$ ,  $i$ ,  $\text{min\_index}$ , and  $j$ . In addition, `swap(.)` uses the temporary variable `temp`. Insertion sort uses only three integer variables:  $n$ ,  $i$ , and  $j$ , as well as the temporary variable in `swap(.)`.

In contrast, a sorting algorithm is *out-of-place* if the number of additional memory cells depends on the number of elements in  $a$ . Merge sort is out-of-place. The lines

```
l = a[:n//2]
r = a[n//2:]
```



mean that we create two *new* lists  $l$  and  $r$  and that we copy the  $n$  elements of  $a$  into  $l$  and  $r$ . Hence, in one recursive call, we need about  $n$  additional memory cells. Thus, the number of additional memory cells in merge sort depends on the size of  $a$ .

### 7.7.1 The hidden space of recursion

When looking at quicksort, one might think that it is in-place: both `partition(.)` and `quicksort_help(.)` do not create any additional lists and both introduce only a constant number of additional indices and temporary variables. However, it turns out that quicksort is actually *not* in-place. Instead, the recursive calls in quicksort require a super-constant amount of additional memory cells and turn quicksort into an out-of-place algorithm.

To understand the relationship between recursion and memory usage, we go back to the von-Neumann architecture and take a closer look at how programs typically use the main memory.

**Memory layout when executing a program.** When we execute a program, the main memory for this program is typically divided into four parts. The details depend on the operating system and the programming language, but the general structure is almost always the same.

- **Code segment (also: program segment, text segment).** The code segment contains the machine code instructions of the program. From here, the CPU fetches the instructions that are executed. Typically, the contents of the code segment are fixed.

- **Data segment.** The data segment contains all the global variables that appear in the program. Typically, the size of the data segment is fixed, but the content changes as the global variables change their values.
- **Heap (also: free space).** The heap<sup>4</sup> is used for the dynamic allocation of large data objects. The programming language offers special functionality to allocate and deallocate objects on the heap, to find free space on the heap, and to remove unused objects. The heap is very flexible, but also slow.
- **Stack.** The stack is used to store local variables in functions and to manage recursive calls. This part of the memory can be accessed quite fast, but it is rather limited. Often, a local variable on the stack serves as some kind of “reference” to a larger object on the heap.

**Function calls and the stack.** When we call a function, the local variables and the actual parameters of the function are temporarily stored on the stack. Once the function call ends, the memory is freed and can be reused.<sup>5</sup>

However, if a function recursively calls itself, then each recursive call needs some memory on the stack. Therefore, the memory usage of a recursive program is proportional to the recursion depth.

In the worst case, quicksort may need  $n$  recursive calls in a row, i.e., the recursion depth of quicksort is  $n$ . This happens for example when the chosen pivot element is always the smallest element in the current sublist. Thus, the number of additional memory cells in quicksort depends on the number of elements in  $a$ , and quicksort is *out-of-place*.

---

<sup>4</sup>Later, in Section 15.2.2, you will encounter another structure that is also called *heap*. These two structures are completely unrelated, so it is important to keep them separate.

<sup>5</sup>Thus, in our analysis of selection sort and insertion sort, we were a bit sloppy: the function calls to `swap(·)` also need additional space on the stack. However, there is only one call at a time, and no recursion takes place. Thus, this additional space on the stack does not depend on the input size.

## KAPITEL 8

### Running Time

#### Lernziele

**KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

**KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.

**KdP7** Du entwickelst Algorithmen zur Lösung vorgegebener *algorithmischer Listen- und Baumprobleme* und stellst diese mithilfe unterschiedlicher Programmierkonzepte in *Scala und Python* dar.

**KdP8** Du analysierst *Algorithmen*, indem du *Korrektheit* (auf Grundlage der Spezifikation), *Speicherplatz* und *Laufzeit* angibst und begründest.



#### 8.1 Fundamentals of running time

When solving an algorithmic problem, we want the resulting solution to be *efficient*. This means that we want our algorithm to be as fast as possible and to consume as little memory as possible (among other goals). Sometimes, these may be conflicting goals. In this class, we focus on the running time of an algorithm, because this is often the primary concern. The topic of memory consumption will be discussed in later classes—but many ideas are very similar as for running time.

First, we need to clearly understand what we mean by a “fast” algorithm. There are many ways to answer this question. In this chapter, we take a *theoretical viewpoint*. That is, our notion of a “fast algorithm” should be independent of the current hardware, the current operating system, or the details of the programming language and the runtime environment. The notion of a “fast algorithm” should purely depend on the algorithm itself. Hence, we would like to have a general, *machine-independent formalism* to analyse our algorithms.

The solution is to define a notion of *elementary operations* that the algorithm executes in order to achieve its goal. Examples of typical elementary operations are:

- comparisons,
- arithmetic or Boolean operations,
- assignments,
- function calls,
- memory look-ups,
- etc.

These elementary operations should be independent of the concrete hardware, but concrete enough to give a representative picture of the “real-world” running time of the algorithm. Depending on the situation, different sets of elementary operations may be appropriate.

Once we have settled on a set of elementary operations, we can count for each specific input how many elementary operations the algorithm needs for this input. However, there are infinitely many inputs, so this approach may not be very useful to get a general picture of the speed of an algorithm.

The solution is to provide a summary of how the running time of an algorithm behaves as the inputs grow larger. That is, our goal is to determine a *running time function* that relates the *input size* to the number of elementary operations that an algorithm needs for this input size. To do this, we first need to make concrete the notion of “input size”. Again, there are many possibilities. Examples of typical input sizes are:

- number of elements in the input,
- size of an input lists list,
- number of binary digits in an input number,
- etc.

Once we have agreed on a notion of an input size, we can fix an  $n \in \mathbb{N}$ , and we can look at all inputs of size  $n$ . To define our running time function, we need to find a single number that represents the performance of the algorithm on all inputs of size  $n$ . Once again, there are many ways to do this. The simplest way is to take the maximum of all these running times, the *worst possible running time* that the algorithm can have on input size  $n$ . This process is called the *worst-case analysis* of an algorithm.

**Worst-case analysis.** We now give the precise definition of worst-case analysis. Let  $A$  be an algorithm and let  $I$  be an input for  $A$ . Then, we denote by  $T_A(I)$  the number of elementary steps that  $A$  performs on  $I$ . We call  $T_A(I)$  the *running time of  $A$  on input  $I$* . Let  $|I|$  denote the size of input  $I$ . Then, the *worst-case running time function*  $T_A : \mathbb{N} \rightarrow \mathbb{N}$  is defined as

$$T_A(n) = \max_{\substack{I \text{ input for } A \\ |I|=n}} T_A(I).$$



## 8.1.1 Linear search

As a concrete example, let us perform a worst-case analysis of the linear search algorithm LS from Section 6.2.1. Recall that LS receives as input a list  $a$  and a search key  $k$  (this is the input  $I$  for LS) and it returns the index of  $k$  in  $a$ ; or **None**, if  $k$  is not in  $a$ .

To perform a worst-case analysis of LS, we first need to agree on two notions: the *elementary operations* and the *input size*.

- **Elementary operations.** We focus on only one type of elementary operation: comparisons between the search key  $k$  and elements of the input list  $a$ . All other steps in LS are “for free”.
- **Input size.** Let  $I = (a, k)$  be an input. We define input size  $|I|$  of  $I$  as the number of elements in  $a$ . We ignore  $k$  for the input size.

Note that different choices for the elementary operations and the input size are possible and reasonable. The goal is to achieve a trade-off between simplicity and usefulness. We want to make the analysis as simple as possible, but we would also like to have a result that is representative of the “real” behavior of LS. This is achieved by our two choices. Counting only comparisons as our elementary operations is simple (because we only need to focus on one thing), but it is also representative (because the main work of LS is in doing comparisons). Ignoring  $k$  for the input size is simple (because we only need to focus on one thing), but it is also representative (because only the length of  $a$  differs between inputs).

With these two definitions in place, we can determine the running time of LS for some concrete inputs. For example, consider the input  $I_1 = ([3, 1, 5, 2, 4], 1)$ . Then, we have  $T_{LS}(I_1) = 2$ , since two comparisons are needed to locate the search key 1 in the list  $[3, 1, 5, 2, 4]$ . On the other hand, for the input  $I_2 = ([3, 1, 5, 2, 4], 2)$ , we have  $T_{LS}(I_2) = 4$ , because four comparisons are necessary to find the search key 2. Finally, for the the input  $I_3 = ([3, 1, 5, 2, 4], 7)$ , we have  $T_{LS}(I_3) = 5$ , because five comparisons are necessary to determine that 7 does not appear in the list.

To determine the *worst-case* running time  $T_{LS}(5)$  for *all* possible inputs of size 5, we must analyze how many comparisons are needed *in the worst case* to find a search key  $k$  in a list  $a$  of 5 elements (or to determine that  $k$  is not present in  $a$ ). The answer is 5, because in the worst case, we must compare  $k$  with every single elmenet of  $a$ . Thus, we have  $T_{LS}(5) = 5$ .

More generally, we would like to determine the worst-case running time  $T_{LS}(n)$  for all possible inputs of size  $n$ . Here, we have  $T_{LS}(n) = n$ , because in the worst case, we must compare the search key  $k$  with every single element in the input list. No matter how the input  $I$  looks like, we always have  $T_{LS}(I) \leq n$ —this is the nature of worst case analysis.



**Attention:** In this section, we used the notations  $T_A(I)$  and  $T_A(n)$ . From now on, we will omit the index  $A$ , if the context is clear. Moreover, when doing the worst-case analysis, we will not use the notation  $T(I)$ . Instead, we only focus on  $T(n)$ . Thus, from now on,  $T(n)$  will denote the *worst-case running time* of the algorithm under consideration.

### 8.1.2 Cartesian product

Let us consider another example. The following algorithm computes the *Cartesian product* of two input lists  $a$  and  $b$ :

```
def cart_product(a, b):
    prod = []
    for x in a:
        for y in b:
            prod.append((x, y))    # (*)
    return prod
```



Again, before we can perform a worst-case analysis of `cart_product(·)`, we first need to agree the *elementary operations* and the *input size*.

- **Elementary operations.** We focus on only one type of elementary operation: the `append(·)`-calls in line `(*)`. All other operations are “for free”.
- **Input size.** Let  $I = (a, b)$  be an input. We define input size  $|I|$  of  $I$  as the *pair*  $(n, m)$  of natural numbers, where  $n$  denotes number of elements in  $a$  and  $m$  denotes the number of elements in  $b$ .

Again, our choices are supposed to allow for a simple analysis, while at the same time providing a realistic picture of the algorithm. Thus, it is reasonable to focus only on the `append(·)`-calls, since this is where the main work happens. In this example, the input size is not given by a single parameter, but by *two* parameters. The length of  $a$  and the length of  $b$ . This is reasonable because these two sizes are independent of each other, and we would like to determine how the running time depends on each of the two parameters. Consequently, the worst-case running time will be a function  $T(n, m)$  of *both*  $n$  and  $m$ .

Now, we fix an input size  $(n, m)$ , and we determine the maximum number of `append(·)`-calls in `(*)` for a list  $a$  of  $n$  elements and a list  $b$  of  $m$  elements. The function `cart_product(·)` goes through all elements of  $a$ . For each such element, it executes the inner **for**-loop to iterate over all elements in  $b$ . Thus, the inner **for**-loop leads to  $m$  `append(·)`-calls in `(*)`. Since the inner **for**-loop is executed exactly  $n$  times—once for each element in  $a$ —the `append(·)`-function is called exactly  $n \cdot m$  times. This behavior is the same for all inputs of size  $(n, m)$ . Hence, the worst-case running time of `cart_product(·)` is  $T(n, m) = n \cdot m$ .

## 8.1.3 Binary search

In Section 6.2.2, we discussed the binary search algorithm:

```
def binary_search(a, k):
    def bin_search(left, right):
        if left >= right:
            return None
        m_pos = left + (right - left) // 2
        m = a[m_pos]
        if k == m:
            return m_pos
        elif k < m:
            return bin_search(left, m_pos)
        else: # m < k
            return bin_search(m_pos + 1, right)
    return bin_search(0, len(a))
```



Now, let us perform a worst-case analysis of binary search. As usual, we first need to agree on the *elementary operations* and the *input size*. We use the same choices as for linear search.

- **Elementary operations.** We focus on only one type of elementary operation: comparisons between the search key  $k$  and elements of the input list  $a$ . All other steps in LS are “for free”.
- **Input size.** Let  $I = (a, k)$  be an input. We define input size  $|I|$  of  $I$  as the number of elements in  $a$ . We ignore  $k$  for the input size.

It is actually quite reasonable to use the same choices for both linear search and binary search. After all, both algorithms solve the same algorithmic problem “Searching in a sorted list”. Thus, it is expected that we will want to compare the two algorithms with respect to their worst-case running time. For this comparison to make sense, we need to make sure that both algorithms are analysed according to the same rules.

Now, our goal is to bound the maximum number of comparisons that binary search performs for an input list with  $n$  elements. In Theorem 6.4, we already gave an analysis of binary search. More precisely, we showed that the maximum number of invocations of `bin_search(·)` for an input list with  $n$  elements is at most  $\log_2(n) + 2$ . However, now we do not count the number of invocations of `bin_search(·)`, but the number of comparisons. Thus, we need to look a bit more closely at `bin_search(·)`. There, we see that every time there is a recursive call in `bin_search(·)`, we perform *two* comparisons (namely, the comparisons  $k == m$  and  $k < m$ ). Furthermore, in the last invocation of `bin_search(·)`, we perform at most one comparison (namely, the comparison  $k == m$ ). Thus, the total number of comparisons in `bin_search(·)` is never more than  $2\log_2(n) + 3$ , and the worst-case running time of binary search is  $T(n) \leq 2\log_2(n) + 3$ .<sup>1</sup>

<sup>1</sup>Note that we write  $T(n) \leq 2\log_2(n) + 3$  and not  $T(n) = 2\log_2(n) + 3$ . The reason is that the actual worst-case running time function may be a bit smaller, because we used estimates that are not

This worst-case running time is a bit complicated, because it contains a constant factor and a constant term that muddy the picture. To make our notation simpler and more succinct, we will use “asymptotic notation” or “ $O$ -notation”, as explained in the next section.

## 8.2 $O$ -notation

As we have seen in the previous section, the running-time function can look quite complicated. This is not in our interest, because we would like to have a simple and succinct representation of the performance of an algorithm. To solve this problem, we will use  *$O$ -notation* (or *asymptotic notation*) to capture the essence of a running-time function, without the distracting details. The result is called the *asymptotic worst-case running time* of an algorithm. To obtain the asymptotic worst-case running-time, we first estimate the running-time function  $T(n)$  and then classify  $T(n)$  according to the  $O$ -notation.

In this class, we will consider the  $O$ -notation only briefly. In later classes, we will go into much more detail. Let  $f: \mathbb{N}_0 \rightarrow \mathbb{R}^+$  be a function. Then,  $O(f(n))$  is defined as the set of all functions  $g: \mathbb{N}_0 \rightarrow \mathbb{R}^+$  that have  $f(n)$  as an *asymptotic upper bound*.<sup>2</sup> We write  $g(n) = O(f(n))$ . For our purposes, the following three rules are sufficient to understand what an “asymptotic upper bound” is:

1. We ignore constant factors. For example, we have

$$5 \cdot \log_2(n) = O(\log_2(n)), \text{ and } 3 \cdot m \cdot n = O(m \cdot n).$$

2. We ignore lower-order additive terms. For example we have

$$\log_2(n) + 1 = O(\log_2(n)) \text{ and } n + \log_2(n) = O(n).$$

3. The  $O$ -notation gives an upper bound. For example, we have

$$\log_2(n) = O(n) \text{ and } n = O(n^2).$$

In addition to simplicity, one main advantage of using asymptotic notation is that it makes our results more *robust*. To perform a precise worst-case analysis, we need to fix a set of elementary operations. Then, we estimate the maximum number of elementary operations that an algorithm performs for any given input size. The precise result depends very much on our precise choice of elementary operations, and different choices may lead to different results. However, among different choices of elementary operations that are sufficiently representative, the worst-case running times usually

---

necessarily tight. With more effort, it may be possible to determine the *exact* worst-case running time. However, we do not follow this route. This would make the argument more difficult, and the analysis is already representative of the behavior of binary search.

<sup>2</sup>Formal definitions are given in later classes

differ only by constant factors and lower-order additive terms. Thus, in each case, we get the same asymptotic worst-case running time. This makes the notion of asymptotic worst-case running time very useful in evaluating the performance of an algorithm.

**Attention:** For the sake of readability, from now on, we will omit the base of the logarithm. That is, whenever we write  $\log(n)$ , we assume that the base of the logarithm is 2.

Moreover, we will sometimes omit the parentheses around the argument of the log-function. This is also supposed to improve the overall readability. For example, we will write  $\log n = O(n)$  instead of  $\log_2(n) = O(n)$ .



### 8.2.1 Typical asymptotic running times

In principle, any function can occur as the asymptotic running time of an algorithm. However, in practice, there are a few asymptotic running times that appear very frequently. Here, we mention a few examples with typical scenarios when they occur:

- **Constant running time**  $O(1)$ . The algorithm performs a simple task whose running time does not depend on the actual size of the input. For example, accessing an element in a list of  $n$  elements in Python.
- **Logarithmic running time**  $O(\log n)$ . Typically occurs if the algorithm performs a recursion that in each step takes constant time and halves the input. For example, binary search.
- **Square-root running time**  $O(\sqrt{n})$ . Typically occurs if the algorithm performs some task that considers a significant portion of the input, but not all of it. For example, to test whether a given natural number  $n$  is prime, we only need  $O(\sqrt{n})$  integer divisions, because if  $n$  is not a prime number, it has a prime factor between 2 and  $\lceil \sqrt{n} \rceil$ . Another example is Grover's search algorithm in quantum computing.
- **Linear running time**  $O(n)$ . Typically occurs if the algorithm performs a single or a constant number of scans over the input. For example, linear search or finding the minimum or maximum element of an unsorted list
- **Linearithmic running time**  $O(n \log n)$ . This is a typical running time that occurs for comparison-based sorting algorithms, such as merge sort or heapsort. We will say more about this below, in Sections 8.3 and 8.4.2.
- **Quadratic running time**  $O(n^2)$ . This is the running time of insertion sort or selection sort (see Section 8.3). This running time also occurs for algorithms that consider all pairs of elements in an input list.

- **Cubic running time**  $O(n^3)$ . This is the running time for Gaussian elimination to solve a set of linear equations. It also occurs for algorithms that consider all triples of elements in an input list.
- **Polynomial running time**  $O(n^k)$ , for some  $k > 0$ . In Computer Science, an algorithm that requires polynomial running time is considered to be *efficient*. We will talk more about this in “Grundlagen der theoretischen Informatik”.
- **Exponential running time**  $O(2^n)$ . Typically occurs in algorithms that need to consider all subsets of a given set of input elements or that need to explore all possibilities in a set of  $n$  binary choices. Algorithms with exponential running time are usually considered to be *inefficient*.
- **Factorial running time**  $O(n!)$ . Typically occurs in algorithms that need to consider all ways to arrange a set of  $n$  elements in a linear order. Algorithms with factorial running time are usually considered to be *inefficient*.

### 8.3 Worst-case analysis of sorting algorithms

We will now perform a worst-case analysis of the sorting algorithms in Chapter 7. Here are the necessary definitions of the *elementary operations* and the *input size*.

- **Elementary operations.** We focus on only one type of elementary operation: comparisons between two elements in the input list  $a$ . All other steps are “for free”.
- **Input size.** Let  $I = a$  be an input. We define input size  $|I|$  of  $I$  as the number of elements in  $a$ .

Hence, in this section,  $T(n)$  will be the worst-case number of comparisons that a sorting algorithm requires for an input list with  $n$  elements.

#### 8.3.1 Selection sort

Recall that in Section 7.2, we implemented selection sort as follows:

```
def selection_sort(a):
    n = len(a)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if a[j] < a[min_index]:    # (*)
                min_index = j
        swap(a, i, min_index)
```



The only place where we compare two elements of  $a$  is in line  $(*)$ . Thus, we must count how often line  $(*)$  is executed.

For this, we fix an iteration  $i$  of the outer **for**-loop. For this  $i$ , the inner **for**-loop is executed  $n - (i + 1)$  times, since  $j$  goes from  $i + 1$  to  $n - 1$ . In each iteration of the inner **for**-loop, the line  $(*)$  is executed exactly once. Hence, the total number of comparisons is

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \dots + 2 + 1 + 0 \\ &= 1 + 2 + \dots + (n-2) + (n-1) \\ &= \frac{n \cdot (n-1)}{2} \\ &= \frac{1}{2}n^2 - \frac{1}{2}n. \end{aligned}$$

Here, we used the well-known formula summation formula for the first  $n - 1$  natural numbers. To obtain the asymptotic running time, we ignore the constant factor  $\frac{1}{2}$  and remove the lower-order additive term  $-\frac{1}{2}n$ . Thus, we can conclude that selection sort has asymptotic worst-case running time  $T(n) = O(n^2)$ .

During the analysis, we did need to not make any assumption on the structure of the input list  $a$ . Hence, we actually found out that selection sort has the same running time for every input of size  $n$ .

### 8.3.2 Insertion sort

The worst-case analysis for insertion sort is very similar to the worst-case analysis for selection sort. However, this time, the precise running time depends on the structure of the input list. We leave the details as an exercise.

### 8.3.3 Merge sort

In Section 7.4, we implemented merge sort as follows:

```
def merge(l, r, a):
    i = 0
    j = 0
    k = 0
    while i < len(l) and j < len(r):
        if l[i] <= r[j]:          # (*)
            a[k] = l[i]
            i = i + 1
        else:
            a[k] = r[j]
            j = j + 1
        k = k + 1
    while i < len(l):
        a[k] = l[i]
        i = i + 1
        k = k + 1
    while j < len(r):
```





```

    a[k] = r[j]
    j = j + 1
    k = k + 1

def merge_sort(a):
    n = len(a)
    if n <= 1:
        return
    l = a[:n//2]
    r = a[n//2:]
    merge_sort(l)
    merge_sort(r)
    merge(l, r, a)

```



The only place where we compare two elements of the input list  $a$  is in the line marked by  $(*)$  in `merge(·)`.<sup>3</sup> Thus, our goal is to count the maximum number of times the line  $(*)$  is executed for any input list  $a$  of length  $n$ . For the sake of simplicity, we will assume that  $n$  is a power of 2, i.e., we will assume that there is some natural number  $k$  such that  $n = 2^k$ .<sup>4</sup>

Let us first focus on a single invocation of the `merge(·)`-function, and suppose that the output list  $a$  has  $m$  elements. The total number of executions of  $(*)$  in a single invocation `merge(·)` can vary (depending on how the elements in  $l$  and  $r$  relate to each other), but it is never more than  $m$ : in each iteration of the first **while**-loop, we add one new element to  $a$ , and  $a$  contains  $m$  elements.<sup>5</sup> Thus, for a single invocation of `merge(·)` with an output list of  $m$  elements, the number of executions of  $(*)$  is at most  $m$ .

Now let us turn to `merge_sort(·)`. In general, the function `merge_sort(·)` performs two recursive calls to `merge_sort(·)` with lists of size  $n/2$ , plus a call to `merge(·)` with an output list of size  $n$ . Thus, the number of executions of  $(*)$  is at most  $n$  (for the call to `merge(·)`) *plus the number of executions of  $(*)$  that are performed by the recursive calls*. Now, we must find a way to deal with this recursion in our worst-case analysis. The idea is to write a *recurrence relation* for the worst-case running time of merge sort. That is, we write

$$T(n) \leq 2 \cdot T(n/2) + n \quad (8.1)$$

to express the fact that the worst-case number of executions of  $(*)$  for an input list of size  $n$  is at most  $n$  (for the call to `merge(·)`), plus the worst-case number of executions of  $(*)$  for the two recursive calls with input lists of size  $n/2$ . In addition, we know that

$$T(0) = T(1) = 0, \quad (8.2)$$

<sup>3</sup>Actually, at this point, the two elements lie in the new lists  $l$  and  $r$ . Originally, however, they were copied from the input list  $a$ . Hence, they count for our elementary steps.

<sup>4</sup>The analysis also holds without this assumption, but the calculations get a bit more messy.

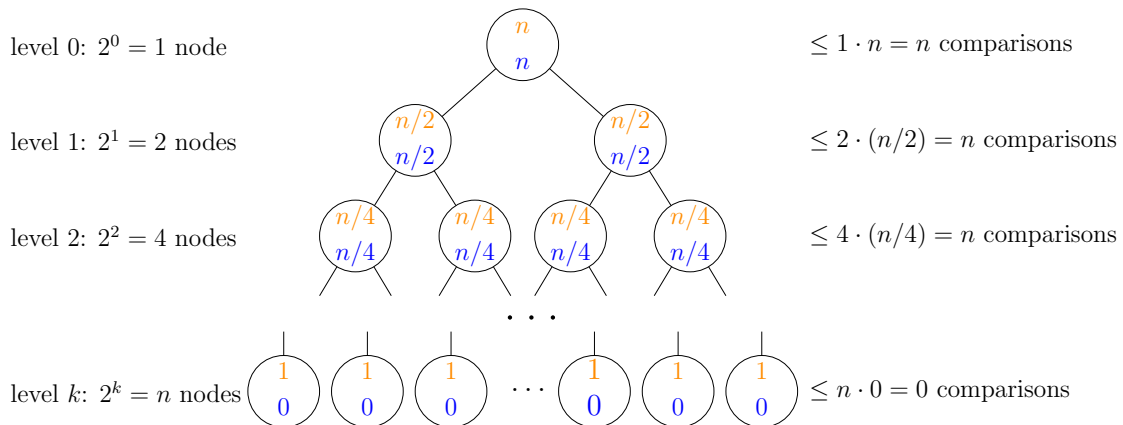
<sup>5</sup>Actually, the number of executions of  $(*)$  is never more than  $m - 1$ , because the first **while**-loop will terminate before either  $l$  or  $r$  is traversed completely. However, this additional amount of precision will not affect the result of the analysis, so we stick with the simpler bound  $m$ .



because if we apply merge sort to a list with at most one element, the algorithm immediately stops, since the list is already sorted.

Now, (8.1) and (8.2) are called a *recurrence relation* for the worst-case running time  $T(n)$ . They give a way to define the value  $T(n)$  for the parameter  $n$  in terms of the value  $T(n/2)$  for the smaller parameter  $n/2$ , together with an anchor for smallest parameters  $n = 0$  and  $n = 1$ . To complete the worst-case analysis of merge sort, our task is now to *solve* the recurrence relation (8.1, 8.2), i.e., to find a function  $T(n)$  that satisfies (8.1, 8.2) and that is as small as possible. Solving recurrence relations is a large topic in the analysis of algorithms, and there are many techniques to approach this task. Here, we will present two simple ways that are often effective in finding a solution.

**The tree method.** In the tree method, we visualize the recurrence relation (8.1, 8.2) in a diagram, and we use this diagram to understand the behavior of the recursion and of the resulting running time. More precisely, we draw a hierarchical diagram that has a *node* for every invocation of `merge_sort(·)`. In each node, we write the current input size, as well as the worst-case number of comparisons that are performed in this invocation of `merge_sort(·)` (without the recursive calls). Then, we connect the different nodes according to the function calls, in a hierarchical manner. The node for the first invocation of `merge_sort(·)` is called the *root* of the tree diagram. The nodes that correspond to the base cases of the recursion are called the *leaves* of the tree diagram. If a node  $v$  represents a direct recursive call from another node  $u$ , we say that  $v$  is a *child* of  $u$  in the tree diagram. The resulting diagram looks like this:



This diagram is called the *tree diagram* for the recurrence relation. We see that the structure of the recursion is very regular: the tree diagram can be partitioned into *levels*, where all the nodes of a level share the same *recursion depth*. Level 0 corresponds to the initial invocation of `merge_sort(·)`; level 1 to the recursive calls of depth 1; level 2 to the recursive calls of depth 2; etc. In each level, the size of the input lists decreases by a factor of two, and the number of nodes increases by a factor of two. That is, at level 0, we have one invocation of `merge_sort(·)` with an input list of size  $n$ ; at level 1, we have two invocations of `merge_sort(·)` with input lists of size  $n/2$ ;<sup>6</sup> etc. The

<sup>6</sup>Recall that we assumed that  $n = 2^k$  is a power of two. In this way,  $n/2$ ,  $n/4$ , etc. are all integers.

recursion finishes at level  $k$ , where we have  $n$  invocations of  $\text{merge\_sort}(\cdot)$  with input lists of size 1. Hence, the total number of levels is  $k + 1$ .

Now, we estimate the worst-case number of comparisons that are performed at each level of the tree diagram. We already saw that if the size of the input list is  $m$ , then the number of comparisons in this invocation of  $\text{merge\_sort}(\cdot)$  is at most  $m$ . Thus, in each level of the tree diagram (except for the last), the total number of comparisons is at most  $n$ . In the last level of the tree diagram, we have reached input lists of size 1, and the number of comparisons is 0. Thus, the total number of comparisons is at most  $n \cdot k + 0$ , because in each of the first  $k$  levels, we have at most  $n$  comparisons, and in the last levels, we have 0 comparisons. Now, since  $n = 2^k$ , we have  $k = \log n$ , so the worst-case running time of merge sort is  $T(n) \leq n \log n$ , and the asymptotic worst-case running time is  $T(n) = O(n \log n)$ .

We remark that this also holds if  $n$  is not a power of two. The same strategy applies: we can draw the tree diagram of the recurrence, and we can still conclude that the total number of comparisons at each level of the tree diagram is at most  $n$ . The only difference is that now we need to worry about rounding issues: the input lists cannot always be split into pieces of exactly the same size. Thus, the tree diagram is not as regular as for the case that  $n = 2^k$ . Some branches of the tree finish earlier than others. However, it is still true that the total number of levels is  $O(\log n)$  and that the total number of comparisons per level is at most  $n$ . Thus, we have  $T(n) = O(n \log n)$ , also for general  $n$ .

**Repeated application of the recurrence relation.** A second way to solve a recurrence relation is to apply the recursive formula repeatedly, until we reach the base of the recursion. For example, observe that (8.1) also gives  $T(n/2) \leq 2 \cdot T(n/4) + n/2$ . Thus, by applying (8.1)  $\ell$  times, we get

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + n \\ &\leq 2 \cdot (2 \cdot T(n/4) + n/2) + n = 4 \cdot T(n/4) + 2n \\ &\leq 4 \cdot (2 \cdot T(n/8) + n/4) + 2n = 8 \cdot T(n/8) + 3n \\ &\leq 8 \cdot (2 \cdot T(n/16) + n/8) + 3n = 16 \cdot T(n/16) + 4n \\ &\leq 16 \cdot (2 \cdot T(n/32) + n/16) + 4n = 32 \cdot T(n/32) + 5n \\ &\leq \dots \\ &\leq 2^\ell \cdot T(n/2^\ell) + \ell \cdot n. \end{aligned}$$

For  $\ell = k$ , we have  $2^\ell = 2^k = n$ , and hence  $n/2^\ell = 1$  and  $T(n/2^\ell) = 0$ , by (8.2). Thus, after  $k$  steps, we reach the recursion base, and we obtain

$$T(n) \leq 2^k \cdot T(n/2^k) + k \cdot n = 2^k \cdot 0 + k \cdot n = k \cdot n.$$

Since  $k = \log n$ , we again find that  $T(n) \leq n \log n$  and hence  $T(n) = O(n \log n)$ .

## 8.3.4 Quicksort

In Section 7.5, we implemented quicksort as follows:

```
def partition(a, l, r):
    pivot = a[l]
    m = l
    for i in range(l + 1, r):
        if a[i] < pivot:          # (*)
            swap(a, m + 1, i)
            m = m + 1
    swap(a, l, m)
    return m

def quicksort(a):
    def quicksort_help(l, r):
        if r - l <= 1:
            return
        m = partition(a, l, r)
        quicksort_help(l, m)
        quicksort_help(m + 1, r)
    quicksort_help(0, len(a))
```

The only place where we compare two elements of the input list  $a$  is in the line marked by  $(*)$  in `partition(·)`. Thus, our goal is to count the maximum number of times the line  $(*)$  is executed for any input list  $a$  of length  $n$ .

Let us first focus on a single invocation of the `partition(·)`-function, and let  $m = \text{end} - \text{start}$  be the size of the sublist under consideration. Then, the line  $(*)$  is executed exactly  $m - 1$  times, since the `for`-loop in `partition(·)` ranges from `start + 1` to `end`. Thus, for a single invocation of `partition(·)` with a sublist of size  $m$ , the number of executions of  $(*)$  is at most  $m$ .<sup>7</sup>

Now, let us turn to `quicksort_help(·)`. In general, the function `quicksort_help(·)` performs a call to `partition(·)` with a sublist of size  $n$ , plus two recursive calls to `quicksort_help(·)`. In contrast to merge sort, the sizes of the sublists in the recursion now depend on the choice of the pivot element. Thus, we get the following recurrence relation for the worst-case running time of quicksort:

$$T(n) \leq T(n_\ell) + T(n_r) + n, \quad (8.3)$$

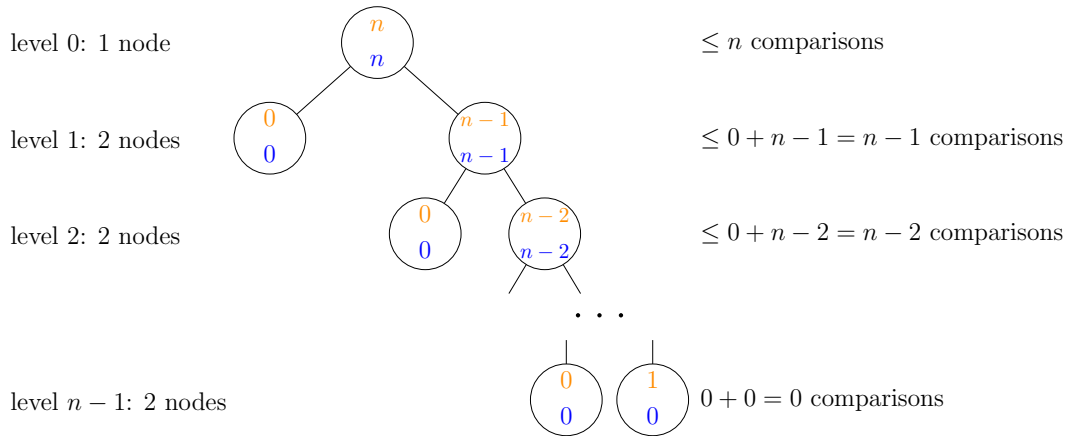
where  $n_\ell = m - 1$  is the size of the left sublist and  $n_r = r - m - 1$  is the size of the right sublist. The exact values of  $n_\ell$  and  $n_r$  depend on the pivot element. We only know that  $0 \leq n_\ell, n_r \leq n - 1$  and that  $n_\ell + n_r = n - 1$  (since the original list is partitioned into the left sublist, the right sublist, and the pivot element). In addition, we have

$$T(0) = T(1) = 0, \quad (8.4)$$

<sup>7</sup>As for merge sort, we simplify the bound from  $m - 1$  to  $m$ , because this does not change the asymptotic running times, but it makes the calculations easier.

since for a sublist with at most one element, `quicksort_help(·)` terminates immediately. Now, (8.3, 8.4) constitutes a recurrence relation for the worst-case running time of quicksort. As for merge sort, we discuss two methods for finding a solution.

**The tree method.** We want to visualize the recurrence relation (8.3, 8.4) in a tree diagram. For this, we need to decide which values for  $n_\ell$  and  $n_r$  should be used. Since we perform a worst-case analysis of quicksort, we focus on the worst possible choice of pivot element in each step. The worst possible pivot element is an element that does not split the current sublist at all. This happens if the pivot element is the smallest or the largest element of the current sublist. Thus, we will focus on the situation that the pivot element is the smallest element of the sublist. This leads to  $n_\ell = 0$  and  $n_r = n - 1$ . For this case, the tree diagram for (8.3, 8.4) looks as follows:



We see that the structure of the recursion is very regular. At the first level, we have exactly one node, at every other level, we have exactly two nodes. Of these two nodes, the left node has input size 0 and no children, and the right node has input size  $n - \ell$ , where  $\ell$  is the level of the node. Thus, the right node has two children if its input size is at least 2, and no children if its input size is 1. The levels go from 0 to  $n - 1$ , and the number of levels is exactly  $n$ . At the last level, the total number of comparisons is 0, at all the other levels, the total number of comparisons is at most  $n - \ell$ , where  $\ell$  is the level. Thus, we can compute the worst-case running time of quicksort as

$$T(n) \leq n + (n - 1) + \cdots + 2 + 0 = \sum_{i=1}^n i - 1 = \frac{n(n + 1)}{2} - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1.$$

Omitting the constant factor  $1/2$  and the lower-order additive term  $n/2 - 1$ , we conclude that the asymptotic worst-case running time of quicksort is  $O(n^2)$ .

**Repeated application of the recurrence relation.** We apply the recurrence relation repeatedly, until we reach the base of the recursion. Again, we focus on the case that  $n_\ell = 0$  and  $n_r = n - 1$ . Then, by applying (8.3)  $\ell$  times, and by using (8.4), we get

$$T(n) \leq T(0) + T(n - 1) + n = T(n - 1) + n$$

$$\begin{aligned}
 &\leq (T(0) + T(n-2) + n-1) + n = T(n-2) + (n-1) + n \\
 &\leq (T(0) + T(n-3) + n-2) + (n-1) + n = T(n-3) + (n-2) + (n-1) + n \\
 &\leq \dots \\
 &\leq T(n-\ell) + (n-\ell+1) + \dots + (n-2) + (n-1) + n.
 \end{aligned}$$

For  $\ell = n-1$ , we have  $n-\ell = 1$  and hence  $T(n-\ell) = 0$ , by (8.4). Thus, after  $n-1$  steps, we reach the recursion base, and we obtain

$$T(n) \leq T(n-(n-1)) + 2 + 3 + \dots + (n-1) + n = 2 + 3 + \dots + (n-1) + n = \frac{1}{2}n^2 + \frac{1}{2}n - 1,$$

as before. Thus, we again find that the asymptotic worst-case running time of quicksort is  $O(n^2)$ .

This result may seem disappointing. If the asymptotic worst-case running time of quicksort is not better than the asymptotic worst-case running time of selection sort, why would we even bother? The answer is that in the case of quicksort, the worst case analysis does not really represent the “typical case”. In practice, for typical input lists, quicksort performs much better. To support this fact theoretically, we need to perform a more precise analysis of quicksort, called *average-case analysis*.<sup>8</sup> The average-case analysis shows that for “typical” inputs, quicksort has asymptotic running time  $O(n \log n)$ . Actually, in this case, quicksort is often preferable over merge sort, because quicksort has a smaller overhead compared to merge sort (merge sort needs to create two new lists in each recursive invocation, while quicksort does everything inside the input list  $a$ ). Another way to improve this result is to find a better way to choose the pivot element. A very simple way is *randomized quicksort*: simply pick a random element of the current sublist as the pivot element. With this change, quicksort can also be shown to have asymptotic (expected) running time  $O(n \log n)$ .

## 8.4 Complexity of algorithmic problems

In Section 8.3, we successfully obtained worst-case running times for different sorting algorithms. The conclusion is that among the sorting algorithms from Chapter 7, merge sort is the fastest, since the asymptotic worst-case running time of merge sort is  $O(n \log n)$ . However, at this point, we cannot be sure that this is best possible. Using some simple observations and a more advanced technique (i.e., divide-and-conquer), we managed to improve the simple sorting algorithms selection sort and insertion sort to the much faster algorithm merge sort. Maybe there are even better techniques that lead to an even faster sorting method? This question raises the issue of the *complexity of algorithmic problems*. Here the idea is to fix an algorithmic problem  $P$ , and to consider *all* possible algorithms that solve  $P$ . Then, we are interested in *upper bounds* and *lower bounds* for the complexity of  $P$ .

---

<sup>8</sup>This is taught in later classes.

- **Upper bounds for the complexity of  $P$ .** If there is an algorithm that solves  $P$  in worst-case running time  $T(n)$ , then  $T(n)$  is called an *upper bound* for the complexity of the algorithmic problem  $P$ . Thus, an upper bound shows how “easy” an algorithmic problem is. Here are some examples:
  - **Searching in an unsorted list.** In Section 8.1.1, we saw that linear search solves this algorithmic problem with worst-case running time  $T(n) = n$ . Hence,  $T(n) = n$  is an upper bound for the algorithmic problem “searching in an unsorted list”.
  - **Sorting.** In Section 8.3.1, we saw that selection sort solves this algorithmic problem with worst-case running time  $T(n) = O(n^2)$ . Hence,  $T(n) = O(n^2)$  is an upper bound for the algorithmic problem “sorting”. However, in Section 8.3.3, we saw that merge sort solves the sorting problem with worst-case running time  $T(n) = O(n \log n)$ . Thus,  $T(n) = O(n \log n)$  is another (better) upper bound for the algorithmic problem “sorting”.
- **Lower bounds for the complexity of  $P$ .** Let  $f(n)$  be a function such that the following holds: *for every* algorithm  $A$  that solves  $P$  and *for every* input size  $n$ , there is *at least* one input  $I$  of size  $n$  such that  $A$  needs *at least*  $f(n)$  elementary operations on input  $I$ . Then,  $f(n)$  is called a *lower bound* for the complexity of the algorithmic problem  $P$ . Thus, a lower bound shows how “hard” an algorithmic problem is. Here are some examples:
  - **Searching in an unsorted list.** In Section 6.2.1, we argued that every algorithm that solves this algorithmic problem needs to perform at least  $n$  comparisons in order to be sure that an input list  $a$  with  $n$  elements does not contain the search key  $k$ . Thus, for every algorithm  $A$  for “searching in an unsorted list” and for every input size  $n$ , there is an input  $I$  of size  $n$  such that  $A$  needs at least  $n$  elementary operations on  $I$  (the input  $I$  consists of an arbitrary list  $a$  with  $n$  elements and a search key  $k$  that does not appear in  $a$ ). It follows that  $f(n) = n$  is a lower bound for the algorithmic problem “searching in an unsorted list”.
  - **Sorting.** Let  $A$  be an algorithm that solves the sorting problem, and let  $a$  be an input list with  $n$  elements. Then, if  $A$  is run on input  $a$ ,  $A$  must use every element of  $a$  in at least one comparison. Otherwise,  $A$  would not be correct. Thus, for every algorithm  $A$  for the sorting problem and for every input size  $n$ , there is an input  $I$  of size  $n$  such that  $A$  needs at least  $n/2$  elementary operations on  $I$  (the input  $I$  consists of an arbitrary list  $a$  with  $n$ ). It follows that  $f(n) = n/2$  is a lower bound for the algorithmic problem “sorting”. However, between  $n/2$  and  $O(n \log n)$ , there is still a significant gap that we would like to close.

**Attention:** It is relatively easy to find an upper bound for an algorithmic problem. It suffices to present *one* algorithm whose running time fits the desired upper bound. In

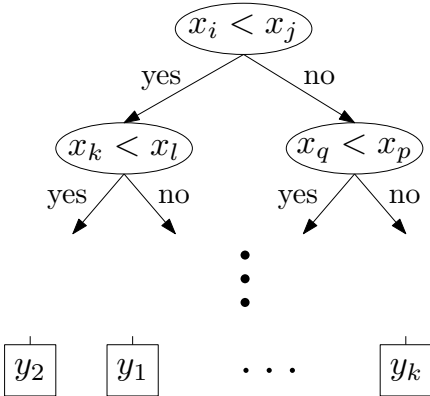


contrast, it is usually much harder to give a lower bound for an algorithmic problem. We must argue about the running times of *all* possible algorithms that solve the algorithmic problem– including those algorithms that we do not know yet. We will talk much more about this issue in “*Grundlagen der theoretischen Informatik*”.

### 8.4.1 Decision trees

Our objective is to find a better lower bound for the sorting problem in the model where the elementary operations are comparisons between the elements from the input list. This is often called the *comparison-based model* for sorting. To derive our lower bound, we need the notion of *decision trees*.

Decision trees are defined as follows: let  $n$  be an input size, and let  $X = [x_1, \dots, x_n]$  be an input sequence that consists of  $n$  elements from a totally ordered universe (i.e., we can perform comparisons on the elements of  $X$ ). Then, a general decision tree for  $X$  might look like this:

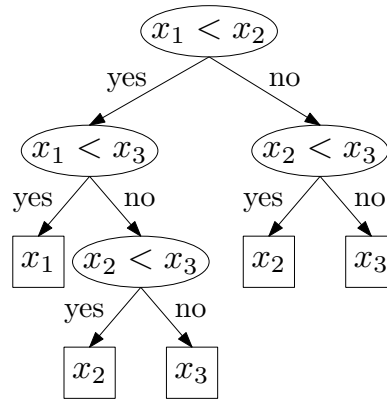


More precisely, a decision tree consists of two kinds of *nodes*: *inner nodes* and *leaf nodes*. Every inner node is labeled with a comparison between two elements of  $X$  and has two outgoing edges, one labeled “yes” and one labeled “no”. The two outgoing edges go to other nodes of the decision tree, the *children* of the corresponding inner node. Every leaf node is labeled with a *result* for the decision tree. The highest node of the decision tree is called the *root*.

Given a concrete set of values for the input sequence  $X$ , we can *execute* the decision tree on these values as follows: we start at the root and perform the corresponding comparison. If the comparison holds, we follow the outgoing edge to the “yes”-child; if the comparison does not hold, we follow the outgoing edge to the “no”-child. We continue in this manner, performing the comparisons and following the corresponding edges, until we reach a leaf. Then, the label of the leaf node is the *result* of the execution on the given input values.

For example, the following decision tree computes the *minimum* element of an input sequence  $X = [x_1, x_2, x_3]$  with three elements:





The *height* of a decision tree is the maximum number of edges between the root and a leaf. In the example above, the decision tree has height 3, since a longest path from the root “ $x_1 < x_2$ ” to a leaf (for example to the leaf labeled  $x_2$ ) has three edges. The following lemma will be crucial for our lower bound:

**Lemma 8.1.** *Let  $T$  be a decision tree. Then, the following holds: if  $T$  has at least  $m$  leaves, then  $T$  has height at least  $\log m$ .*

*Beweis.* The statement of the lemma is a direct contrapositive of the following statement: “if  $T$  has height less than  $h$ , then  $T$  has less than  $2^h$  leaves”. This statement follows directly from the following two observations: (i) if the height of  $T$  is  $h = 0$ , then  $T$  has only  $2^0 = 1$  leaf (the root is also a leaf); and (ii) if we go from height  $h$  to height  $h + 1$ , the maximum number of leaves can increase at most by a factor of two (because an inner node has exactly two children).  $\square$

**Attention:** Unlike the algorithms that we have seen so far, decision trees are always defined for a *fixed* input size  $n$ . Thus, if we want to solve an arbitrary algorithmic problem with decision trees, we need to describe one decision tree for each possible input size.



### 8.4.2 A lower bound for comparison-based sorting

Using the notion of decision trees, we can now derive a general lower bound for the sorting problem in the comparison-based model.

**Satz 8.2.** *Let  $A$  be comparison-based sorting algorithm, and let  $n$  be an input size. Then, there exists an input  $I$  such that  $A$  needs at least  $\frac{n}{2}(\log n - 1)$  comparisons on  $I$ .*

*Beweis.* First, we observe that the execution of  $A$  on an input  $I$  of size  $n$  can be modeled by a decision tree  $T$  for input sequences of size  $n$ : we start to execute the algorithm  $A$  on a generic input  $I$  of size  $n$ . Then, at some point,  $A$  will perform a first comparison between two elements of  $I$ . Before this comparison, the behavior of  $A$  will be the same



on all inputs of size  $n$ . After this, the next steps of  $A$  will be the same for all inputs of size  $n$  where the first comparison yields the same result. Then, at some point,  $A$  will perform a second comparison between two elements of  $I$ . Again,  $A$  will behave in the same way for all inputs where the first and the second comparison yield the same result, and so on, for all further comparisons. In the end,  $A$  will output some sorting permutation for  $I$ , where the sorting permutation only depends on the results of the comparisons. Thus, the behavior of  $A$  on inputs of size  $n$  can be modeled as a (huge) decision tree  $T$  whose leaves are labeled with all the possible sorting permutations and whose inner nodes correspond to comparisons between the elements of the input list.

Now, for every leaf of the decision tree  $T$ , there is at least one possible input  $I$  of size  $n$  that leads to this leaf. Thus, the worst-case running time of  $A$  for input size  $n$  corresponds exactly to the height of  $T$ . We now want to use Lemma 8.1 to provide a lower bound on the height of  $T$  and thus on the worst-case running time of  $A$ .

For this, we observe that  $T$  must have at least  $n!$  leaves, because there are  $n!$  possible sorting permutations for input lists of size  $n$ . Thus, Lemma 8.1 shows that  $T$  has height at least  $\log(n!)$ . It remains to find an explicit lower bound for  $\log(n!)$ . For this, we can estimate

$$\begin{aligned}
 n! &= n \cdot (n-1) \cdot \dots \cdot \underbrace{\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \dots \cdot 2 \cdot 1}_{\lceil n/2 \rceil \text{ factors}} \\
 &\geq \underbrace{\left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \dots \cdot \left\lfloor \frac{n}{2} \right\rfloor}_{\lceil n/2 \rceil \text{ factors}} \cdot \underbrace{1 \cdot \dots \cdot 1 \cdot 1}_{\lfloor n/2 \rfloor \text{ factors}} \\
 &= \left\lfloor \frac{n}{2} \right\rfloor^{\lceil n/2 \rceil} \\
 &\geq \left(\frac{n}{2}\right)^{n/2}.
 \end{aligned}$$

Hence, we have

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2} (\log n - \log 2) = \frac{n}{2} (\log n - 1).$$

The theorem follows. □

Omitting the constant factor  $1/2$  and the lower-order additive term  $-n/2$ , Theorem 8.2 shows that the sorting problem has the asymptotic lower bound  $n \log n$ . Since this matches the asymptotic worst-case upper bound that is given by merge sort, we can conclude that merge sort is an *asymptotically optimal* algorithm for the sorting problem in the comparison-based model.

**Attention:** There are sorting algorithms with an asymptotic worst-case running time that is better than  $O(n \log n)$ , e.g., radix sort, counting sort, or bucket sort. However, these algorithms are not comparison-based. They use other techniques and rely on additional assumptions on the input elements (e.g., that the input elements are small integers and can be used as indices in a list).



#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP2** Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP8** Du analysierst *Algorithmen*, indem du *Korrektheit* (auf Grundlage der Spezifikation), *Speicherplatz* und *Laufzeit* angibst und begründest.



### 9.1 Fundamentals of correctness

In this chapter, we address another crucial property of algorithms: *correctness*. Correctness means that an algorithm does exactly what it is supposed to do. Formal proofs of correctness for algorithms play a key role in many areas of Computer Science, e.g., in information security, algorithms design, and software development.

First, we must discuss what exactly we mean by saying that an algorithm is “correct”. Formally, we say that an algorithm is *correct* if it *fulfills* a given *specification*. A specification consists of two logical conditions: a *precondition* and a *postcondition*. An algorithm *fulfills* a specification if for all inputs that fulfill the precondition, the algorithm produces an output that fulfills the postcondition. Thus, the notion of “correctness of an algorithm” is always relative to a given specification. Our first task in proving an algorithm correct lies in finding a useful specification that captures the algorithmic problem that we are interested in. For example, the following specification is not very useful:

```
# Precondition: None
# Postcondition:
```



```
# Effect: Something is printed on the screen
# Result: Something is returned
def foo():
    print("Hello")
    return 42
```



Even though the function `foo(·)` fulfils the specification, this information is useless. The underlying specification is meaningless. A useful specification would be:

```
# Precondition: None
# Postcondition:
#   Effect: "Hello" is printed on the screen
#   Result: 42 is returned
def foo():
    print("Hallo")
    return 42
```



Now, we can see that `foo(·)` fulfills a useful specification.

As another example, consider the following specification for the algorithmic problem “searching in a sorted list”:

```
# Input: a: List[U], k: U, U is totally ordered universe
# Precondition: a is sorted in ascending order
# Postcondition:
#   Effect: None
#   a[result] == k, if k appears in a
#   result == None, otherwise
```



Then, both binary search and linear search fulfill this specification. On the other hand, consider the following specification for the algorithmic problem “searching in an unsorted list”:

```
# Input: a: List[U], k: U, U is a totally ordered universe
# Precondition: None
# Postcondition:
#   Effect: None
#   a[result] == k, if k appears in a
#   result == None, otherwise
```



Then, linear search fulfills this specification, but binary search does not. Thus, we see that “correctness” is not an intrinsic property of an algorithm, but very much depends on the given specification.

Now, suppose that we have a useful specification  $S$  and a program  $P$  that implements a candidate algorithm for  $S$ . How can we determine whether  $P$  fulfills the specification  $S$  or not? Essentially, there are two possibilities: we can *test*  $P$  against  $S$ , or we can provide a *formal proof* that  $P$  fulfills  $S$ .

**Program testing.** To test a program  $P$  against a specification  $S$ , we simply run  $P$  on several carefully selected inputs that satisfy the precondition. Then, we check whether the resulting outputs meet the postcondition (this test can be automatized). The main challenge lies in finding appropriate test inputs that make it likely that possible flaws in the program are discovered. There is a whole subarea of software engineering that is dedicated to this question.

However, there is still an issue: most programs can have infinitely many possible inputs (e.g., this is the case for programs that implement sorting algorithms). Thus, if we really want to be sure that our program *always* works, we would need to test it with infinitely many inputs—this is clearly not feasible. Therefore, testing is a useful tool to find errors, but it will usually not prove the correctness of an algorithm.

“Program testing can be used to show the presence of bugs, but never to show their absence!” – Edsger W. Dijkstra

“Beware of bugs in the above code; I have only proved it correct, not tried it.” — Donald E. Knuth

**Formal proof of correctness.** If we really want to be sure that our program is correct, we need to give a formal mathematical proof of correctness. Such proofs can be quite cumbersome, but they are able to show definitely the absence of mistakes. It is possible to give completely formal correctness proofs that can be checked by a computer (e.g., using *Hoare logic*). In this chapter, we will follow a more relaxed approach. However, the main ideas are the same. We will look at three scenarios:

1. Correctness proofs for simple linear programs,
2. Correctness proofs for programs with loops, and
3. Correctness proofs for programs with recursion.

## 9.2 Correctness of simple linear programs

First, we look at simple “linear” programs that do not have loops or recursion. Here, we can often use direct mathematical arguments and a careful analysis of the code to obtain our desired correctness proof. As an example, consider the following program. It is supposed to calculate the *absolute value* of a given input number.

```
# abs(x: float): float
# Precondition: None
# Postcondition:
#   Effect: None
#   Result: result == |x|
'''Tests: Your task ;-) '''
def abs(x):
```



```
# Return x or -x, whichever is larger.
if -x > x:
    return -x
else:
    return x
```



Before we can prove the correctness of `abs(·)`, we first need the mathematical definition of the absolute value  $|x|$ :

$$|x| = \begin{cases} -x, & \text{if } x < 0, \text{ and} \\ x, & \text{otherwise.} \end{cases} \quad (9.1)$$

Now, we can argue as follows: let  $x$  be an input number that satisfies the precondition. The precondition is empty, so  $x$  is an arbitrary real point number. We distinguish two cases:

- **Case 1:**  $x < 0$ : In this case, we have  $-x > 0$ , and hence  $-x > 0 > x$ . Thus, the `if`-instruction in `abs(·)` will execute the first branch, and the result of `abs(·)` is  $-x$ . This is exactly what (9.1) requires.
- **Case 2:**  $x \geq 0$ : In this case, we have  $-x \leq 0$ , and hence  $-x \leq 0 \leq x$ . Thus, the `if`-instruction in `abs(·)` will execute the second branch, and the result of `abs(·)` is  $x$ . Again, this is what (9.1) states.

In conclusion, we have shown that `abs(·)` computes the absolute value  $|x|$  correctly.

### 9.3 Correctness of programs with loops

We now turn to programs with loops. We will focus on `while`-loops, since every `for`-loop can be implemented as a `while`-loop. As we saw in Section 2.4.3, the general structure of a `while`-loop is as follows:

```
while C:
    B
```



Here,  $C$  is a Boolean expression and  $B$  is a block of instructions. We call  $C$  the *loop condition* and  $B$  the *loop body* of the `while`-loop. Now, how can we show that a `while`-loop fulfills a given specification? The main issue is that the loop body of a `while`-loop is not executed only once, but again and again, with different settings of the variables. Often, we cannot even say how often the loop body is executed. Thus, it is not enough to argue about a *single* iteration of the `while`-loop, but we need to find a way to argue about *all possible iterations at once*.

The solution is to find a property that *all* the iterations of the `while`-loop have in common. Such a property is called the *invariant* of the `while`-loop. There are many possible invariants, but we need to find an invariant that is *useful*, in the sense that the

invariant follows from the precondition and implies the postcondition. In some sense, the invariant captures the intention that the **while**-loop is supposed to satisfy.

Using an appropriate invariant, we can show that a **while**-loop is *partially correct*: if the **while**-loop finishes, then the postcondition is fulfilled. However, a **while**-loop could iterate forever. Thus, we also need to show *termination*: the **while**-loop will finish after a finite number of iterations. If we have shown partial correctness and termination for a **while**-loop, then we say that the **while**-loop is *totally correct*.

Formally, let  $W$  be a **while**-loop with loop condition  $C$  and loop body  $B$ . To show that  $W$  is *totally correct*, we need the following steps:

1. **Show partial correctness.** For this, we need to find a suitable *invariant* for  $W$ . An invariant  $Inv$  of  $W$  is a logical condition that satisfies the following properties:
  - **Initialization.** The invariant  $Inv$  holds *right before*  $W$  (i.e., before the first time that the loop condition  $C$  is evaluated).
  - **Maintenance.** Assume that the invariant  $Inv$  and the loop condition  $C$  hold together *before* the loop body  $B$  is executed. Then, *after* the loop body  $B$  is executed, the invariant  $Inv$  holds again, for the new settings of the variables (as effected by  $B$ ).
  - **Conclusion.** Assume that the invariant  $Inv$  and the negation of the loop condition, **not**  $C$ , hold together. Then, the postcondition of the specification can be derived.

Finding a suitable invariant is not trivial, and it requires practice and experience. Once we have found a candidate invariant  $Inv$ , we need to state it explicitly and to argue that it satisfies the three properties “Initialization”, “Maintenance”, and “Conclusion”. For this, we can use direct mathematical arguments.

“Initialization” and “Maintenance” together say that  $Inv$  is a common property for *all* iterations of  $W$ .<sup>1</sup> To show “Initialization”, we can use the precondition of the specification. “Conclusion” shows that  $Inv$  is useful, in the sense that it can be used to show that after  $W$  finishes, the postcondition is fulfilled.

2. **Show termination.** To show that  $W$  eventually terminates, we need to find some measure of *progress* over the loop iterations, and to show that this progress implies that eventually, the loop will end.

Admittedly, this prescription is very vague. In the examples that we discuss here, it will be very clear what to do. In general, however, this vagueness is no coincidence: in “*Grundlagen der Theoretischen Informatik*”, you will learn that it is *undecidable* whether a given **while**-loop  $W$  terminates. This means that there is no general method that works for *all* **while**-loops, but every new **while**-loop poses a new challenge to our human creativity.

We will now discuss two examples.

<sup>1</sup>If you are familiar with *mathematical induction*, then you might see some similarities. This is no coincidence. Formally, to show that the invariant holds for all iterations of  $W$ , we must apply a mathematical induction on the number of loop iterations.

**First example: finding the minimum.** Let us consider the example of finding the index of the minimum element in an list input list  $a$ .

```
# minimum(a: List[U]): int, U is a totally ordered universe
# Precondition: len(a) > 0
# Postcondition:
#   Effect: None
#   Result: The index of the smallest element in a is returned.
def minimum(a):
    m = 0
    i = 1
    while i < len(a):
        if a[i] < a[m]:
            m = i
            i = i + 1
    return m
```

Now, we would like to show that `minimum(·)` is correct. For this, we go through the steps:

1. **Show partial correctness.** We need to find a suitable invariant  $Inv$  for the **while**-loop in `minimum(·)`. For this, we need to determine what all the iterations of the **while**-loop have in common. In this case, the answer is easy: in each iteration, the index  $m$  identifies the minimum element among the first  $i$  elements of  $a$ . Formally, we will use the following invariant:

$$Inv : i \leq \text{len}(a) \wedge m = \text{argmin}_{j=0, \dots, i-1} a[j].$$

Now, we check that  $Inv$  satisfies the three properties:

- **Initialization.** Right before the **while**-loop, we have  $i = 1$  and  $m = 0$ . The precondition implies that  $\text{len}(a) \geq 1$ , so we have  $i \leq \text{len}(a)$ . The precondition also implies that  $a[0]$  exists, and hence  $m = \text{argmin}_{j=0, \dots, 0} a[j]$ , since  $a[0]$  is the minimum of the 1-element sublist  $a[0:1]$ . Thus, it follows that  $Inv$  holds right before the **while**-loop.
- **Maintenance.** Assume that the loop condition  $i < \text{len}(a)$  and  $Inv$  both hold. That is, we have

$$i < \text{len}(a) \wedge i \leq \text{len}(a) \wedge m = \text{argmin}_{j=0, \dots, i-1} a[j]. \quad (9.2)$$

More succinctly, we can write (9.2) as

$$i < \text{len}(a) \wedge m = \text{argmin}_{j=0, \dots, i-1} a[j], \quad (9.3)$$

since the condition  $i < \text{len}(a)$  is stronger than the condition  $i \leq \text{len}(a)$ . Now, we execute the loop body  $B$ . Let  $i'$  and  $m'$  be the resulting values of  $i$  and  $m$  (and we use  $i$  and  $m$  to denote the values *before*  $B$  was executed). Then, we have to show that  $Inv$  holds for  $i'$  and  $m'$ . That is, we need to show that

$$i' \leq \text{len}(a) \wedge m' = \text{argmin}_{j=0, \dots, i'-1} a[j]. \quad (9.4)$$



Looking at the loop body, we see that  $i' = i + 1$ , and hence (9.4) is equivalent to

$$i + 1 \leq \text{len}(a) \wedge m' = \text{argmin}_{j=0, \dots, i} a[j]. \quad (9.5)$$

Now, the first part of (9.5) follows immediately from (9.3): (9.3) states that  $i < \text{len}(a)$  and  $i$  is an integer, so we have  $i + 1 \leq \text{len}(a)$ . For the second part, we distinguish two cases:

- **Case 1:  $a[i] \geq a[m]$ .** In this case, the **if**-block in the **while**-loop is not executed, and we have  $m' = m$ . Furthermore, since  $a[i] \geq a[m]$ , the second part of (9.3) gives

$$m = \text{argmin}_{j=0, \dots, i} a[j],$$

and the second part of (9.5) follows.

- **Case 2:  $a[i] < a[m]$ .** In this case, the **if**-block in the **while**-loop is executed, and we have  $m' = i$ . Furthermore, since  $a[i] < a[m]$ , the second part of (9.3) and the transitivity of  $<$  give

$$i = \text{argmin}_{j=0, \dots, i} a[j],$$

and the second part of (9.5) follows.

Thus, we have shown that (9.5) holds, and hence *Inv* is maintained by the **while**-loop.

**Attention:** The invariant *Inv* has to hold only at the beginning and at the end of the loop body. During the execution of the loop body, the invariant might temporarily be false.



- **Conclusion.** Assume that the negation of the loop condition,  $i \geq \text{len}(a)$ , and *Inv* both hold. That is, we have

$$i \geq \text{len}(a) \wedge i \leq \text{len}(a) \wedge m = \text{argmin}_{j=0, \dots, i-1} a[j]. \quad (9.6)$$

More succinctly, we can write (9.6) as

$$i = \text{len}(a) \wedge m = \text{argmin}_{j=0, \dots, i-1} a[j], \quad (9.7)$$

since the conditions  $i \geq \text{len}(a)$  and  $i \leq \text{len}(a)$  imply that  $i = \text{len}(a)$ . Now, (9.7) implies that

$$m = \text{argmin}_{j=0, \dots, \text{len}(a)-1} a[j].$$

This says that  $m$  is the index of a minimum element in  $a$ . Thus, we have shown that once the **while**-loop finishes, the invariant *Inv* implies the postcondition.

2. **Show termination.** To show termination, we need to identify some kind of progress in the **while**-loop. For this, we consider the variable  $i$ . The variable  $i$  is initialized to 1, and in each iteration of the **while**-loop, it is increased by the *fixed* amount 1. The **while**-loop finishes once  $i$  reaches the *fixed* value  $\text{len}(a)$ . Thus, the **while**-loop will end after a finite number of steps.

Now, we have completed all the necessary steps, and we have shown that `minimum(.)` is totally correct.

**Second example: computing the binary representation.** As a second example, we consider the computation of the binary representation of a given natural number.

```
# dec2bin(n: int): List[int]
# Precondition: n > 0
# Postcondition:
#   Effect: none
#   Result: a list with the binary representation of n is returned,
#           going from the lowest order to the highest order bit, i.e.
#           n = sum_(j=0)^(len(s)-1) b[j]*(2**j)
def dec2bin(n):
    n1 = n
    b = []
    while n1 > 0:
        b.append(n1 % 2)
        n1 = n1 // 2
    return b
```

Let us once again state the postcondition. Let  $b$  be the output list of `dec2bin(.)`. Then, the postcondition states

$$n = \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j. \quad (9.8)$$

Now, we would like to show that `dec2bin(.)` is correct. For this, we go through the steps:

1. **Show partial correctness.** We need to find a suitable invariant  $\text{Inv}$  for the **while**-loop in `dec2bin(.)`. For this, we need to determine what all the iterations of the **while**-loop have in common. In this case, the answer is as follows: in each iteration, the list  $b$  contains a *partial* binary representation of the lower order bits of  $n$ , while  $n1$  contains the remaining bits, shifted to the right. Formally, we will use the following invariant:

$$\text{Inv} : n1 \geq 0 \wedge n = n1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j.$$

Now, we check that  $\text{Inv}$  satisfies the three properties:

- **Initialization.** Right before the **while**-loop, we have  $n_1 = n$  and  $b = []$ . The precondition implies that  $n_1 \geq 0$ . Furthermore, since  $\text{len}(b) = 0$ , we also have

$$n = n_1 \cdot 2^0 = n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j,$$

since the sum over  $j$  is empty. Thus, it follows that Inv holds right before the **while**-loop.

- **Maintenance.** Assume that the loop condition  $n_1 > 0$  and Inv both hold. That is, we have

$$n_1 > 0 \wedge n_1 \geq 0 \wedge n = n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j. \quad (9.9)$$

More succinctly, we can write (9.9) as

$$n_1 > 0 \wedge n = n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j. \quad (9.10)$$

since the condition  $n_1 > 0$  is stronger than the condition  $n_1 \geq 0$ . Now, we execute the loop body B. Let  $n_1'$  and  $b'$  be the resulting values of  $n_1$  and  $b$  (and we use  $n_1$  and  $b$  to denote the values *before* B was executed). Then, we have to show that Inv holds for  $n_1'$  and  $b'$ . That is, we need to show that

$$n_1' \geq 0 \wedge n = n_1' \cdot 2^{\text{len}(b')} + \sum_{j=0}^{\text{len}(b') - 1} b'[j] \cdot 2^j. \quad (9.11)$$

Looking at B, we see that  $n_1' = n_1 // 2$  and that  $b' = b.append(n_1 \% 2)$ . Now, the first part of (9.11) follows immediately from (9.10): (9.10) states that  $n_1 > 0$  and  $n_1$  is an integer, so we have  $n_1' = n_1 // 2 \geq 0$ . For the second part, we first make the following observations: through the assignments in the loop body, we have

$$n_1 = 2 \cdot (n_1 // 2) + (n_1 \% 2) = 2 \cdot n_1' + (n_1 \% 2),$$

and

$$\text{len}(b') = \text{len}(b) + 1,$$

and

$$b'[\text{len}(b')] - 1] = n_1 \% 2,$$

and

$$b'[j] = b[j], \text{ for } j = 0, \dots, \text{len}(b) - 1.$$

Thus, we can start with the second part of (9.10), and we calculate

$$\begin{aligned}
 n &= n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j \\
 &= (2 \cdot n_1' + (n_1 \% 2)) \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b') - 2} b'[j] \cdot 2^j \\
 &= n_1' \cdot 2^{\text{len}(b')} + b'[\text{len}(b') - 1] \cdot 2^{\text{len}(b') - 1} + \sum_{j=0}^{\text{len}(b') - 2} b'[j] \cdot 2^j \\
 &= n_1' \cdot 2^{\text{len}(b')} + \sum_{j=0}^{\text{len}(b') - 1} b'[j] \cdot 2^j.
 \end{aligned}$$

This is exactly the second part of (9.11). Thus, we have shown that (9.11) holds, and hence *Inv* is maintained by the **while**-loop.

- **Conclusion.** Assume that the negation of the loop condition,  $n_1 \leq 0$ , and *Inv* both hold. That is, we have

$$n_1 \leq 0 \wedge n_1 \geq 0 \wedge n = n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j. \quad (9.12)$$

More succinctly, we can write (9.12) as

$$n_1 = 0 \wedge n = n_1 \cdot 2^{\text{len}(b)} + \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j. \quad (9.13)$$

since the conditions  $n_1 \leq 0$  and  $n_1 \geq 0$  imply that  $n_1 = 0$ . Now, (9.13) implies that

$$n = \sum_{j=0}^{\text{len}(b) - 1} b[j] \cdot 2^j.$$

This is exactly the desired postcondition (9.8) which says that *b* is the binary representation of *n*. Thus, we have shown that once the **while**-loop finishes, the invariant *Inv* implies the postcondition.

2. **Show termination.** To show termination, we need to identify some kind of progress in the **while**-loop. For this, we consider the variable  $n_1$ . The variable  $n_1$  is initialized to *n*, and in each iteration of the **while**-loop, we perform an *integer* division by the *fixed* value 2. The **while**-loop finishes once  $n_1$  reaches the *fixed* value 0. Thus, the **while**-loop will end after a finite number of steps.

**Attention:** It is important that we perform an *integer* division in each iteration. Otherwise, it might happen that we keep dividing  $n_1$  by 2, without ever reaching 0. This is exactly what happens in the famous story of Achilles and the tortoise.



Now, we have completed all the necessary steps, and we have shown that `dec2bin(.)` is totally correct.

## 9.4 Correctness of programs with recursion

In Chapters 6 and 7, we have often used recursion to solve algorithmic problems. Thus, we need a way to prove that programs with recursion are correct. A strategy for such proofs is already immediate from the idea of recursion: as discussed in Section 4.3, a recursive program has two components: the *base of the recursion* and the *recursion step*. The base of the recursion is supposed to solve small inputs directly; the recursion step solves larger inputs by using recursive calls to process smaller inputs. Thus, to show that a recursive program is correct, we need to do two things: (i) we need to verify that the base cases are correct; and (ii) we need to show that the recursion steps are correct, *assuming that the recursive calls fulfill the specification*. If we have done this, we know that the recursive program is *partially correct*: if the recursion terminates, then the result satisfies the postcondition.

As for loops, we also need to argue *termination*: for every possible input, the recursion will eventually reach a base case. If we have argued that both partial correctness and termination hold, then we say that the recursive program is *totally correct*.

Formally, to show that a recursive program is correct, we can use the following steps:

1. **Show partial correctness.** For partial correctness, we need to consider two things.
  - **Base cases.** We need to argue that the program is correct for the bases cases of the recursion.
  - **Recursion steps.** We need to argue the following: suppose that the recursive calls inside the program are correct. Then, the entire program yields a result that satisfies the postcondition.<sup>2</sup>
2. **Show termination.** We need to argue that for any input, the recursion will reach a base case after a finite number of steps.

We will now discuss three examples.

**First example: computing the factorial.** As a first example, we look at a recursive program to compute the factorial of a given natural number  $n$ .

<sup>2</sup>Again, this is very similar to *mathematical induction*, and indeed, formally, we use mathematical induction on the number of the recursive calls to show that a recursive algorithm is correct.

```
# fac(n: int): int
# Precondition: n >= 0
# Postcondition:
#   Effect: None
#   Result: n! is returned
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)
```



We now want to show the correctness of `fac(·)`. For this, we recall the mathematical definition of the factorial function: we have

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n,$$

with the special case  $0! = 1$ . Now, we go through the steps:

### 1. Show partial correctness.

- **Base cases.** The only base case of the recursion occurs for  $n = 0$ . For this input, `fac(·)` returns `1`. This is correct, since  $0! = 1$ .
- **Recursion steps.** By the precondition and the `if`-statement in `fac(·)`, the recursion step occurs for  $n > 0$ . Since  $n > 0$ , we have  $n - 1 \geq 0$ , and hence the recursive call `fac(n - 1)` satisfies the precondition. Thus, we can assume that the recursive call computes the correct result, i.e., that `fac(n - 1)` correctly returns  $(n - 1)!$ . Then, we can conclude:

$$\text{fac}(n) = n \cdot \text{fac}(n - 1) = n \cdot (n - 1)! = n!.$$

Hence, the recursion step is correct.

Since both the base case and the recursion step are correct, the program is partially correct.

- ### 2. Show termination.
- To show termination, we need to identify some progress throughout the recursion. For this, we consider the parameter  $n$ . The precondition states that  $n \geq 0$ , and the recursion stops when  $n = 0$ . Furthermore, in each recursive call,  $n$  is decreased by 1. Thus, if  $n$  satisfies the precondition, then the recursion will finish after a finite number of recursive calls.

Now, we have completed all the necessary steps, and we have shown that `fac(·)` is totally correct.

**Second example: computing the minimum.** We again want to prove the correctness of a program that computes the index of the minimum element in an input list `a`. This time, the algorithm is recursive.



```
# minimum(a: List[U]): int, U is a totally ordered universe
# Precondition: len(a) > 0
# Postcondition:
#   Effect: None
#   Result: The index of the smallest element in a is returned.
def minimum(a):
    n = len(a)
    if n == 1:
        return 0
    else:
        m = minimum(a[:-1])
        if a[m] <= a[n - 1]:
            return m
        else:
            return n - 1
```

We would like to show that `minimum(·)` is correct. For this, we go through the steps.

### 1. Show partial correctness.

- **Base cases.** The only base case of the recursion occurs for  $n = \text{len}(a) = 0$ . For this input, `minimum(·)` returns 0. This is correct, since if  $a$  consists of only one element, then the minimum of  $a$  is exactly this element, namely  $a[0]$ .
- **Recursion steps.** By the precondition and the **if**-statement in `minimum(·)`, the recursion step occurs for  $n = \text{len}(a) \geq 2$ . Since  $n \geq 2$ , we have that the sublist  $a[:-1]$  has at least one element. Hence, the recursive call `minimum(a[:-1])` satisfies the precondition. Thus, we can assume that the recursive call computes the correct result, i.e., that  $m$  is the index of the minimum among the first  $n - 1$  elements of  $a$ :

$$m = \operatorname{argmin}_{j=0,\dots,\text{len}(a)-2} a[j]. \quad (9.14)$$

Now, we distinguish two cases:

- **Case 1:  $a[m] \leq a[n - 1]$ .** In this case, the inner **if**-block in `minimum(·)` is executed, and we return  $m$ . This is correct, since  $a[m] \leq a[n - 1]$  and (9.14) give

$$m = \operatorname{argmin}_{j=0,\dots,\text{len}(a)-1} a[j].$$

- **Case 2:  $a[m] > a[n - 1]$ .** In this case, the **if**-block in `minimum(·)` is not executed, and we return  $n - 1$ . This is correct, since  $a[m] > a[n - 1]$ , (9.14), and the transitivity of  $>$  give

$$n - 1 = \operatorname{argmin}_{j=0,\dots,\text{len}(a)-1} a[j].$$

Hence, the recursion step is correct.

Since both the base case and the recursion step are correct, the program is partially correct.

2. **Show termination.** To show termination, we need to identify some progress throughout the recursion. For this, we consider the length of  $a$ ,  $\text{len}(a)$ . The precondition states that  $\text{len}(a) \geq 1$ , and the recursion stops when  $\text{len}(a) = 1$ . Furthermore, in each recursive call,  $\text{len}(a)$  is decreased by 1. Thus, if  $a$  satisfies the precondition, then the recursion will finish after a finite number of recursive calls.

Now, we have completed all the necessary steps, and we have shown that `minimum(.)` is totally correct.

**Third example: binary search** Let us now look binary search. In Section 6.2.2, we implemented binary search as follows:

```
# binary_search(a: List[U], k: U): int      U is a totally ordered universe
# Precondition: a is sorted in ascending order
# Postcondition:
#   Effect: None
#   a[result] == k, if k appears in a
#   result == None, otherwise
def binary_search(a, k):
    def bin_search(left, right):
        if left >= right:
            return None
        m_pos = left + (right - left) // 2
        m = a[m_pos]
        if k == m:
            return m_pos
        elif k < m:
            return bin_search(left, m_pos)
        else:
            return bin_search(m_pos + 1, right)
    return bin_search(0, len(a))
```

If we want to argue about the correctness, we need to give the specification of the helper function `bin_search(.)`.

```
# bin_search(left: int, right: int): int
# Precondition: xs is in ascending order
# binary_search(a: List[U], k: U): int      U is a totally ordered universe
# Precondition: a is sorted in ascending order
# Postcondition:
#   Effect: None
#   left <= result < right and a[result] == k, if k appears in a[left:right]
#   result == None, otherwise
def bin_search(left, right):
    ...
```

We prove the correctness of the helper function `bin_search(.)`. The correctness of `binary_search(.)` then follows immediately, since `binary_search(.)` consists of a



single call to `bin_search(·)` with parameters `0` and `len(a)`. Now, we go through the steps:

### 1. Show partial correctness.

- **Base cases.** The recursion has *two* base cases.
  - The first base case occurs for  $\text{left} \geq \text{right}$ . For this case, `bin_search(·)` returns **None**. This is correct, since in this case, `a[left:right]` is the empty list and cannot contain the search key `k`.
  - The second base case occurs for  $k = m$ . In this case, `bin_search(·)` returns `m_pos`. This is correct, since in this case, we have that `a[m_pos] = k` (this was explicitly checked by the **if**-instruction), and that  $\text{left} \leq m\_pos < \text{right}$  (because we know that  $\text{left} < \text{right}$  and by the definition of `m_pos`).

Hence, both base cases are correct.

- **Recursion steps.** By the second **if**-instruction in `bin_search(·)`, there is a recursive call for either  $k < m$  or  $k > m$ . In either case, the precondition is fulfilled, because the recursive call uses the same `a` as the current call, and this `a` fulfills the precondition. Thus, we can assume that the recursive call computes the correct result. We also know that  $\text{left} \leq m\_pos < \text{right}$ , as was argued in the second base case. Let us consider the two possibilities for the recursive call:

- **Case 1:  $k < m$ .** In this case, we call `bin_search(left, m_pos)`. Since this recursive call is correct, we know that if `k` appears in `a[left:m_pos]`, then `bin_search(left, m_pos)` gives an index in `a[left:m_pos]` where `k` appears; and if `k` does not appear in `a[left:m_pos]`, then the result of `bin_search(left, m_pos)` is **None**.

Now, we need to show that the current call to `bin_search(·)` is correct, that is, we need that if `k` appears in `a[left:right]`, then the current call gives an index in `a[left:right]`, where `k` appears; and if `k` does not appear in `a[left:right]`, then the current call returns **None**. However, this is clear, because by the precondition, we know that `a` is sorted, and by the current case, we know that  $k < m$ . Thus, we have that `k` appears in `a[left:right]` if and only if `k` appears in `a[left:m_pos]`, and we can use the result of the recursive call.

- **Case 2:  $k > m$ .** In this case, we call `bin_search(m_pos + 1, right)`. Since this recursive call is correct, we know that if `k` appears in the sublist `a[m_pos + 1:right]`, then `bin_search(m_pos + 1, right)` gives an index in this sublist where `k` appears; and if `k` does not appear in `a[m_pos + 1:right]`, then the result of `bin_search(m_pos + 1, right)` is **None**.

To show that the current call to `bin_search(·)` is correct, we note that by the precondition, we know that `a` is sorted, and by the current case, we

know that  $k > m$ . Thus, we have that  $k$  appears in  $a[\text{left}:\text{right}]$  if and only if  $k$  appears in  $a[m_{\text{post}} + 1:\text{right}]$ , and we can use the result of the recursive call.

Hence, the recursion step is correct.

Since both the base case and the recursion step are correct, the program is partially correct.

2. **Show termination.** In Section 8.1.3, we already argued that binary search has asymptotic worst-case running time  $O(\log n)$ . This means in particular means that the algorithm terminates after a finite number of steps.

Now, we have completed all the necessary steps, and we have shown that `bin_search(·)` is totally correct.

## 9.5 Correctness of sorting algorithms

Now, we will show the correctness of two sorting algorithms. For this, we employ the techniques we have seen in the previous sections.

### 9.5.1 Selection sort

Let us start with the correctness proof for selection sort. In Section 7.2, we implemented selection sort as follows:

```
# selection_sort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
def selection_sort(a):
    n = len(a)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if a[j] < a[min_index]:
                min_index = j
        swap(a, i, min_index)
```

This algorithm does not use any recursion, but **for**-loops. For our correctness proof, we will use an equivalent version that uses **while**-loops (and renames the variable `min_index` to `m`):

```
# selection_sort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
```

```
#      Result: None
def selection_sort2(a):
    n = len(a)
    i = 0
    while i < n:
        # (*)
        m = i
        j = m + 1
        while j < n:
            if a[j] < a[m]:
                m = j
            j = j + 1
        # (**)
        swap(a, i, m)
        i = i + 1
```



First, we note that the part of `selection_sort2(·)` that is between the lines `(*)` and `(**)` is almost identical to the code of the function `minimum` in the first example in Section 9.3. The main difference is that `m` is now initialized with `i` and that the loop variable is called `j` and starts at `m + 1` instead of `1`. Thus, we can reuse the correctness proof from the first example in Section 9.3 to obtain the following fact:

The inner **while**-loop in `selection_sort2(·)` always finishes after a finite number of steps. When we reach the line `(**)`, then we have

$$m = \operatorname{argmin}_{k=i, \dots, \operatorname{len}(a)-1} a[k]. \quad (9.15)$$

Now, we would like to show that `selection_sort2(·)` is correct. For this, we go through the steps:

1. **Show partial correctness.** We need to find a suitable invariant `Inv` for the outer **while**-loop in `selection_sort2(·)`. For this, we need to determine what all the iterations of the **while**-loop have in common. In this case, the answer is this: in each iteration, the sublist `a[0:i]` contains the `i` smallest elements of `a` in sorted order, and `a[i+1:n]` contains the remaining elements. Formally, we will use the following invariant:

$$\text{Inv} : i \leq n \quad (9.16)$$

$$\wedge a \text{ is a permutation of the input list} \quad (9.17)$$

$$\wedge a[0] \leq a[1] \leq \dots \leq a[i-1] \quad (9.18)$$

$$\wedge \text{if } i \geq 1, \text{ then } a[j] \geq a[i-1], \text{ for } j = i, \dots, n-1. \quad (9.19)$$

Now, we check that `Inv` satisfies the three properties:

- **Initialization.** Right before the outer **while**-loop, we have `i = 0`. Thus, it follows that `Inv` holds right before the **while**-loop: (9.16) holds because the length of any list is always at least 0; (9.17) holds because the input list `a` has not been changed; and (9.18) and (9.19) hold trivially because `i = 0`.

- **Maintenance.** Assume that the loop condition  $i < n$  and  $\text{Inv}$  both hold. Since  $i < n$  is stronger than (9.16), this means that we have

$$i < n \quad (9.20)$$

$$\wedge a \text{ is a permutation of the input list} \quad (9.21)$$

$$\wedge a[0] \leq a[1] \leq \dots \leq a[i-1] \quad (9.22)$$

$$\wedge \text{if } i \geq 1, \text{ then } a[j] \geq a[i-1], \text{ for } j = i, \dots, n-1. \quad (9.23)$$

Now, we execute the loop body  $B$ . Let  $i'$  and  $a'$  be the resulting values of  $i$  and  $a$  (and we use  $i$  and  $a$  to denote the values *before*  $B$  was executed). Then, we have to show that  $\text{Inv}$  holds for  $i'$  and  $a'$ . That is, we need to show that

$$i' \leq n$$

$$\wedge a' \text{ is a permutation of the input list}$$

$$\wedge a'[0] \leq a'[1] \leq \dots \leq a'[i'-1]$$

$$\wedge \text{if } i' \geq 1, \text{ then } a'[j] \geq a'[i'-1], \text{ for } j = i', \dots, n-1.$$

Looking at the loop body, we see that  $i' = i + 1$ , so this is equivalent to

$$i + 1 \leq n \quad (9.24)$$

$$\wedge a' \text{ is a permutation of the input list} \quad (9.25)$$

$$\wedge a'[0] \leq a'[1] \leq \dots \leq a'[i] \quad (9.26)$$

$$\wedge \text{if } i \geq 0, \text{ then } a'[j] \geq a'[i], \text{ for } j = i + 1, \dots, n-1. \quad (9.27)$$

Now, (9.24) follows immediately from (9.20): (9.20) states that  $i < n$  and  $i$  is an integer, so we have  $i + 1 \leq n$ . (9.25) follows from (9.21) and the fact that  $a'$  is obtained from  $a$  by switching the positions of two elements.<sup>3</sup> (9.26) follows from (9.22), (9.23), the fact that  $a'[i] = a[m]$  and that  $m$  is by (9.15) an index between  $i$  and  $n-1$ . (9.27) follows from (9.23), (9.15), and the fact that  $a'[i] = a[m]$  and that  $a'[m] = a[i]$ . Thus, we have shown that  $\text{Inv}$  is maintained by the outer **while**-loop.

- **Conclusion.** Assume that the negation of the loop condition,  $i \geq n$ , and  $\text{Inv}$  both hold. Since (9.16) states  $i \leq n$ , we have  $i = n$ . Now, (9.17) and (9.18) become

$$a \text{ is a permutation of the input list}$$

$$\wedge a[0] \leq a[1] \leq \dots \leq a[n-1].$$

This is exactly the postcondition of the sorting problem. Thus, we have shown that once the **while**-loop finishes, the invariant  $\text{Inv}$  implies the postcondition.

<sup>3</sup>Formally, we need a correctness proof for  $\text{swap}(\cdot)$ , but this function is so simple, that we can omit it here.

2. **Show termination.** First, we note that in Section 9.3, we already showed that the inner **while**-loop always terminates after a finite number of iterations. Thus, every iteration of the outer **while**-loop is finite, and we need to show that the outer **while**-loop terminates after a finite number of iterations. For this, we consider the variable  $i$ . The variable  $i$  is initialized to  $0$ , and in each iteration of the outer **while**-loop, it is increased by the *fixed* amount  $1$ . The outer **while**-loop finishes once  $i$  reaches the *fixed* value  $n$ . Thus, the outer **while**-loop will end after a finite number of steps.

Now, we have completed all the necessary steps, and we have shown that `selection_sort2(·)` is totally correct.

### 9.5.2 Quicksort

As a second example, we give a correctness proof for quicksort. Quicksort is quite an interesting example, since it combines loops and recursion. First, we consider the function `partition(·)`. In Section 7.5, we implemented `partition(·)` as follows:

```
# partition(a: List[U], l: int, r: int): int
# U is a totally ordered universe
# Precondition: 0 <= l < r <= len(a),
# Postcondition:
#     Effect: If m is the return value, then all elements are still in a, and
#             for l <= i < m, we have a[i] < a[m] and
#             for m < i < r, we have a[m] <= a[i].
#     Result: The index m of the pivot element is returned.
def partition(a, l, r):
    pivot = a[l]
    m = l
    for i in range(l + 1, r):
        if a[i] < pivot:
            swap(a, m + 1, i)
            m = m + 1
    swap(a, l, m)
    return m
```

For our correctness proof, we rewrite `partition(·)` to use a **while**-loop instead of a **for**-loop, and we rename `pivot` to `p`:

```
# partition(a: List[U], l: int, r: int): int
# U is a totally ordered universe
# Precondition: 0 <= l < r <= len(a),
# Postcondition:
#     Effect: If m is the return value, then all elements are still in a, and
#             for l <= i < m, we have a[i] < a[m] and
#             for m < i < r, we have a[m] <= a[i].
#     Result: The index m of the pivot element is returned.
def partition2(a, l, r):
```

```

p = a[l]
m = l
i = l + 1
while i < r:
    if a[i] < p:
        swap(a, m + 1, i)
        m = m + 1
    i = i + 1
swap(a, l, m)
return m

```



Now, we would like to show that `partition2(·)` is correct. For this, we go through the steps:

1. **Show partial correctness.** We need to find a suitable invariant  $Inv$  for the **while**-loop in `partition2(·)`. For this, we need to determine what all the iterations of the **while**-loop have in common. In this case, the answer is this: in each iteration, the sublist  $a[l + 1:m + 1]$  contains the elements of  $a[l + 1:i]$  that are smaller than  $p$ , and the sublist  $a[m + 1:i]$  contains the elements of  $a[l + 1:i]$  that are larger than or equal to  $p$ . Formally, we will use the following invariant:

$$Inv: i \leq r \quad (9.28)$$

$$\wedge a[l+1:i] \text{ is a permutation of the corresponding sublist of the input} \quad (9.29)$$

$$\wedge a[j] < p, \text{ for } j = l + 1, \dots, m \quad (9.30)$$

$$\wedge a[j] \geq p, \text{ for } j = m + 1, \dots, i - 1. \quad (9.31)$$

Now, we check that  $Inv$  satisfies the three properties:

- **Initialization.** Right before the **while**-loop, we have  $m = l$  and  $i = l + 1$ . Thus, it follows that  $Inv$  holds right before the **while**-loop: (9.28) holds because the precondition states  $l < r$  and hence  $i = l + 1 \leq r$ . And (9.29), (9.30), and (9.31) hold because the relevant sublists are all empty.
- **Maintenance.** Assume that the loop condition  $i < r$  and  $Inv$  both hold. Since  $i < r$  is stronger than (9.28), this means that we have

$$i < r \quad (9.32)$$

$$\wedge a[l+1:i] \text{ is a permutation of the corresponding sublist of the input} \quad (9.33)$$

$$\wedge a[j] < p, \text{ for } j = l+1, \dots, m \quad (9.34)$$

$$\wedge a[j] \geq p, \text{ for } j = m+1, \dots, i - 1. \quad (9.35)$$

Now, we execute the loop body  $B$ . Let  $i'$ ,  $m'$ , and  $a'$  be the resulting values of  $i$ ,  $m$ , and  $a$  (and we use  $i$ ,  $m$ , and  $a$  to denote the values *before*  $B$  was executed). Then, we have to show that  $Inv$  holds for  $i'$ ,  $m'$ , and  $a'$ . That is, we need to

show that

$$\begin{aligned} & i' \leq r \\ & \wedge a'[l+1:i'] \text{ is a permutation of the corresponding sublist of the input} \\ & \wedge a'[j] < p, \text{ for } j = l+1, \dots, m' \\ & \wedge a'[j] \geq p, \text{ for } j = m'+1, \dots, i' - 1. \end{aligned}$$

Looking at the loop body, we see that  $i' = i + 1$ , so this is equivalent to

$$i + 1 \leq r \tag{9.36}$$

$$\wedge a'[l+1:i+1] \text{ is a permutation of the corresponding sublist of the input} \tag{9.37}$$

$$\wedge a'[j] < p, \text{ for } j = l+1, \dots, m' \tag{9.38}$$

$$\wedge a'[j] \geq p, \text{ for } j = m'+1, \dots, i. \tag{9.39}$$

Now, (9.36) follows immediately from (9.32): (9.32) states that  $i < r$  and  $i$  is an integer, so we have  $i + 1 \leq r$ . (9.37) follows from (9.33) and the fact that  $a'$  is either identical to  $a$  or obtained from  $a$  by switching the positions of two elements. Now, we distinguish two cases:

- **Case 1:  $a[i] \geq p$ .** In this case,  $m' = m$  and  $a$  is unchanged. Thus, (9.38) and (9.39) follow from (9.34) and (9.35) and the fact that  $a[i] \geq p$ .
- **Case 2:  $a[i] < p$ .** In this case,  $m' = m + 1$  and  $a'$  is obtained from  $a$  by switching the elements at position  $i$  and  $m + 1$ . Thus, (9.38) and (9.39) follow from (9.34) and (9.35) and the fact that  $a[i] < p$ .

Thus, we have shown that  $\text{Inv}$  is maintained by the **while**-loop.

- **Conclusion.** Assume that the negation of the loop condition,  $i \geq r$ , and  $\text{Inv}$  both hold. Since (9.28) states  $i \leq r$ , we have  $i = r$ . Now, (9.29), (9.30), (9.31) become

$$\begin{aligned} & \wedge a[l+1:r] \text{ is a permutation of the corresponding sublist of the input} \\ & \wedge a[j] < p, \text{ for } j = l+1, \dots, m \\ & \wedge a[j] \geq p, \text{ for } j = m+1, \dots, r-1. \end{aligned}$$

Now, for the final result, we still swap  $p = a[l]$  with  $a[m]$ . Thus, the final postcondition becomese

$$\begin{aligned} & \wedge a[l:r] \text{ is a permutation of the corresponding sublist of the input} \\ & \wedge a[j] < a[m], \text{ for } j = l, \dots, m-1 \\ & \wedge a[j] \geq a[m], \text{ for } j = m+1, \dots, r-1. \end{aligned}$$

This is exactly the desired postcondition of `partition2(·)`. Thus, we have shown that once the **while**-loop finishes, the invariant  $\text{Inv}$  implies the postcondition.

2. **Show termination.** We consider the variable  $i$ . The variable  $i$  is initialized to  $l + 1$ , and in each iteration of the **while**-loop, it is increased by the *fixed* amount 1. The **while**-loop finishes once  $i$  reaches the *fixed* value  $r$ . Thus, the **while**-loop will end after a finite number of steps.

Now, we have completed all the necessary steps, and thus `partition2(·)` is totally correct.

Finally, we can argue about the correctness of quicksort. In Section 7.5, we implemented quicksort as follows:

```
# quicksort(a: List[U]): NoneType    # U is a totally ordered universe
# Precondition: None
# Postcondition:
#     Effect: The elements of a are rearranged in increasing order.
#     Result: None
def quicksort(a):
    def quicksort_help(l, r):
        if r - l <= 1:
            return
        m = partition2(a, l, r)
        quicksort_help(l, m)
        quicksort_help(m + 1, r)
    quicksort_help(0, len(a))
```

Similar to the correctness proof for binary search in Section 9.4, we can now prove the correctness of `quicksort(·)` by focusing on the helper function `quicksort_help(·)`. The specification of `quicksort_help(·)` is as follows:

```
# quicksort_help(l: int, r: int): NoneType
# Precondition: None
# Postcondition:
#     Effect: The elements of a[l:r] are rearranged in increasing order
#     Result: None
def quicksort_help(l, r):
    ...
```

Once we prove the correctness of `quicksort_help(·)`, the correctness of `quicksort(·)` follows immediately, because `quicksort(·)` consists of one call to `quicksort_help(·)` with parameters `0` and `len(a)`. Now, we go through the steps of a correctness proof for a recursive function:

### 1. Show partial correctness.

- **Base cases.** The base case occurs for  $r - l \leq 1$ . In this case, `quicksort_help(·)` returns immediately. This is correct, since in this case, the sublist `a[l:r]` consists of at most one element, and hence it is already sorted.
- **Recursion steps.** The recursion step occurs for  $r - l \geq 2$ . In the recursion step, we first call `partition2(a, l, r)`. By our correctness proof for



`partition2(·)`, we know that after this call we have

- $\wedge a[l:r]$  is a permutation of the corresponding sublist of the input
- $\wedge a[j] < a[m]$ , for  $j = l, \dots, m-1$
- $\wedge a[j] \geq a[m]$ , for  $j = m+1, \dots, r-1$ .

Next, we call `quicksort_help(l, m)` and `quicksort_help(m + 1, r)`. There is no precondition, thus we can assume that both calls are correct. Thus, we know that after these calls, the two sublists  $a[l:m]$  and  $a[m+1:r]$  are rearranged in increasing order. Together with the result of `partition2(·)`, this means that the whole sublist  $a[l:r]$  has been rearranged in increasing order. Hence, the recursion step is correct.

Since both the base case and the recursion step are correct, the program is partially correct.

2. **Show termination.** In Section 8.3.4, we already argued that quicksort has asymptotic worst-case running time  $O(n^2)$ . This means in particular means that the algorithm terminates after a finite number of steps.

Now, we have completed all the necessary step, and shown that `quicksort_help(·)` is totally correct. By the discussion above, this also means that `quicksort(·)` is total correct.

# Entwurf

## IV

### Functional Programming

## KAPITEL 10

### Functional Programming in Scala

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP3** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *funktionaler Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.



#### 10.1 Foundations of functional programming

As we saw in Section 2.4, the central notion in imperative programming is the *state*. An *imperative program* is a sequence of instructions, where each instruction changes the state of the system. The goal of a computation is to reach a desired *output state*.

In contrast, the central notion in functional programming is the *expression*. A *functional program* is a set of (mathematical) functions, constants, and expressions. The goal of a computation is to *evaluate* a given expression. Functions and recursion are crucial building blocks to obtain interesting expressions. In a functional program, there is no state. The value of an expression depends only on the expression itself. For example, let us consider the following Python snippet:

```
>> y = 0
>>> def f(x):
...     global y
...     y = y + 1
...     return 2 * x + y
...
>>> f(1)
```



```
3
>>> f(1)
4
>>> f(1)
5
```



The function  $f(\cdot)$  does not only return a value, but it also changes the state of the system ( $y$  is a global variable). We say that  $f(\cdot)$  has a *side effect*. Even more, the result of  $f(\cdot)$  does not only depend on the parameter  $x$ , but also on the state of the system. All of this is not allowed in functional programming.

In the context of functional programming, functions may not have side effects, and their output may depend only on the input parameters. This is called *referential transparency*.

## 10.2 Introduction to Scala

In the remainder of the class, we will use the programming language Scala to illustrate fundamental ideas in functional programming and object-oriented programming, and to explain basic data structures. Here are some noteworthy features of Scala:

- **Scala is a *multiparadigm* language.** That is, in Scala, we can write imperative, functional, and object-oriented programs. In this part, we focus on the functional part of Scala. The imperative and object-oriented features will be discussed in the next part.
- **Scala is a *JVM-language*.**<sup>1</sup> This means that Scala depends on the runtime environment, libraries, and tools that were originally developed for the Java programming language.
- **Scala is a *compiled* language.** Before we can run a Scala-program, a special program, the *compiler*, must translate it into machine language. This machine language program can then be executed by the JVM. During this translation process, the compiler can find possible errors in the program and make suggestions on how to fix them. In contrast, Python is an *interpreted* language: to run a Python-program, a special program, the *interpreter*, reads the Python-program line by line and executes each line on the fly. In this way, many errors are only discovered when the line that contains the error is executed.

**Attention:** Strictly speaking, being compiled or interpreted is not a property of a programming language, but of an *implementation* of a programming language. It is perfectly possible to write a compiler for Python or an interpreter for Scala. The point is that the most common implementation of Python is an interpreter and the most common implementation of Scala is a compiler.



<sup>1</sup>JVM stands for *Java virtual machine*.

- **Scala is *statically typed*.** This means that before we can use a variable, we must *declare* it in our program. With this declaration, we also fix a type for the variable. Then, the variable can be used only for data objects of this type. In contrast, Python is *dynamically typed*. In Python, we do not need to declare variables, and a variable can be associated with data objects of widely different types throughout the execution of a program.

In this class, we will work with Scala 3. While Scala 2 had a syntax that was very close to Java, the syntax of Scala 3 was relaxed to be more like Python (e.g., in Scala 3, we can mark blocks by using indentation, as in Python, whereas in Scala 2, we need to mark blocks by curly brackets.) Thus, using Scala 3 makes it easier for us to build on the skills that we have acquired during the first parts of this class.

**Declarations.** As mentioned above, in Scala, we must *declare* a variable before we can use it. There are two possible ways to declare a variable: with a *value declaration* or with a *variable declaration*. In a functional context, we use only value declarations. Variable declarations will be discussed later, in Section 13.1. The syntax for a value declaration is as follows:

```
val name: Type = expression
```

This declares a new variable name of type **Type**. Scala evaluates expression and stores the resulting value in name. After that, we can use the variable name in expressions in our program, but we are not allowed to assign a new value to name. We say that name is *immutable*. Thus, variables that are declared with a value declaration behave like mathematical constants and not like variables in an imperative program—in a functional program, there is no state. Here are a few examples:

```
scala> val n: Int = 42*13
val n: Int = 546

scala> n = 12
-- [E052] Type Error: -----
1 | n = 12
  | ^^^^^^
  | Reassignment to val n

scala> val b: Boolean = (true || false) && true
val b: Boolean = true

scala> val f: Double = 1.1 * 2.2 + 0.1
val f: Double = 2.5200000000000005
```

**Attention:** Scala has *type inference*. That is, we can often omit the type in a variable declaration, and Scala will guess an appropriate type on its own. However, it is good programming style to write the type explicitly. In this class, we will do so.



**Conditional expressions.** As we already saw in Section 2.3, conditional expressions are multi-part expressions where we have a condition that determines which expression is evaluated. Conditional expressions in Scala look very much like **if**-statements. The syntax is as follows:

```
if condition1 then
  expression1
else if condition2 then
  expression2
...
else
  expressionn
```

Here is an example:

```
scala> if 10 > 12 then "Hello" else "World"
val res: String = World

scala> if 46 - 12 < 30 then 4.4 else 1.2
val res: Double = 1.2

scala> if 10 > 12 then "One" else if 7 < 8 then "Two" else "Three"
val res: String = Two
```

Scala also has an **if**-statement, with basically the same syntax. We will see it in Section 13.1.

**Loops.** There are **while**- and **for**-loops in Scala. They are very similar to their Python-counterparts. However, loops are not allowed in functional programs, so we will postpone the discussion of loops in Scala to 13.1.

**Functions.** Since Scala is statically typed, we need to give the types of the parameters and of the return value when we declare a function. This is called the *signature* of the function. Furthermore, in the functional context, the function body does not consist of a sequence of statements, but of a single *expression*. This expression may be quite complex and involve recursion and long case distinctions. When the function is called, the corresponding expression is evaluated. The general syntax looks like this:

```
def name(param1: Type1, param2: Type2 ...): ReturnType =
  expression
```

Here is a concrete example, using a conditional expression:

```
scala> def func(a: Int, b: Int): Int =
  |   if a > b then
  |     3 + 5
```

```
|      else  
|      -3
```

```
scala> func(1,2)  
val res: Int = -3
```

```
scala> func(2,1)  
val res: Int = 8
```

For very small functions, we can write everything into a single line. This may look nicer for some people.<sup>2</sup>

```
scala> def func(a: Int, b: Int): Int = if a > b then 3 + 5 else -3
```

```
scala> func(1,2)  
val res: Int = -3
```

```
scala> func(2,1)  
val res: Int = 8
```

In Scala, functions are “first class citizens”. They can be treated as values and can be assigned to variables. We will say much more about this in Chapter 11. Here is a first example:

```
scala> val increase: Int => Int = (x: Int) => x + 1  
val increase: Int => Int = Lambda/0x0000776feb5a1c00@367e0b0e
```

```
scala> increase(10)  
val res: Int = 11
```

```
scala> increase(23)  
val res: Int = 24
```

Here, `increase` is a function whose input is an integer `a`. The output is `x + 1`. This can be shortened to:

```
scala> val increase: Int => Int = (_:Int) + 1  
val increase: Int => Int = Lambda/0x0000776feb5a2be8@5b9ed77b
```

```
scala> increase(10)  
val res: Int = 11
```

```
scala> increase(23)  
val res: Int = 24
```

**The main function.** For most of this class, we will use the REPL to show examples of Scala code. However, just like in Python, it is also possible to write Scala code into a

---

<sup>2</sup>Just a matter of taste...

file that can be executed from the command line. In Python, we would just write a sequence of instructions into a file, and the Python interpreter would start to execute the file line by line, from top to bottom. In Scala, there are two main differences:

1. Since Scala is a *compiled* language, the file will be compiled before it is executed. During this compile-step, the file is checked for syntax and semantic errors. If the compilation fails, the program cannot be run (in contrast, the Python interpreter often reports an error only when it is encountered during the execution of the program).
2. Scala programs need an *entry-point*. In a Scala program, we need to define a dedicated *main-function*. The main-function is executed when the program is called from the command line. The main function has return type `Unit` and is marked with the annotation `@main`. Here is an example of a Scala-file with a main function:

```
def foo(x: Int): Int = 2*x

@main
def test(): Unit =
  println(s"foo(10) = ${foo(10)}")
```

This program defines a function `foo` and a main-function `test`. The main-function calls `foo` and prints the result onto the terminal.<sup>3</sup> The output is as follows:

```
wolfi@work:~$ scala main.scala
Starting compilation server
Compiling project (Scala 3.6.3, JVM (22))
Compiled project (Scala 3.6.3, JVM (22))
foo(10) = 20
```

## 10.3 Data types in Scala

As we said in Section 10.2, the type of an expression in Scala is fixed at compile time. In variable declarations and in function signatures, we must provide the type information for the compiler. In this section, we will discuss some important data types in Scala—most of them are very similar to their Python counterparts.

<sup>3</sup>The Scala-instruction `println` outputs a string onto the terminal. Actually, main-functions and input/output belong into the imperative world, but some of our later functional examples do not work in the REPL and need the Scala-compiler and a main-function.



## 10.3.1 Primitive data types

As we saw in Section 3.1, *primitive data types* are data types whose data objects consist of a single atomic value. In Scala, primitive data types are sometimes also called *value types*. All other types are called *reference types*. Later, we will examine this distinction in more detail. Here are the most important primitive data types in Scala:

- **Boolean**. The data type **Boolean** represents truth values. There are two truth values: **true** and **false**.<sup>4</sup> The logical operations are: `&&` (and), `||` (or), and `!` (not). As in Python, the logical operations evaluate expressions in a lazy manner, e.g.,

```
false && foo()
```

never evaluates `foo()`, since the first operand **false** already determines the value of the `&&`-operation.

- **Byte**. The data type **Byte** represents integer numbers in  $\{-2^7, \dots, 2^7 - 1\}$ . The integer numbers are stored in 8-bit two's complement representation (see Section 3.3.3 for more details on the internal representation of signed integers).
- **Short**. The data type **Short** represents integer numbers in  $\{-2^{15}, \dots, 2^{15} - 1\}$ . The integer numbers are stored in 16-bit two's complement representation .
- **Int**. The data type **Int** represents integer numbers in  $\{-2^{31}, \dots, 2^{31} - 1\}$ . The integer numbers are stored in 32-bit two's complement representation.
- **Long**. The data type **Long** represents integer numbers in  $\{-2^{63}, \dots, 2^{63} - 1\}$ . The integer numbers are stored in 64-bit two's complement representation.
- **Float**. The data type **Float** represents fractional numbers. The fractional numbers are stored as 32-bit floating-point numbers according to the IEEE-754 standard (see Section 3.3.4 for more details on the internal representation of floating-point numbers).
- **Double**. The data type **Double** represents fractional numbers. The fractional numbers are stored as 64-bit floating-point numbers according to the IEEE-754 standard .
- **Char**. The data type **Char** represents single characters, for example `'a'`, `'4'`, `'T'`. Internally, the values of **Char** are just unsigned integers from  $\{0, \dots, 2^{16} - 1\}$ . These integers are interpreted as characters according to the UNICODE standard (see Section 3.3.5 for more details).
- **Unit**. The data type **Unit** is used to represent the *absence* of a value. A variable of type **Unit** has only one possible value, namely `()`. This is similar to `NoneType` and **None** Python (see Section 3.1.1).

<sup>4</sup>Unlike in Python, the truth values in Scala are not capitalized.

The elementary operations for numbers in Scala are very similar to those in Python. We note that the type system in Scala is much more detailed than in Python: Python has one data type for integers that represents *arbitrarily large* signed integers. Scala has *four* data types for signed integers, all of them with a bounded range and each with a different precision. Similarly, Python has one data type for floating-point numbers, whereas Scala has two, again with a choice of precision. This shows the different design choices for the two programming languages: Python favors simplicity over performance. Scala gives us more detailed control, at the cost of an increased complexity.

**Type compatibility and widening.** In Scala, every variable has a type. If we assign an expression to a variable, then the Scala compiler checks whether the types of the expression and of the variable match. If so, the assignment is allowed, if not, the compiler produces an error.

```
scala> val d: Double = 1.1
val d: Double = 1.1

// We can assign an expression of type Double to a variable of type Double.
scala> val e: Double = d
val e: Double = 1.1

// We cannot assign an expression of type Double to a variable of type Int.
scala> val i: Int = d
-- [E007] Type Mismatch Error: -----
1 | val i: Int = d
  |           ^
  |           Found:    (d : Double)
  |           Required: Int
```

Since Scala has four types for signed integers and two types for floating-point numbers, a strict enforcement of the typing rule for assignments would be very inconvenient: formally, an expression of type `Byte` has a different type than a variable of type `Int`. However, an assignment should still be possible. For this, we have the notion of *type compatibility*: we say that type *A* is *compatible* with type *B* if we can use an expression of type *A* whenever an expression of type *B* is expected. In Scala, smaller integer types are compatible with larger integer types, and smaller floating-types are compatible with larger integer types (this is called *widening*).<sup>5</sup> Thus, the following assignments are possible:

```
scala> val b: Byte = 127
val b: Byte = 127
```

<sup>5</sup>This is an example of a *type hierarchy*. Later, in Section 13.6, we will talk more about type hierarchies and inclusion polymorphism in the context of object-oriented programming.

```
scala> val s: Short = b
val s: Short = 127

scala> val i: Int = s
val i: Int = 127

scala> val l: Long = b
val l: Long = 127

scala> val l2: Long = i
val l2: Long = 127

scala> val f: Float = 1.1f
val f: Float = 1.1

scala> val d: Double = f
val d: Double = 1.100000023841858
```

However, the other direction does not work:

```
scala> val i: Int = 127
val i: Int = 127

scala> val b: Byte = i
-- [E007] Type Mismatch Error: -----
1 | val b: Byte = i
  |               ^
  |               Found:    (i : Int)
  |               Required: Byte

scala> val d: Double = 1.1
val d: Double = 1.1

scala> val f: Float = d
-- [E007] Type Mismatch Error: -----
1 | val f: Float = d
  |               ^
  |               Found:    (d : Double)
  |               Required: Float
```

**Attention:** When using numerical constants, we sometimes need to be explicit about the desired type in order to avoid type errors. For example, the following assignment results in an error:



```
scala> val f: Float = 1.1 + 2.2
-- [E007] Type Mismatch Error: -----
1 | val f: Float = 1.1 + 2.2
  |               ^^^^^^^^^
  |               Found:    (3.3000000000000003d : Double)
  |               Required: Float
```

The reason is that Scala interprets the expression on the right hand side as having type **Double**, and hence the result is not type compatible with the variable `f`. To fix this, we can add the letter `f` to the floating-point constants to indicate that they should be treated as values of type **Float**:

```
scala> val f: Float = 1.1f + 2.2f
val f: Float = 3.3000002
```

If we insist on going into the other direction, we have to make this explicit (this is called *narrowing*). This, however, might result in unexpected consequences we try to narrow a value that lies outside the range of the target type.

```
scala> val i: Int = 10
val i: Int = 10

scala> val b: Byte = i.toByte
val b: Byte = 10

scala> val j: Int = 500
val j: Int = 500

scala> val c: Byte = j.toByte
val c: Byte = -12 //Oops.
```

There are also other type compatibility relationships between the primitive data types in Scala (e.g., **Int** is compatible with **Float**; **Char** is compatible with **Int**, but not with **Byte**, etc.) We refer to the Scala documentation for more details.

### 10.3.2 Simple composite data types

Similar to Python (see Section 3.2), Scala also supports composite data types that collect several data items into a single unit. Here, we will look at two important examples that are relevant in the functional context: Strings and tuples. We will learn much more about composite data types in Scala in Chapter 13 when we talk about imperative and object-oriented programming in Scala.

**Strings.** Strings are *immutable sequences of characters*. Unlike Python, Scala distinguishes between characters and strings: `'a'` is a character, a value of type **Char**; whereas `"a"`

is a string of length one, i.e., a value of type **String**. The two types are not compatible: to go from a character to a string, we need to use an explicit type conversion:

```
scala> val c: Char = 'a'
val c: Char = a

scala> val s: String = c
-- [E007] Type Mismatch Error: -----
1 | val s: String = c
  |               ^
  |               Found:    (c : Char)
  |               Required: String

scala> val s: String = c.toString
val s: String = a
```

To go from a string to a character, we can use Scala's index notation: given a string *s* and an integer *i*, we write *text(i)* to access the *i*-th character in *s* (starting from 0).

```
scala> val s: String = "Hello"
val s: String = Hello

scala> s(0)
val res: Char = H

scala> s(3)
val res: Char = l

scala> s(10)
java.lang.StringIndexOutOfBoundsException: Index 10 out of bounds for length 5
```

The length of a string *s* can be obtained with the notation *s.length*.

```
scala> val s: String = "Hello"
val s: String = Hello

scala> s.length
val res: Int = 5
```

Every data value in Scala has a (more or less) human-readable representation as a string. We can obtain use this representation in a larger string by using the syntax *s"\$var1 text \$var2"*. For example, we can write.

```
scala> val f: Float = 3.7
val f: Float = 3.7

scala> val i: Int = 10
val i: Int = 10
```

```
scala> val s: String = s"f is $f and i is $i."  
val s: String = f is 3.7 and i is 10.
```

The notation `s"..."` means that the string is a *formatted string* and that the occurrences of `$v` are replaced by the text representation of the current value of `v`. In fact, we can use complete expressions after `$`:

```
scala> val s: String = s"1 + 1 = ${1+1}"  
val s: String = 1 + 1 = 2
```

**Tuples.** Like in Python, we can collect a finite number of values of different data types into a unit. This is called a tuple.

```
scala> val student: (String, String, Int) = ("Toni", "Mustermensch", 22)  
val student: (String, String, Int) = (Toni, Mustermensch, 22)
```

We can use Scala's index notation to access individual elements of a tuple.

```
scala> student(0)  
val res: String = Toni  
  
scala> student(2)  
val res: Int = 22  
  
scala> student(10)  
-- Error: -----  
1 | student(10)  
  |      ^^  
  |      index out of bounds: 10
```

Tuples are *immutable*. Once a tuple is created, we cannot change its values.

```
scala> student(0) = "Max"  
-- [E008] Not Found Error: -----  
1 | student(0) = "Max"  
  | ^^^^^^^  
  | value update is not a member of (String, String, Int)
```

## 10.4 Syntactic structures for defining functions

Functions are central to the functional programming paradigm. To obtain meaningful functional programs, we need to be able to write functions that support complex case distinctions and recursion patterns. To facilitate this, functional programming languages have developed several syntactic constructs that make it easy to define

complex functions. Here, we will discuss two such constructs: pattern matching and guards.

### 10.4.1 Pattern matching

Suppose we would like to implement a function that involves a longer case distinction. We already saw that this can be achieved with a conditional expression. However, a nicer and cleaner way is to use *pattern matching*. Here, the idea is to look at an unknown expression *e* and to *match* it with a list of *pattern* expressions. That is, we go through the list of pattern expressions until we encounter the first pattern expression whose structure corresponds to the structure of *e*. Once a pattern expression matches, we evaluate the associated expression and return the result. In Scala, this is implemented using the **match**-operator. Here is an example:

```
scala> def message(day: String): String =
|     day match
|         case "Monday"    => "Start of the week"
|         case "Friday"    => "Almost done"
|         case "Saturday"  => "Weekend"
|         case "Sunday"    => "Weekend over"
|         case _           => "Work"

scala> message("Friday")
val res: String = Almost done

scala> message("Sunday")
val res: String = Weekend over

scala> message("Wednesday")
val res: String = Work

scala> message("Hello")
val res: String = Work
```

The function `message` matches `day` with the pattern expressions, row by row. At the first match, Scala will evaluate the associated expression. If there is more than one matching pattern, **match** will evaluate the first one.

The pattern `_` is called a *wild card*. It matches with every expression. In this example, `_` plays the same role as the **else**-branch in conditional expression. We can also use variables in the pattern expressions. If the pattern matches, then the variable is associated with the corresponding part of the input expression and can be used in the corresponding expression. Here is a simple example:

```
scala> def message(day: String): String =
|     day match
|         case "Monday"    => "Start of the week"
|         case "Friday"    => "Almost done"
```

```
|      case "Saturday" => "Weekend"  
|      case "Sunday"   => "Weekend over"  
|      case x           => s"$x is another day"
```

```
scala> message("Tuesday")  
val res: String = Tuesday is another day
```

```
scala> message("Hello")  
val res: String = Hello is another day
```

We can also use more complicated expressions for the expression we want to match. For example:

```
scala> def number(n: Int): String =  
|      n - 1 match  
|      case 0 => "Foo"  
|      case 1 => "Bar"  
|      case x => s"$x"
```

```
scala> number(1)  
val res: String = Foo
```

```
scala> number(2)  
val res: String = Bar
```

```
scala> number(10)  
val res: String = 9
```

```
scala> number(0)  
val res: String = -1
```

Pattern matching can be combined with tuples to extract and match different components of a tuple. Here is an example:

```
scala> def name(student: (String, String, Int), lastName: Boolean): String =  
|      (student, lastName) match  
|      case (_, last, _), true    => last  
|      case (first, _, _), false => first
```

```
scala> name(("Max", "Mustermensch", 22), true)  
val res: String = Mustermensch
```

```
scala> name(("Katharina", "Mustermach", 42), false)  
val res: String = Katharina
```



### 10.4.2 Guards

Guards are a way to add further conditions to the patterns in a **match**-expression. For example, let us consider the following recursive implementation of the Fibonacci-numbers that uses pattern matching:

```
scala> def fib(n: Int): Int =  
  |   n match  
  |     case 0 => 1  
  |     case 1 => 1  
  |     case _ => fib(n - 1) + fib(n - 2)  
  
scala> fib(1)  
val res: Int = 1  
  
scala> fib(10)  
val res: Int = 89  
  
scala> fib(-1)  
java.lang.StackOverflowError
```

To check that the parameter *n* should not be negative,<sup>6</sup> we can add additional *guards* to the patterns. This might look as follows:

```
scala> def fib2(n: Int): Int =  
  |   n match  
  |     case 0 => 1  
  |     case 1 => 1  
  |     case x if x > 1 => fib(n - 1) + fib(n - 2)  
  |     case x if x < 0 => throw new Exception("n may not be negative!")  
  
scala> fib2(1)  
val res: Int = 1  
  
scala> fib2(10)  
val res: Int = 89  
  
scala> fib2(-1)  
java.lang.Exception: n must not be negative!
```

## 10.5 Tail recursion

Let us take a closer look at the function `fib` from Section 10.4.2. This function turns out to be quite inefficient:

---

<sup>6</sup>A suitable specification for `fib` would not allow this.

```
scala> def fib(n: Int): Int =  
  |   n match  
  |     case 0 => 1  
  |     case 1 => 1  
  |     case _ => fib(n - 1) + fib(n - 2)  
  
scala> fib(40)  
val res: Int = 165580141
```

Even for relatively small parameters, such as  $n = 40$ , there is a noticeable delay in the computation of `fib(n)`.<sup>7</sup> This comes from the fact that when evaluating `fib(n)`, Scala will recompute many values of `fib` over and over (particularly the ones for small  $n$ ). For example, here is what happens for `fib(4)`:

```
fib(4)  
= fib(3) + fib(2)  
= (fib(2) + fib(1)) + fib(2)  
= ((fib(1) + fib(0)) + fib(1)) + fib(2)  
= ((1 + fib(0)) + fib(1)) + fib(2)  
= ((1 + 1) + fib(1)) + fib(2)  
= (2 + fib(1)) + fib(2)  
= (2 + 1) + fib(2)  
= 3 + fib(2)  
= 3 + (fib(1) + fib(0))  
= 3 + (1 + fib(0))  
= 3 + (1 + 1)  
= 3 + 2  
= 5
```

Thus, to evaluate `fib(4)`, we evaluate `fib(0)` two times and `fib(1)` three times.

To improve this, we can use a general technique, called the *accumulator technique*. Here, we compute the function value step by step, storing the intermediate result in a parameter (the *accumulator*) that is passed to the recursive call. The recursive call can then immediately compute the final result instead of passing it back to the caller.

One additional benefit of the accumulator technique is that the resulting recursive function is *tail recursive*. We say that a recursive function is *tail recursive* if it contains a single recursive call that is on the outside of the function expression. Tail recursion is a particularly efficient way of recursion that constitutes the functional analog to iterative loops. Before we see how tail recursion can be used to compute the Fibonacci numbers, we first look at the slightly simpler case of the factorial function.

### 10.5.1 Computing the factorial function

Here is a straightforward way to implement the factorial function  $n!$  in Scala:

---

<sup>7</sup>One can prove that the number of arithmetic operations in the recursive implementation of `fib(n)` is  $O(\phi^n)$ , where  $\phi = \frac{1+\sqrt{5}}{2}$  is the golden ratio.

```
scala> def fac(n: Int): Int =  
  |   n match  
  |     case 0 => 1  
  |     case _ => n * fac(n - 1)  
  
scala> fac(10)  
val res: Int = 3628800
```

This implementation is not tail recursive, since in the second case of the **match**-expression, the recursive call to **fac** is not on the outside. This results in the fact that the space for the recursion can grow with **n**. Here is what happens for **fac(5)**:

```
fac(5)  
= 5 * fac(4)  
= 5 * (4 * fac(3))  
= 5 * (4 * (3 * fac(2)))  
= 5 * (4 * (3 * (2 * fac(1))))  
= 5 * (4 * (3 * (2 * (1 * fac(0)))))  
= 5 * (4 * (3 * (2 * (1 * 1))))  
= 5 * (4 * (3 * (2 * 1)))  
= 5 * (4 * (3 * 2))  
= 5 * (4 * 6)  
= 5 * 24  
= 120
```

We can use the accumulator technique to make **fac** tail recursive. For this, we need a helper function **step** that introduces the accumulator. The helper function **step** computes the factorial step by step, passing the partial product in the accumulator:

```
scala> def fac2(n: Int): Int =  
  |   def step(acc: Int, n: Int): Int =  
  |     n match  
  |       case 0 => acc  
  |       case _ => step(n * acc, n - 1)  
  |   step(1, n)  
  
scala> fac2(10)  
val res: Int = 3628800
```

As we can see, **step** is now tail recursive, since there is only one recursive call that occurs on the outside of the expression of the recursion step. The parameter **acc** serves as the accumulator: it stores the partial product for the factorial function and passes it to the next recursive call. In the base case, the complete product has been computed in the accumulator, and we simply return **acc**. Here is what happens for **fac2(5)**:

```
fac2(5)  
= step(1, 5)  
= step(5 * 1, 5 - 1)
```

```
= step(5, 4)
= step(4 * 5, 4 - 1)
= step(20, 3)
= step(3 * 20, 3 - 1)
= step(60, 2)
= step(2 * 60, 2 - 1)
= step(120, 1)
= step(1 * 120, 1 - 1)
= step(120, 0)
= 120
```

### 10.5.2 Computing the Fibonacci numbers

Let us come back to the Fibonacci numbers. To make `fib` tail recursive, we again use a helper function `step` that introduces the accumulator. However, the idea is now a bit different: to compute `fib(n)`, we need to know two predecessor values `fib(n - 1)` and `fib(n - 2)`. Thus, we use *two* accumulators in which we construct the current function value and its predecessor. The construction occurs bottom up during the recursive calls:

```
scala> def fib2(n: Int): Int =
|     def step(acc1: Int, acc2: Int, n: Int): Int =
|         n match
|             case 0 => 1
|             case 1 => acc1
|             case _ => step(acc1 + acc2, acc1, n - 1)
|     step(1, 1, n)
|
scala> fib2(40)
val res: Int = 165580141
```

Again, the helper function `step` is tail recursive, since the recursion step has a single recursive call that is on the outside. Moreover, we can prove that this implementation is now much more efficient, since each function value is computed only once.<sup>8</sup> Here is what happens for `fib2(4)`:

```
fib2(4)
= step(1, 1, 4)
= step(1 + 1, 1, 4 - 1)
= step(2, 1, 3)
= step(2 + 1, 2, 3 - 1)
= step(3, 2, 2)
= step(3 + 2, 3, 2 - 1)
= step(5, 3, 1)
= 5
```

<sup>8</sup>One can show that in this implementation, we only need  $O(n)$  arithmetic operations.

### 10.5.3 Why tail recursion?

Tail recursive functions have several advantages. First of all, we have seen that tail recursion may be more time efficient. For example, we managed to decrease the number of arithmetic operations that are needed to compute the  $n$ -th Fibonacci number from  $O(\phi^n)$  to  $O(n)$ . This comes from the fact that the same function values are not evaluated multiple times. Moreover, the result of the recursive call is not needed to evaluate another expression (there is no back-tracking of the recursive calls).

Additionally, the Scala compiler can detect whether a given function is tail recursive. If so, the Scala compiler translates this recursion into an imperative program that uses a loop. In this way, we gain two advantages. On the one hand, we can write code in the beautiful context of functional programming, where we do not need to worry about state and the messiness of imperative programming. On the other hand, we get the efficiency of imperative programming. In particular, the resulting imperative loop only needs a constant amount of space overhead, whereas the recursion needs additional space that is proportional to the depth of the recursion (see Section 7.7.1 for more details).

The following example shows how the Scala compiler optimizes the recursion.

```
scala> def boom_nontail(n: Int): Int =
  |   n match
  |     case 0 => throw new Exception("Boom")
  |     case _ => n + boom_nontail(n - 1)

scala> def boom_tail(n: Int): Int =
  |   n match
  |     case 0 => throw new Exception("Boom")
  |     case _ => boom_tail(n - 1)

scala> boom_nontail(10)
java.lang.Exception: Boom
  at rs$line$15$.boom_nontail(rs$line$15:3)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)
  at rs$line$15$.boom_nontail(rs$line$15:4)

scala> boom_tail(10)
java.lang.Exception: Boom
  at rs$line$16$.boom_tail(rs$line$16:3)
```

`boom_nontail` is not tail recursive. The error message for the call `boom_nontail(10)` shows 10 recursive invocations in the stack trace of the exception. `boom_tail` is tail

recursive. The error message for the call `boom_tail(10)` shows only one invocation on the stack—because `boom_tail` was first translated into a loop by the Scala compiler.

### Lists and Higher-Order Functions

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP3** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *funktionaler Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP7** Du entwickelst Algorithmen zur Lösung vorgegebener *algorithmischer Listen- und Baumprobleme* und stellst diese mithilfe unterschiedlicher Programmierkonzepte in *Scala und Python* dar.



#### 11.1 Lists in Scala

*Lists* are a central data type in the context of functional programming. In Python, we have already encountered lists, and we have used them extensively. However, the imperative lists in Python are quite different from the functional lists in Scala. Here are some of the main differences:

- **Lists in Scala are *immutable*.** Once a list in Scala has been created, we cannot change the elements in the list. If we want to change a list, we have to create a new one.
- **All elements in a Scala list have the same type.** Since Scala is statically typed, we need to fix a type for a list variable. This type also defines the type of the elements in the list.

- **Lists in Scala are *not* random-access.** In Python, lists support random access. That is, we can use the index notation to access every element in a Python list in constant time. In Scala, lists are *recursive structures*. This means in particular that to access an element in a Scala list, we must visit all the elements that come before it; when accessing the  $i$ -th element of a Scala list, we need  $i$  steps.

We will now give the formal definition of a list in Scala.

**Structure of a list in Scala.** Let  $T$  be a type. Then, the *lists of type  $T$*  are defined inductively as follows:

- **The empty list.** The *empty list* `Nil` is a list of type  $T$ . It does not contain any elements.
- **Putting a new element in front of a list.** Let  $x$  be a value of type  $T$ , and let  $xs$  be a list of type  $T$ . Then  $x :: xs$  is also a list of type  $T$ . It is the list that is obtained by writing  $x$ , followed by all the elements of  $xs$ .

The Scala-type for the lists of type  $T$  is `List[T]`. We call `::` the *cons-operator*.<sup>1</sup> The cons-operator gets a new first element  $x$  and an existing list  $xs$ , and it creates a new list that has  $x$  as the first element, followed by the elements of  $xs$ . We call  $x$  the *head* of the resulting list, and  $xs$  the *tail* of the resulting list.<sup>2</sup> Here, are some examples:

```
scala> val a: List[String] = Nil
val a: List[String] = List()

scala> val b: List[Int] = 12 :: 13 :: 7 :: Nil
val b: List[Int] = List(12, 13, 7)

scala> val c: List[Char] = 'a' :: 'b' :: Nil
val c: List[Char] = List(a, b)
```

Here, `a` is the empty list of type `String`. The list `b` is a list of type `Int` that contains the elements 12, 13, 7, in this order. The construction of `b` is as follows: we start with the empty list of type `Int`, `Nil`. Then, we use the cons-operator for 7 and the empty list to create a list that contains only the element 7. Then, we use the cons-operator with 13 and the list that contains 7 to create a list that contains 13 and 7. Finally, we use the cons-operator for 12 and the list that contains 13 and 7 to create a list that contains 12, 13, and 7 in this order. The list `c` is constructed similarly, it contains the `Char` elements 'a' and 'b', in this order.

A slightly more convenient way for defining a list is to use the *constructor method* for lists. Here, we can just write `List`, followed by the elements of the list, in parentheses.

<sup>1</sup>This terminology comes from the programming language LISP. cons is short for *constructor*, since the cons-operator constructs a new list by adding a new first element.

<sup>2</sup>In LISP, the terminology is CAR (Contents of the Address Part of the Register) for head and CDR (Contents of the Decrement Part of the Register) for tail.



```
scala> val d: List[Boolean] = List(true, false, false)
val d: List[Boolean] = List(true, false, false)
```

## 11.2 Programming with lists

Now, we would like to write functional programs that operate on lists. As is common in the functional context, the main tools for this will be pattern matching and recursion.

First, let us see how we can use pattern matching to access the constituent parts of a list. The following two functions allow us to access the first element of a list of type `Int` (the head), and to remove the first element of a list of type `Int` to get to the list with the remaining elements (the tail). For both functions, the precondition requires that the list is not empty. They are implemented as follows:

```
scala> def head(a: List[Int]): Int =
  |   a match
  |     case Nil => throw new Exception("No head.")
  |     case x::_ => x

scala> def tail(a: List[Int]): List[Int] =
  |   a match
  |     case Nil => throw new Exception("No tail.")
  |     case _::xs => xs

scala> head(List(1,2,3))
val res: Int = 1

scala> tail(List(1,2,3))
val res: List[Int] = List(2, 3)

scala> head(List(1))
val res: Int = 1

scala> tail(List(1))
val res: List[Int] = List()
```

The pattern `Nil` matches the empty list. The pattern `x::xs` matches a non-empty list. The variable `x` is bound to the first element of the non-empty list, the variable `xs` is bound to the list of the remaining elements.

The functions `tail` and `head` are already defined in Scala. To call these predefined functions, we need a slightly different syntax:

```
scala> List(1,2,3).head
val res: Int = 1

scala> List(1,2,3).tail
val res: List[Int] = List(2, 3)
```

The reason for this slightly different syntax comes from the object-oriented nature of Scala. This will be discussed in Chapter 13. The following example uses pattern matching to test whether a list of type `Int` is empty.

```
scala> def isEmpty(a: List[Int]): Boolean =  
  |   a match  
  |     case Nil => true  
  |     case _  => false  
  
scala> isEmpty(Nil)  
val res: Boolean = true  
  
scala> isEmpty(List(1))  
val res: Boolean = false  
  
scala> isEmpty(tail(List(1)))  
val res: Boolean = true
```

Next, we look at *recursive* functions for lists. The inductive structure of Scala-lists leads directly to a way for defining recursive functions: the base case deals with the empty list `Nil`; and the recursion step needs to compute the result for a non-empty list `x::xs`, given that the result for the list of remaining elements `xs` is known. As a first example, let us look at a function that computes the *length* of a list of `Int`s. The length of the empty list is 0. Given the length of the remaining list `xs`, the length of `x::xs` is one more:

```
scala> def length(a: List[Int]): Int =  
  |   a match  
  |     case Nil => 0  
  |     case _::xs => length(xs) + 1  
  
scala> length(List(1,2,3))  
val res: Int = 3  
  
scala> length(List())  
val res: Int = 0
```

Here is an example evaluation of `length`:

```
length(1::2::3::Nil)  
= length(2::3::Nil) + 1  
= (length(3::Nil) + 1) + 1  
= ((length(Nil) + 1) + 1) + 1  
= ((0 + 1) + 1) + 1  
= (1 + 1) + 1  
= 2 + 1  
= 3
```

As a second example, we want to find the element at index  $i$  of a given list of **Ints**. The empty list does not have any elements, so we get an error. The element at index  $0$  of a non-empty list  $x::xs$  is the first element  $x$ . Otherwise, the element at index  $i > 0$  in a non-empty list  $x::xs$  is the element at index  $i - 1$  in the remaining list  $xs$ .

```
scala> def elementAt(a: List[Int], i: Int): Int =
|   (a, i) match
|     case (Nil, _) => throw new Exception("Invalid index.")
|     case (_, i) if i < 0 => throw new Exception("Invalid index.")
|     case (x::_, 0) => x
|     case (x::xs, i) => elementAt(xs, i - 1)

scala> elementAt(List(1,2,3), 0)
val res: Int = 1

scala> elementAt(List(1,2,3), 2)
val res: Int = 3

scala> elementAt(List(1,2,3), 4)
java.lang.Exception: Invalid index
```

Here is an example evaluation of `elementAt`:

```
elementAt(1::2::3::Nil, 1)
= elementAt(2::3::Nil, 0)
= 2
```

Finally, we implement a recursive function to *concatenate* two lists  $a$  and  $b$  of **Ints**. Concatenations means that we create a new list that first contains all the elements of  $a$ , followed by all the elements of  $b$ . The recursion is on  $a$ : if  $a$  is empty, the concatenation of  $a$  and  $b$  consists of all the elements in  $b$ . If  $a$  is the non-empty list  $x::xs$ , the concatenation is obtained by first concatenating the remaining list  $xs$  with  $b$ , and putting  $x$  in front of the result.

```
scala> def concat(a: List[Int], b: List[Int]): List[Int] =
|   a match
|     case Nil => b
|     case x::xs => x::concat(xs, b)

scala> concat(List(1,2,3), List(4,5,6))
val res: List[Int] = List(1, 2, 3, 4, 5, 6)
```

Here is an example evaluation of `concat`:

```
concat(1::2::3::Nil, 4::5:6::Nil)
= 1::concat(2::3::Nil, 4::5:6::Nil)
= 1::2::concat(3::Nil, 4::5:6::Nil)
= 1::2::3::concat(Nil, 4::5:6::Nil)
= 1::2::3::4::5:6::Nil
```

Let us now analyze the running time of `concat`. Recall from Chapter 8 that for this, we first need to agree on the elementary operations and the input size:

- **Elementary operations.** We count the number of `cons`-operations  $::$ .
- **Input size.** Let  $I = (a, b)$  be an input. We define input size  $|I|$  of  $I$  as the number of elements in  $a$ . We ignore  $b$  for the input size.

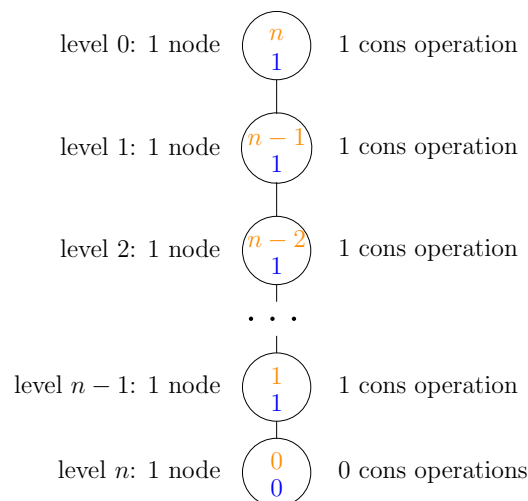
With these definitions, we can apply the techniques for the analysis of recursive algorithms that we saw in Section 8.3.3. We write a recurrence relation for the worst-case running time of `concat`. In the base case, for  $n = 0$ , we do not need any `cons`-operations. In the recursion step, for  $n \geq 1$ , we recurse on a list of length  $n - 1$  and then perform one `cons`-operation. Thus, we get

$$T(0) = 0, \text{ and}$$

$$T(n) = T(n - 1) + 1, \text{ for } n \geq 1.$$

This is a very easy recurrence relation. However, for completeness, let us see how the two techniques from Section 8.3.3 can be applied to obtain a solution.

**The tree method.** The tree diagram for the recursion looks as follows:



The tree has  $n + 1$  levels. The number of `cons`-operations on each of the first  $n$  levels is 1, the number of `cons`-operations on the last level is 0. Thus, the total number of `cons`-operations is  $n$ .

**Repeated application of the recurrence relation.** We repeatedly apply the recurrence relation until we reach the base case:

$$T(n) = T(n - 1) + 1$$

$$= (T(n - 2) + 1) + 1$$

$$\begin{aligned} &= T(n-2) + 2 \\ &= (T(n-3) + 1) + 2 \\ &= T(n-3) + 3 \\ &= \dots \\ &= T(n-\ell) + \ell. \end{aligned}$$

For  $\ell = n$ , we have  $n - \ell = 0$ , and hence  $T(n - \ell) = 0$ . Thus, after  $n$  steps, we reach the recursion base, and we obtain

$$T(n) = T(n - n) + n = 0 + n = n.$$

Hence, we get that the worst-case running time of `concat` is  $T(n) = n = O(n)$ . We note that this running time is the same for all inputs of size  $n$ , and that the running time of `concat` does *not* depend on the length of the second list `b`.

Scala already has implementations of the functions `length`, `elementAt`, and `concat`, but with a slightly different syntax:

```
scala> val a: List[Int] = List(1, 2, 3)
val a: List[Int] = List(1, 2, 3)

scala> a.length
val res: Int = 3

scala> a(1)
val res: Int = 2

scala> a:::List(3, 4, 5)
val res: List[Int] = List(1, 2, 3, 3, 4, 5)
```

### 11.2.1 Polymorphism

By now, we have seen several functions that operate on lists. At this point, however, there is a problem: all of our functions are defined for lists of type `Int`. We did this because we need to give a concrete type in the function signature. But what do we do if we want to compute the length of a list of, say, `Booleans`? Do we need to write a `length`-function that is specific to lists of type `Boolean`, and so on, for every type that we might need to use in a list? Of course, this would be quite inconvenient. To get around this problem, programming languages offer a feature called *polymorphism*.

Polymorphism means that the same symbol or name in a programming language can be used with different types. This should happen in a well-defined manner that ensures the type-safety of the overall program. There are three main types of polymorphism, and we will have a brief look at each of them.

**Ad-hoc polymorphism.** Ad-hoc polymorphism means that we can have multiple functions with the same name, but with different type signatures. In this case, we say

that the function name is *overloaded*. If a call to an overloaded function appears in the program code, the compiler uses the types of the actual parameters to determine which function is called. Here is an example in Scala:

```
def foo(x: Int): Unit = println("This is foo for ints.")

def foo(b: Boolean): Unit = println("This is foo for booleans.")

def foo(c: Char): Unit = println("This is foo for chars.")

@main
def test(): Unit =
  foo(10)
  foo(true)
  foo('a')
```

This program creates the following output.

```
This is foo for ints.
This is foo for booleans.
This is foo for chars.
```

**Attention:** This example does not work in the REPL, because if we define a new function with an existing name in the REPL, the old function with this name will no longer exist.

One main example of ad-hoc polymorphism is *operator overloading*: the same operator, say, `+`, can be used for different data types:

```
scala> 1+1
val res: Int = 2

scala> 1.1 + 1.1
val res: Double = 2.2
```

Here, the first `+` operates on integers, and the second `+` operates on floating-point numbers. Internally, the implementations are completely different, and ad-hoc polymorphism makes it possible that we can use the same name for both of them.

A second main example of ad-hoc polymorphism is in the context of object-oriented programming, where it allows us to have several constructors for a class. This will be discussed in Section 13.4.

Unfortunately, for our present problem of writing list-functions that can work with arbitrary types, ad-hoc polymorphism does not help much. With ad-hoc polymorphism, we would still need to create different functions for lists of different data types. The only advantage would be that we could give the same name to all of these functions.

**Inclusion polymorphism.** Inclusion polymorphism (also called *subtyping*) allows us to introduce *subtype-supertype-relationships* between types: a type *A* can be made a *subtype* of another type *B*. In this case, type *B* is the *supertype* of type *A*. Then, subtypes are *type compatible* with supertypes. That is, a subtype can be used everywhere where a supertype is expected. For example, in Scala, there is a type **Any** that is the supertype of *every* possible type. If we want to write a function that works for every type, we can use the type **Any** in the signature:

```
scala> def giveSnd(a: Any, b: Any): Any = b

scala> giveSnd(true, 'a')
val res: Any = a

scala> giveSnd(100, false)
val res: Any = false
```

A main disadvantage of inclusion polymorphism is that information gets lost. In the example above, we have no way of showing that the return value of `giveSnd` must have the same type as the second parameter. Also, the return value will always be of type **Any**, and we will need to do additional work to use it again as a value of a more useful type, such as **Int**.

Thus, even though inclusion polymorphism does offer a way to solve our problem of writing general list functions, it does have major disadvantages, and will not pursue this solution here. We will learn much more about inclusion polymorphism in Section 13.6, in the context of *inheritance* in object-oriented programming.

**Parametric polymorphism.** In parametric polymorphism, we can introduce one or more *type parameters* for a function. These type parameters stand for arbitrary concrete types for which the function can be used. The advantage of using type parameters is that they document which types should be the same, and that for each concrete invocation of the function, they are replaced by the concrete types. In this way, no information gets lost. Here is how our `giveSnd`-example from the description of inclusion polymorphism looks with type parameters:

```
scala> def giveSnd[A, B](a: A, b: B): B = b

scala> giveSnd(true, 'a')
val res: Char = a

scala> giveSnd(100, false)
val res: Boolean = false
```

In the function definition, we must declare our type parameters in square brackets (in our example, we have chosen the type parameters **A** and **B**). Then, we can use the type parameters in the function signature, just like regular types. Note that when we call the function `giveSnd`, the type parameters are instantiated with the types of

the actual parameters, e.g., `giveSnd(true, 'a')` has return type `Char`, because for this invocation, we have `A = Boolean` and `B = Char`; and `giveSnd(100, false)` has return type `Boolean`, because for this invocation, we have `A = Int` and `B = Boolean`.

Now we can rewrite our list functions with parametric polymorphism, so that they can be used with any type:

```
def head[A](a: List[A]): A =
  a match
    case Nil => throw new Exception("No head.")
    case x::_ => x

def tail[A](a: List[A]): List[A] =
  a match
    case Nil => throw new Exception("No tail.")
    case _::xs => xs

def isEmpty[A](a: List[A]): Boolean =
  a match
    case Nil => true
    case _ => false

def length[A](a: List[A]): Int =
  a match
    case Nil => 0
    case _::xs => length(xs) + 1

def elementAt[A](a: List[A], i: Int): A =
  (a, i) match
    case (Nil, _) => throw new Exception("Invalid index")
    case (_, i) if i < 0 => throw new Exception("Invalid index")
    case (x::_, 0) => x
    case (x::xs, i) => elementAt(xs, i - 1)

def concat[A](a: List[A], b: List[A]): List[A] =
  a match
    case Nil => b
    case x::xs => x::concat(xs, b)
```

Here are two more functions for general lists: The function `take` takes the first `n` elements of a list `a`. If `a` has fewer than `n` elements, then `take` returns the complete list. The function `drop` removes the first `n` elements of a list `a`. If `a` has fewer than `n` elements, it returns the empty list. Both functions are implemented with a straightforward recursion.

```
scala> def take[A](n: Int, a: List[A]): List[A] =
  |   (n, a) match
  |     case (0, _) => Nil
  |     case (_, Nil) => Nil
  |     case (_, x::xs) => x::take(n - 1, xs)
```



```
scala> def drop[A](n: Int, a: List[A]): List[A] =
|       (n, a) match
|         case (0, _) => a
|         case (_, Nil) => Nil
|         case (_, x::xs) => drop(n - 1, xs)

scala> take(2, List(1, 2, 3))
val res: List[Int] = List(1, 2)

scala> drop(2, List(1, 2, 3))
val res: List[Int] = List(3)

scala> take(5, List(1, 2, 3))
val res: List[Int] = List(1, 2, 3)

scala> drop(5, List(1, 2, 3))
val res: List[Int] = List()
```

### 11.2.2 Reversing a list

Now we would like to implement a function that *reverses* a given list *a*. That is, the function should create a new list that contains the elements of *a* in order from the last to the first element. A simple recursion is as follows: if *a* is the empty list *Nil*, the result is *Nil*; if *a* is the non-empty list *x::xs*, we recursively reverse the list *xs* of the remaining elements. Then, we put the first element *x* at the back of the resulting list. For this, we use the concat-operator *::* (we must convert *x* to the one-element list *List(x)* so that *::* can be applied):

```
scala> def reverse[A](a: List[A]): List[A] =
|       a match
|         case Nil => Nil
|         case x::xs => reverse(xs):::List(x)

scala> reverse(List(1, 2, 3))
val res: List[Int] = List(3, 2, 1)
```

Here is an example evaluation of *reverse*:

```
reverse(1::2::3::Nil)
= reverse(2::3::Nil):::List(1)
= (reverse(3::Nil):::List(2)):::List(1)
= ((reverse(Nil):::List(3)):::List(2)):::List(1)
= ((Nil:::List(3)):::List(2)):::List(1)
= (List(3):::List(2)):::List(1)
= List(3, 2):::List(1)
= List(3, 2, 1)
```

Let us now analyze the running time of `reverse`. We use the following conventions for the elementary operations and the input size:

- **Elementary operations.** We count the number of cons-operations  $::$ .
- **Input size.** The input size is the number of elements in the input list `a`.

Now, we apply the techniques for the analysis of recursive algorithms that we saw in Section 8.3.3. We write a recurrence relation for the worst-case running time of `reverse`. In the base case, for  $n = 0$ , we do not need any cons-operations. In the recursion step, for  $n \geq 1$ , we recurse on a list of length  $n-1$ . Then, we create a list `List(x)` with one element, and we apply the concat-operator to add this list to the recursively constructed list `reverse(xs)`. To create `List(x)`, we need one cons-operation. To concatenate this list with `reverse(xs)`, we need  $n-1$  additional cons-operations, because `xs` has  $n-1$  elements, and because in Section 11.2, we saw that the number of cons-operations for the concat-operator equals the length of the first list. Thus, we get the following recurrence relations:

$$T(0) = 0, \text{ and}$$
$$T(n) = T(n-1) + n, \text{ for } n \geq 1.$$

This is almost the same recurrence relation as for the worst-case running time of quicksort from Section 8.3.4, except that the quicksort recurrence had the base case  $T(1) = 0$  instead of  $T(1) = 1$ . This is just an additive difference of 1, and we get that the worst-case running time  $T(n)$  of `reverse` is  $T(n) = n^2/2 + n/2 = O(n^2)$ .

This is not very efficient. Fortunately, we can use tail recursion with the accumulator technique to obtain a much faster `reverse`-function. The idea is that after  $i$  steps, the accumulator contains the reversal of the first  $i$  elements in the input list `a`. To process the  $(i+1)$ -th element of the input list, we need to obtain the reversal of the first  $i+1$  elements of the input list. For this, it suffices to put the  $(i+1)$ -th element of `a` in front of the accumulator. This requires a single cons-operation:

```
scala> def reverse2[A](a: List[A]): List[A] =  
  |   def step(acc: List[A], b: List[A]): List[A] =  
  |     b match  
  |       case Nil => acc  
  |       case x::xs => step(x::acc, xs)  
  |   step(Nil, a)  
  
scala> reverse2(List(1,2,3))  
val res: List[Int] = List(3, 2, 1)
```

To analyze the running time of `reverse2`, we must analyze the running time of `step`.

- **Elementary operations.** We count the number of cons-operations  $::$ .
- **Input size.** Let  $I = (acc, b)$  be an input for `step`. The input size is  $|I| = b.length$ , the number of elements in the list `b`. The length of `acc` does not count for the input size.

The base case of step does not use any cons-operations. In the recursion step, we perform one cons-operation, and then we recurse on a list of size  $n - 1$ . Thus, we get the following recurrence relation:

$$T(0) = 0, \text{ and} \\ T(n) = T(n - 1) + 1, \text{ for } n \geq 1.$$

In Section 11.2, we saw that this solves to  $T(n) = n = O(n)$ , which is much faster. Here is an example evaluation of reverse2:

```
reverse2(1::2::3::Nil)
= step(Nil, 1::2::3::Nil)
= step(1::Nil, 2::3::Nil)
= step(2::1::Nil, 3::Nil)
= step(3::2::1::Nil, Nil)
= 3::2::1::Nil
```

### 11.3 Higher-order functions

In Scala, functions are treated as *values*. In particular, this means that we can apply operations to functions and that we can use functions as parameters or return values of other functions. A function that has another function as an input parameter or an output value (or both) is called a *higher-order function*.

The following function takes a function  $f$  as parameter and creates a function that is obtained by applying  $f$  twice:

```
scala> def applyTwice[A](f: A => A): A => A = x => f(f(x))
def applyTwice[A](f: A => A): A => A
```

The parameter and the return value of applyTwice have type  $A \Rightarrow A$ . This type represents a function that expects a value of type  $A$  and returns a value of type  $A$ . The function body of applyTwice is  $x \Rightarrow f(f(x))$ . This is called a  *$\lambda$ -expression* or an *anonymous function*. It defines a function that has as input the variable  $x$  and computes the expression  $f(f(x))$ .  $\lambda$ -expressions are a way to directly define function values, without using **def**.

Let us now see how applyTwice can be used. For this, consider the following two functions:

```
scala> def addOne(x: Int); Int = x + 1

scala> def negate(x: Boolean): Boolean = !x
```

We can now use applyTwice to obtain two new functions as follows:

```
scala> val addTwo: Int => Int = applyTwice(addOne)
val addTwo: Int => Int = Lambda/0x00007bad43561400@2403c1c5

scala> val identity: Boolean => Boolean = applyTwice(negate)
val identity: Boolean => Boolean = Lambda/0x00007bad43561400@38447073

scala> addTwo(1)
val res: Int = 3

scala> identity(true)
val res: Boolean = true
```

Note that `addTwo` and `identity` are function *values*. They are obtained by applying the higher-order function `applyTwice` to the functions `addOne` and `negate`. After they have been defined, the function values can be used like functions that were declared with `def`.

Currying.

### 11.3.1 Higher-order functions for lists

Higher-order functions go together very well with functional lists. They give us a very elegant and universal method for processing lists. Let us look at a few examples.

**dropWhile and takeWhile.** In Section 11.2.1, we saw the functions `take` and `drop`. These functions allow us to take or remove a given number `n` of elements from the beginning of an input list `a`. This time, instead of a number of elements `n`, we would like to specify a *predicate* `p`, i.e., a function `p` of type `A => Boolean`. Then, we want to take or drop elements as long as `p` returns `true`. The functions are implemented with a simple recursion: for the empty list `Nil`, the result is the empty list. For the non-empty list `x::xs`, we check whether the first element `x` fulfills `p`. If so, we take or drop `x`, and we continue with the remaining list `xs`. If not, we stop the recursion. The implementation looks as follows:

```
scala> def takeWhile[A](p: A => Boolean, a: List[A]): List[A] =
|   a match
|     case Nil => Nil
|     case x::xs if p(x) => x::takeWhile(p, xs)
|     case _ => Nil

scala> def dropWhile[A](p: A => Boolean, a: List[A]): List[A] =
|   a match
|     case Nil => Nil
|     case x::xs if p(x) => dropWhile(p, xs)
|     case _ => a
```

```
scala> takeWhile((x: Int) => x % 2 == 0, List(2, 4, 6, 1, 10, 7))
val res: List[Int] = List(2, 4, 6)

scala> dropWhile((x: Int) => x % 2 == 0, List(2, 4, 6, 1, 10, 7))
val res: List[Int] = List(1, 10, 7)
```

Here,  $(x: \text{Int}) \Rightarrow x \% 2 == 0$  is again a  $\lambda$ -expression. In this case, we explicitly specify the type of the function parameter  $x$  to be `Int`.

**filter.** One of the most common higher-order functions for lists is the function `filter`. It receives a predicate  $p: A \Rightarrow \text{Boolean}$  and an input list  $a$  of type  $A$ , and it returns the sublist of  $a$  that contains all elements in  $a$  that fulfill  $p$ . It is implemented as follows:

```
scala> def filter[A](p: A => Boolean, a: List[A]): List[A] =
|   a match
|     case Nil => Nil
|     case x::xs if p(x) => x::filter(p, xs)
|     case _::xs => filter(p, xs)

scala> filter((_: Int) % 2 == 0, List(2, 4, 6, 1, 10, 7))
val res: List[Int] = List(2, 4, 6, 10)
```

Here,  $(_: \text{Int}) \% 2 == 0$  is another a  $\lambda$ -expression, now using the *wildcard notation*, where we do not use an explicit parameter  $x$ , but a wildcard  $_: \text{Int}$  to represent an anonymous parameter of type `Int` that occurs only once in the function expression.

The function `filter` is already implemented in Scala. To use it, we can use the following syntax:

```
scala> List(2, 4, 6, 1, 10, 7).filter((_: Int) % 2 == 0)
val res: List[Int] = List(2, 4, 6, 10)
```

**map.** Another common higher-order function is `map`. It receives a function  $f: A \Rightarrow B$  and an input list  $a$  of type  $B$ , and it returns the list that is obtained by applying  $f$  to every element in  $a$ . It is implemented as follows:

```
scala> def map[A, B](f: A => B, a: List[A]): List[B] =
|   a match
|     case Nil => Nil
|     case x::xs => f(x)::map(f, xs)

scala> map((x: Int) => x*x, List(1, 2, 3, 4, 5))
val res: List[Int] = List(1, 4, 9, 16, 25)
```

The function `map` is already implemented in Scala. To use it, we can use the following syntax:

```
scala> List(1, 2, 3, 4, 5).map((x: Int) => x*x)
val res: List[Int] = List(1, 4, 9, 16, 25)
```

**fold.** *Folding* provides a way to implement simple recursive patterns with a single higher-order function, without having to go through the routine steps of the recursion definition. In the previous sections, we have seen a general pattern for defining simple recursive functions on lists: first, we define a base case for the empty list `Nil`. Then, we define a recursion step for the non-empty list `x::xs`. For this, we need to provide an expression that computes the desired result, given the first element `x` and the desired result for the remaining list `xs`. Let us see three more examples of this general pattern:

```
scala> def sum(a: List[Int]): Int =
  |   a match
  |     case Nil => 0
  |     case x::xs => x + sum(xs)

scala> def prod(a: List[Int]): Int =
  |   a match
  |     case Nil => 1
  |     case x::xs => x * prod(xs)

scala> def forAll[A](p: A => Boolean, a: List[A]): Boolean =
  |   a match
  |     case Nil => true
  |     case x::xs => p(x) && forAll(p, xs)

scala> sum(List(1, 2, 3, 4))
val res: Int = 10

scala> prod(List(1, 2, 3, 4))
val res: Int = 24

scala> forAll((_: Int) > 0, List(1, 2, 3, 4))
val res: Boolean = true
```

The function `sum` computes the sum of a list of integers; the function `prod` computes the product of a list of integers; and the function `forAll` checks whether all elements of a list satisfy a given predicate.

For each function, the overall recursion pattern is essentially the same. The differences lie in the base cases (for `sum`, the base case is `0`; for `prod`, the base case is `1`; and for `forAll`, the base case is `true`); and in the precise expression for the recursion step (`x + sum(xs)` for `sum`; `x * prod(xs)` for `prod`; and `p(x) && forAll(p, xs)` for `forAll`). When writing such functions, we will probably resort to copy-and-paste for most of the function, changing only the expressions for the base case and the recursion step.

To simplify this, we can define a higher-order function `foldr` (*fold right*). The function `foldr` receives a value for the base case and a function for the recursion step,

and it turns it into a general recursive function for lists. The implementation looks as follows:

```
scala> def foldr[A, B](f: (A, B) => B, e: B, a: List[A]): B =  
  |   a match  
  |     case Nil => e  
  |     case x::xs => f(x, foldr(f, e, xs))
```

In the implementation of `foldr`, we recognize the general recursion pattern. The parameter `e` represents the value for the base case. The function `f` represents the recursion step: the first parameter for `f` is the first element `x` of the non-empty list, the second parameter is the result of the recursive call for the list of remaining elements `xs`. Using `foldr`, we can directly implement our three functions without having to write the recursion explicitly. Instead, it suffices to provide the value for the base case and the expression for the recursion step:

```
scala> def sum2(a: List[Int]): Int = foldr((_: Int) + ( _: Int), 0, a)  
  
scala> def prod2(a: List[Int]): Int = foldr((_: Int) * ( _: Int), 1, a)  
  
scala> def forAll2[A](p: A => Boolean, a: List[A]): Boolean =  
  |   foldr(p(_: A) && ( _: Boolean), true, a)
```

The expression `(_: Int) + ( _: Int)` is a  $\lambda$ -expression that defines an anonymous function with *two* parameters of type `Int`, using the wildcard-notation (note that each wildcard represents a new parameter). Unravelling the recursion, the `foldr`-function computes the following expression for an input list  $x_0, x_1, \dots, x_{n-1}$ :

$$f(x_0, f(x_1, \dots, f(x_{n-1}, e) \dots)).$$

This explains the name *fold right*, since the expression is evaluated from right to left. Now it is a nice activity to implement the recursive functions that we have seen so far using `foldr`:

```
// Compute the length of a list.  
scala> def length[A](a: List[A]): Int = foldr((_: A, n: Int) => n + 1, 0, a)  
  
// Concatenate two lists.  
scala> def concat[A](a: List[A], b: List[A]): List[A] =  
  |   foldr((_: A)::(_: List[A]), b, a)  
  
// take while  
scala> def takeWhile[A](p: A => Boolean, a: List[A]): List[A] =  
  |   foldr((x: A, rs: List[A]) => if p(x) then x::rs else Nil, Nil, a)  
  
// Reverse a list.  
scala> def reverse[A](a: List[A]): List[A] =  
  |   foldr((x: A, rs: List[A]) => rs::List(x), Nil, a)
```

```
// Filter.
scala> def filter[A](p: A => Boolean, a: List[A]): List[A] =
  |   foldr((x: A, rs: List[A]) => if p(x) then x::rs else rs, Nil, a)

scala> def map[A, B](f: A => B, a: List[A]): List[B] =
  |   foldr(f(_: A)::(_: List[B]), Nil, a)

// Compute the factorial function.
scala> def fac(n: Int): Int = foldr((_: Int) * (_: Int), 1, (1 to n).toList)
```

We can proceed similarly for tail-recursive functions with the accumulator technique. Here, we need an initial value for the accumulator, and we need a function that updates the accumulator, given the old value of the accumulator and the next element in the list. The `foldl`-function implements the general pattern for a tail-recursive function with an accumulator:

```
scala> def foldl[A, B](f: (B, A) => B, e: B, a: List[A]): B =
  |   a match
  |     case Nil => e
  |     case x::xs => foldl(f, f(e, x), xs)
```

Unravelling the recursion, the `foldl`-function computes the following expression for an input list  $x_0, x_1, \dots, x_{n-1}$ :

$$f(\dots f(f(f(e, x_0), x_1), \dots, x_{n-1})).$$

This explains the name *fold left*, since the expression is evaluated from left to right. Here we implement our tail-recursive functions with `foldl`:

```
scala> def fac2(n: Int): Int =
  |   foldl((_: Int) * (_: Int), 1, (1 to n).toList)

def fib2(n: Int): Int =
  |   foldl((p: (Int, Int), x: Int) => (p(0) + p(1), p(0)),
  |       (0, 1), List.fill(n-1)(0))

scala> def reverse2[A](a: List[A]): List[A] =
  |   foldl((rs: List[A], x: A) => x::rs, Nil, a)
```

Both `foldr` and `foldl` are already implemented in Scala. The syntax is as follows:

```
lscala> List(1, 2, 3, 4).foldRight(0)((_: Int) + (_: Int))
val res: Int = 10

scala> List(1, 2, 3, 4).foldLeft(1)((_: Int) * (_: Int))
val res: Int = 24
```



**Attention:** If you look at Scala's implementations of the fold-functions, you will see that they are not recursive. Instead, they are based on loops. The advantage is that loops are more efficient, since they do not use a recursion stack. However, this chapter is about functional programming. Thus, we show the recursive implementations that follow the functional paradigm. Furthermore, we believe that the recursive implementations are easier to understand and explain the idea of folding better.

ADVANCED COMMENT: At this point, you may wonder what happened to the functions `elementAt`, `take`, `drop`, and `dropWhile`. For these functions, the recursion pattern is more involved, so their implementation with the folding functions is not as straightforward as for the other functions. However, it is still possible to implement them with folding. We give the implementations for the interested reader, without going into further details:

```
scala> def elementAt[A](a: List[A], i: Int): A =
  |   def step(x: A, f: Int => A): Int => A =
  |     (j : Int) => if j == 0 then x else f(j - 1)
  |   foldr(step, (_: Int) => throw new Exception("Invalid index."), a)(i)

scala> def take[A](n: Int, a: List[A]): List[A] =
  |   def step(x: A, f: Int => List[A]): Int => List[A] =
  |     (j : Int) => if j == 0 then Nil else x::f(j - 1)
  |   foldr(step, (_: Int) => Nil, a)(n)

scala> def drop[A](n: Int, a: List[A]): List[A] =
  |   def step(x: A, f: Int => List[A]): Int => List[A] =
  |     (j : Int) => if j == 0 then x::f(0) else f(j - 1)
  |   foldr(step, (_: Int) => Nil, a)(n)

scala> def dropWhile[A](p: A => Boolean, a: List[A]) : List[A] =
  |   def step(x: A, f: Boolean => List[A]): Boolean => List[A] =
  |     (b: Boolean) => if b || !p(x) then x::f(true) else f(false)
  |   foldr(step, (_: Boolean) => Nil, a)(false)
```

We leave it to the interested reader to judge whether these implementations are an improvement.

## 11.4 Functional implementations of sorting algorithms

Finally, we show how to implement the sorting algorithms from Chapter 7 according to the functional paradigm. We focus only on insertion sort and quicksort, and we leave it to you to implement selection sort or merge sort in a functional manner.

**Attention:** Since functional lists are immutable, our functional sorting algorithms cannot sort in-place. Instead, the sorting algorithms will return new lists that contain

the elements of the original list in sorted order. Thus, the functional sorting algorithms have no effect, but a result (this is exactly what *referential transparency* means).

### 11.4.1 Insertion sort

Let us start with insertion sort, see Section 7.3. In insertion sort, the idea was to successively insert the next element of the input list into a list that is already sorted. Thus, we first implement a function `insert` that inserts an integer `y` into a sorted list `a` of integers. The function `insert` uses recursion: if `a` is empty, the result is the list that contains only `y`. If `a` is the non-empty list `x::xs`, we compare `y` with `x`. If `y` is smaller than `x`, then all elements of `a` are larger than `y`, and we put `y` in front of `a`. Otherwise, if `x` is smaller than or equal to `y`, we put `x` in front, and we insert `y` recursively into the list of remaining elements `xs`. Here is the code:

```
scala> def insert(a: List[Int], y: Int): List[Int] =  
  |   a match  
  |     case Nil => List(y)  
  |     case x::_ if y < x => y::a  
  |     case x::xs => x::insert(xs, y)
```

Next, we implement insertion sort with the help of the `insert`-function. We use tail recursion with the accumulator technique, where the accumulator represents the sorted prefix of the input list.

```
scala> def insertionSort(a: List[Int]): List[Int] =  
  |   def step(acc: List[Int], b: List[Int]): List[Int] =  
  |     b match  
  |       case Nil => acc  
  |       case x::xs => step(insert(acc, x), xs)  
  |   step(Nil, a)  
  
scala> insertionSort(List(4,7,2,1,-1,6,3))  
val res: List[Int] = List(-1, 1, 2, 3, 4, 6, 7)
```

Since the recursion pattern of `insertionSort` matches the standard pattern for tail recursion with the accumulator technique, we can also write it in one line, using `fold left` (see Section 11.3).

```
scala> def insertionSort2(a: List[Int]): List[Int] = a.foldLeft(Nil)(insert)  
def insertionSort2(a: List[Int]): List[Int]  
  
scala> insertionSort2(List(4,7,2,1,-1,6,3))  
val res: List[Int] = List(-1, 1, 2, 3, 4, 6, 7)
```

We invite the reader to compare this implementation of insertion sort to the imperative implementation in Section 7.3.

### 11.4.2 Quicksort

Next, we implement quicksort (see Section 7.5). Here, we can directly write down the algorithmic idea using higher-order functions. This leads to a very concise and intuitive implementation of the algorithm.

```
scala> def quicksort(a: List[Int]): List[Int] =  
  |   a match  
  |     case Nil => Nil  
  |     case x::xs => quicksort(xs.filter(_<x)) ::: List(x)  
  |                   ::: quicksort(xs.filter(_>=x))  
  
scala> quicksort(List(4,7,2,1,-1,6,3))  
val res: List[Int] = List(-1, 1, 2, 3, 4, 6, 7)
```

### Functional Data Types

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP3** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *funktionaler Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und -paradigmen.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP7** Du entwickelst Algorithmen zur Lösung vorgegebener *algorithmischer Listen- und Baumprobleme* und stellst diese mithilfe unterschiedlicher Programmierkonzepte in *Scala und Python* dar.



#### 12.1 Type classes

Let us go back to our functional implementation of sorting algorithms from Section 11.4. You may have wondered why we implemented our sorting algorithms for lists of type `Int`, and not for general lists. Here is again our implementation of insertion sort from Section 11.4.1.

```
scala> def insert(a: List[Int], y: Int): List[Int] =
  |   a match
  |     case Nil => List(y)
  |     case x::_ if y < x => y::a    //(*)
  |     case x::xs => x::insert(xs, y)

scala> def insertionSort(a: List[Int]): List[Int] =
  |   def step(acc: List[Int], b: List[Int]): List[Int] =
  |     b match
```

```
|
|         case Nil => acc
|         case x::xs => step(insert(acc, x), xs)
| step(Nil, a)
```

This choice may seem strange. After all, in Section 11.2.1, we introduced *parametric polymorphism*, and we showed how to implement our list functions for a general types. However, in the case of sorting, we cannot use parametric polymorphism directly. The reason is that insertionSort does not work for an *arbitrary* type. It only works for types that represent values from a *totally ordered universe*. Technically, we can see this in the second case of insert (line (\*)): there, we need to apply the *comparison operator*  $x < y$  to determine where  $y$  should go in the input list. Thus, insert (and hence insertionSort) does not work for arbitrary types, but only for types that provide a “<”-operator.

This problem can be solved with *type classes*. A type class represents a set of types that provide a certain common set of operations.<sup>1</sup> In other words, a type class collects operations that each type of the class must implement. We will consider two examples.

### 12.1.1 The type class Ordering

The type class **Ordering** collects types that provide a compare-operation. The compare-operation can be used to determine the order relation between two values of a type. Most of the types that we have seen so far belong to **Ordering**, for example, **Char**, **Boolean**, **String**, the number types, and many more.

Each type in the type class **Ordering** must implement the following function:

```
// Precondition: None
// Postcondition:
//   Effect: None
//   Result: A negative number is returned if a is smaller.
//           A positive number is returned if a is larger.
//           0 is returned if a and b are equal.
def compare(a: A, b: A): Int
```

Here are some examples of how we can use compare for some types in **Ordering**.

```
scala> Ordering[Double].compare(3.4, 2.3)
val res: Int = 1           // i.e., 3.4 > 2.3

cala> Ordering[String].compare("Hallo", "Hello")
val res: Int = -4          // i.e., "Hallo" < "Hello"

scala> Ordering[Boolean].compare(false, false)
val res: Int = 0
```

<sup>1</sup>In Scala, a type class is implemented using the notion of a **trait**. We will talk more about traits in Section 14.1

Now, we can use the type class **Ordering** to implement a polymorphic insertion sort in Scala. The idea is to use parametric polymorphism, and to *restrict* the type variable **A** to **Ordering**.

```
scala> def insert[A: Ordering](a: List[A], y: A): List[A] =
|   a match
|     case Nil => List(y)
|     case x::_ if Ordering[A].compare(y, x) < 0 => y::a //(*)
|     case x::xs => x::insert(xs, y)

scala> def insertionSort[A: Ordering](a: List[A]): List[A] =
|   def step(acc: List[A], b: List[A]): List[A] =
|     b match
|       case Nil => acc
|       case x::xs => step(insert(acc, x), xs)
|   step(Nil, a)

scala> insertionSort(List(4,2,7,6,10))
val res: List[Int] = List(2, 4, 6, 7, 10)

scala> insertionSort(List('k','o','n','z','e','p','t'))
val res: List[Char] = List(e, k, n, o, p, t, z)
```

The restriction is implemented by the syntax `[A: Ordering]`. This means: `A` is an arbitrary type from the type class `Ordering`. The operation `compare` in line `(*)` can only be used because `A` belongs to `Ordering`.

However, the code would be much more readable if we could to use `y < x` instead of the very unwieldy expression `Ordering[A].compare(y, x) < 0`. In many cases, Scala gives us the opportunity to generate the corresponding operators. This can be achieved as follows:

```
scala> def insert[A: Ordering](a: List[A], y: A): List[A] =
|   val ord = summon[Ordering[A]]
|   import ord.mkOrderingOps
|   a match
|     case Nil => List(y)
|     case x::_ if y < x => y::a
|     case x::xs => x::insert(xs, y
```

At this point, you may wonder how it is possible to implement a polymorphic sorting algorithm in Python (our Python implementations in Chapter 7 are actually polymorphic). There are two reasons: first, Python is dynamically-typed, so the input list to the sorting algorithm can be of arbitrary type; second, Python uses *duck typing*:<sup>2</sup> when we use the comparison operator in a sorting algorithm, Python checks whether such an operator exists for the data object at hand. If so, Python uses this operator, and everything is fine. If not, Python produces a runtime error. This is different from

<sup>2</sup>Inspired by the *duck test*: “If it walks like a duck, swims like a duck, and quacks like a duck, then it must be a duck.”

the static type system of Scala. Here, it does not suffice that the comparison operator *exists* for a given type **A**. It is also necessary that someone explicitly puts **A** into the type class **Ordering** (we will see in Section 12.3 how this is done).

## 12.1.2 The type class **Numeric**

Another important type class is called **Numeric**. This type class contains all the classic number data types (including **Char**). **Numeric** provides basic arithmetic such as **+**, **-**, or **\***, but also a few conversion operations. More precisely, a type **A** in **Numeric** must provide the following operations:

```
def compare(a: A, b: A): Int
def fromInt(x: Int): A
def minus(x: A, y: A): A
def negate(x: A): A
def parseString(str: String): Option[A]
def plus(x: A, y: A): A
def times(x: A, y: A): A
def toDouble(x: A): Double
def toFloat(x: A): Float
def toInt(x: A): Int
def toLong(x: A): Long
```

We are not going to discuss these functions and their specifications in detail. Instead, we want to briefly mention two important issues:

- The type class **Numeric** prescribes a **compare**-method. In fact, every type in **Numeric** automatically also belongs to **Ordering**.
- In **Numeric**, there is no division-function. In fact, the division function is defined in other type classes, like, e.g., **Integral** or **Fractional**. A pure **Numeric**-type does not need to have a way to divide elements.

The type classes define a of hierarchical structure.<sup>3</sup> Our examples are structured as follows: **Numeric** is a subset of **Ordering**. **Integral** and **Fractional** are subsets of **Numeric**. The data types **Int** or **Long** are elements of **Integral**, while **Double** and **Float** are elements of **Fractional**.

As in **Ordering**, we can import syntax to use **+** or **\*** instead of **plus** or **times**.

```
val num = summon[Numeric[A]]
import num.mkNumericOps
```

Next, in Section 12.2, we discuss how we can create our own data types and how we can put them into type classes like **Ordering** or **Numeric**.

<sup>3</sup>You may have a look at the graph at <https://www.scala-lang.org/api/3.6.3/scala/math/Numeric.html>.

## 12.2 Algebraic data types

Up to now, we have only used predefined data types or (in the exercises) *type aliases* like

```
type Time = (Int, Int, Int)
```

Now, we want to define *our own* data types. In Scala, we have at least two possibilities to do this: we can use object-oriented programming concepts like classes or traits—we will discuss this in Chapter 13—or we can use *algebraic data types*—a concept of functional programming. An algebraic data type is a completely new type that is obtained through the structured combination of other types.

As a first example, consider the following data type **Direction**. It represents the four main compass directions:

```
scala> enum Direction:
  | case N, E, S, W
```

This defines a new algebraic data type **Direction**. It has four *constructors*, namely **N**, **E**, **S**, and **W**. We can think of the constructors as defining the different values that data objects of type **Direction** can have. Thus, in its simplest form, an algebraic data type just consists of a name (e.g., **Direction**), and of a list of possible values (e.g., **N**, **E**, **S**, and **W**). To use the different values of an algebraic data type **A**, we need to use the syntax **A**.<**Constructor**>.

```
scala> Direction.N
val res: Direction = N

scala> Direction.E
val res: Direction = E
```

By writing **import A.\***, we can avoid the type name.

```
scala> import Direction.*

scala> N
val res: Direction = N

scala> E
val res: Direction = E
```

In functions, we can use pattern matching to access the different possible values of an algebraic data type. For example, the following function inverts a compass direction:



```
scala> def invert(d: Direction): Direction =
  |   import Direction.*
  |   d match
  |     case N => S
  |     case S => N
  |     case E => W
  |     case W => E
  |
scala> invert(N)
val res: Direction = S
```

### 12.2.1 Parametrized algebraic data types

The constructors of an algebraic data type may also have *parameters*. These parameters are values of a certain data type that are stored with the constructor and that are part of the value of the algebraic data type.

In the next example, we have an algebraic data type **Shape** that represents geometric shapes. It has two constructors: **Rectangle** and **Circle**. With each rectangle, we store its width and its height; with each circle, we store its radius:

```
scala> enum Shape:
  |   case Rectangle(width: Double, height: Double)
  |   case Circle(radius: Double)
  |
scala> var s: Shape = Shape.Rectangle(1, 2)
var s: Shape = Rectangle(1.0, 2.0)
scala> var c: Shape = Shape.Circle(100)
var c: Shape = Circle(100.0)
```

Again, we can use pattern matching to access the constructors and parameters of an algebraic data type. For example, the following function computes the area of a geometric shape *s*:

```
scala> def area(s: Shape): Double =
  |   import Shape.*
  |   s match
  |     case Rectangle(w, h) => w*h
  |     case Circle(r) => math.Pi*r*r
  |
scala> area(s)
val res: Double = 2.0
scala> area(c)
val res: Double = 31415.926535897932
```

### 12.2.2 Polymorphic algebraic data types

We can use parametric polymorphism in order to use a general data type as a parameter for an algebraic data type. The following data type represents the possibility that a variable might have a value of type **A**, or not.

```
scala> enum Option[+A]:
  |   case None
  |   case Some(value: A)
```

This algebraic data type is already implemented in Scala. It is useful when defining a *partial* function, i.e., a function that does not always have a return value. For example, this is the case for a function that opens a file or a function that divides two numbers:

```
scala> def divide(a: Int, b: Int): Option[Int] =
  |   import Option.*
  |   if b == 0 then None else Some(a/b)

scala> divide(10, 5)
val res: Option[Int] = Some(2)

scala> divide(10, 0)
val res: Option[Int] = None
```

**Attention:** You may wonder why we need to write **Option[+A]** instead of **Option[A]**. In fact, if we omit the “+” in front of the type variable **A**, we get the following mysterious error message:

```
scala> enum Option[A]:
  |   case None
  |   case Some(value: A)
  |
-- Error: -----
2 |   case None
  |   ^^^^^^^^^
  |   cannot determine type argument for enum parent class Option,
  |   type parameter type A is invariant
```

The reason for this error message (and for the need to write “[+A]” instead of “[A]” is that Scala does not support algebraic data types directly. Instead, Scala translates an algebraic data type into objects and classes, constructs from the world of object oriented programming (see Chapter 13). One side-effect of this translation is that all instances of the algebraic data type **Option[A]** will have the same **None** value (i.e., **None** is the same for **Option[Int]**, **Option[Cha]**, **Option[String]**, etc.) For this to be possible, Scala relies on the *inclusion polymorphism* of object-oriented programming. This inclusion polymorphism makes it possible to give a single type to **None** that works for all instan-

ces of `Option[A]`. But which one? There are two candidates: `None: Option[Nothing]`, the smallest possible type, or `None: Option[Any]`, the largest possible type. By writing `[+A]`, we indicate that Scala should use `Option[Nothing]`, the smallest possible type:

```
scala> enum Option[+A]:
  |   case None
  |   case Some(value: A)

scala> Option.None
val res: Option[Nothing] = None
```

This is the right choice for immutable functional data types.

### 12.2.3 Recursive algebraic data types

Algebraic data types can be *recursive*. That is, when defining a constructor of an algebraic data type `A`, we can use `A` as the type of a parameter. In this way, an algebraic data type can refer to itself in its constructors. Of course, we also need some constructors that do not have a value of type `A` as a parameter. These constructors serve as the *base cases* of the recursion. Values of a recursive algebraic data type can be visualized as tree-like diagrams.

As an example, we want to implement an algebraic data type that represents *natural numbers*. In “Diskrete Strukturen für Informatik”, you learned that natural numbers are characterized by the *Peano axioms*. In particular, the Peano axioms state that the natural numbers can be constructed with two ingredients: the smallest natural number 0, and the *successor function*  $n \mapsto n + 1$ . We have the following inductive definition of the natural numbers:

- **Zero.** 0 is a natural number.
- **Successor.** If  $n$  is a natural number, then  $(n + 1)$  is a natural number. We say that  $(n + 1)$  is the *successor* of  $n$ .

For example, “3” is a symbol that represents the natural number  $((0 + 1) + 1) + 1$ , the “successor of the successor of the successor of 0”. We can implement this inductive definition directly as a recursive algebraic data type. `Nat`:

```
scala> enum Nat:
  |   case Zero
  |   case PlusOne(n: Nat)
```

This is a recursive algebraic data type, because the constructor `PlusOne` has a parameter of type `Nat`. The base case of the recursion is given by the constructor `Zero`. Then, the natural number 3 as represented as the value `PlusOne(PlusOne(PlusOne(Zero)))` of `Nat`.

```
scala> import Nat.*

scala> PlusOne(PlusOne(PlusOne(Zero)))
val res: Nat = PlusOne(PlusOne(PlusOne(Zero)))
```

To implement functions on recursive algebraic data types, we can use pattern matching together with recursion. For example, here is a function to convert a **Nat**-number to an integer.

```
scala> def toInt(n : Nat): Int =
|   import Nat.*
|   n match
|     case Zero => 0
|     case PlusOne(m) => toInt(m) + 1

scala> toInt(PlusOne(PlusOne(PlusOne(Zero))))
val res: Int = 3
```

We can also do it with tail recursion:

```
scala> def toInt2(n : Nat): Int =
|   import Nat.*
|   def step(acc: Int, m: Nat): Int =
|     m match
|       case Zero => acc
|       case PlusOne(o) => step(acc + 1, o)
|   step(0, n)

scala> toInt2(PlusOne(PlusOne(PlusOne(Zero))))
val res: Int = 3
```

The following function converts from **Int** to **Nat**.

```
scala> def fromInt(n : Int): Nat =
|   import Nat.*
|   n match
|     case 0 => Zero
|     case _ => PlusOne(fromInt(n - 1))

scala> fromInt(3)
val res: Nat = PlusOne(PlusOne(PlusOne(Zero)))
```

Finally, let us see two more functions: one computes the sum of two **Nat**-numbers, and the other one compares whether two **Nat**-numbers are equal.

```
scala> def plus(n: Nat, m: Nat): Nat =
|   n match
|     // base case: 0 + m = m
```

```
|      case Zero => m
|      // step: (o + 1) + m = (o + m) + 1
|      case PlusOne(o) => PlusOne(plus(o, m))
|
scala> def equals(n: Nat, m: Nat): Boolean =
|      (n, m) match
|      | case (Zero, Zero) => true
|      | case (PlusOne(o), PlusOne(p)) => equals(o, p)
|      | case _ => false
```

We could go on and define further functions for multiplication, subtraction, power, etc. Even more, we can use these functions as *mathematical definitions* of addition, multiplication, etc., and we can start to prove well-known arithmetic laws, such as the neutrality of **Zero** (that is,  $\text{plus}(n, \text{Zero}) == \text{plus}(\text{Zero}, n) == n$ ) or the commutativity and associativity of addition (that is,  $\text{plus}(n, m) == \text{plus}(m, n)$  as well as  $\text{plus}(\text{plus}(n, m), o) == \text{plus}(n, \text{plus}(m, o))$ ). In fact, this process mirrors how the natural numbers are constructed in mathematics from the axioms of set theory (we refer the interested reader to later classes on this topic).

### 12.2.4 Lists

We can also combine parametric polymorphism and recursion in an algebraic data type. As an example, we show that functional lists are actually a specific algebraic data types. Recall our recursive definition of functional lists from Section 11.1.

- **The empty list.** The *empty list* **Nil** is a list of type **T**.
- **Putting a new element in front of a list.** Let  $x$  be a value of type **T**, and let  $xs$  be a list of type **T**. Then  $x :: xs$  is also a list of type **T**. Here,  $::$  is called the *cons-operator*.

This definition can be implemented directly as an algebraic data type. Since **List**, **Nil**, and  $::$  are already defined in Scala, we use different names for our own implementation, but the meaning is the same.

```
scala> enum MyList[+A]:
|   case Empty
|   case Cons(x: A, xs: MyList[A])
```

Now we can reimplement some of the list functions from Chapter 11 using our own algebraic data type **MyList**. Except for the different names for the type constructors, there is absolutely no difference to the previous implementations.

```
scala> def length[A](a: MyList[A]): Int =
|   import MyList.*
|   a match
|   | case Empty => 0
```

```

|         case Cons(x, xs) => length(xs) + 1

scala> def concat[A](a: MyList[A], b: MyList[A]): MyList[A] =
|       import MyList.*
|       a match
|         case Empty => b
|         case Cons(x, xs) => Cons(x, concat(xs, b))

scala> def map[A, B](f: A => B, a: MyList[A]): MyList[B] =
|       import MyList.*
|       a match
|         case Empty => Empty
|         case Cons(x, xs) => Cons(f(x), map(f, xs))

scala> import MyList.*

scala> length(Cons(1, Cons(2, (Cons(3, Empty)))))
val res: Int = 3

scala> concat(Cons(1, Cons(2, (Cons(3, Empty)))), Cons(4, Cons(5, Empty)))
val res: MyList[Int] = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Empty)))))

scala> map((_:Int)*2, Cons(1, Cons(2, (Cons(3, Empty)))))
val res: MyList[Int] = Cons(2, Cons(4, Cons(6, Empty)))

```

### 12.3 Instances of type classes

Let us come back to our polymorphic functional implementation of insertion sort from Section 12.1.1. If we want to sort lists whose elements are of type `Nat`, we run into the problem that there is no possibility to compare `Nats`. Hence, we have to *put `Nat` into the type class `Ordering`*. To achieve this, we must implement the function `compare`. This is done as follows:

```

enum Nat:
  case Zero
  case PlusOne(n: Nat)

object Nat:
  given Ordering[Nat] with
    def compare(n: Nat, m: Nat) =
      (n, m) match
        case (Zero, Zero) => 0
        case (Zero, _) => -1
        case (_, Zero) => 1
        case (PlusOne(o), PlusOne(p)) => compare(o, p)

```

The first line `object Nat:` is necessary. It defines a *companion object* for the algebraic data type `Nat`. We will learn more about companion objects in Section 13.5.

**Attention:** This example does not work in the REPL. Instead, these definitions need to be written into a common file and compiled with the Scala compiler.



Now we can sort lists of type `Nat`:

```
@main
def test(): Unit =
  import Nat.*
  val n1: Nat = PlusOne(Zero)
  val n2: Nat = PlusOne(PlusOne(Zero))
  val n3: Nat = PlusOne(PlusOne(PlusOne(Zero)))
  val n4: Nat = PlusOne(PlusOne(PlusOne(PlusOne(Zero))))

  val l: List[Nat] = List(n3, n2, n4, n1)
  println(insertionSort(l))
```

The output is as follows:

```
List(PlusOne(Zero),
     PlusOne(PlusOne(Zero)),
     PlusOne(PlusOne(PlusOne(Zero))),
     PlusOne(PlusOne(PlusOne(PlusOne(Zero))))
```

Finally, if we wanted to use numeric operations on `Nat`, we would also need to put `Nat` into the type class `Numeric`. For this, we would have to extend our companion object with the line `given Numeric[Nat] with` and to implement all the necessary functions from Section 12.1.2. However, this means that we would also need to implement the function

```
def negate(x : Nat): Nat
```

that negates a natural number. Negation is not defined for natural numbers, so maybe `Nat` should not belong to `Numeric`, after all.

## 12.4 Arithmetic expressions

As a final example, we implement *arithmetic expressions* as algebraic data types. Arithmetic expressions are expressions of the form  $-(1 + 4) * 10$ . They are an essential building block in almost all programming languages. Formally, the set of possible arithmetic expressions is defined inductively, just like functional lists or the natural numbers. Let `A` be an arbitrary type. Then, the *arithmetic expressions over A* are defined as follows:

- **Values.** Every element of `A` is an arithmetic expression over `A`.

- **Sum.** Suppose that  $a_1$  and  $a_2$  are arithmetic expressions over **A**. Then,  $(a_1 + a_2)$  is an arithmetic expression over **A**.
- **Product.** Suppose that  $a_1$  and  $a_2$  are arithmetic expressions over **A**. Then,  $(a_1 \cdot a_2)$  is an arithmetic expression over **A**.
- **Negation.** Suppose that  $a$  is an arithmetic expression over **A**. Then,  $(-a)$  is an arithmetic expression over **A**.

For example  $t = ((-4) + (3 \cdot 7))$  is an arithmetic expression over **Int**. The definition of arithmetic expressions can be implemented directly as an algebraic data type:

```
scala> enum Arith[A]:
  |   case Value(v: A)
  |   case Plus(a1: Arith[A], a2: Arith[A])
  |   case Times(a1: Arith[A], a2: Arith[A])
  |   case Negate(a: Arith[A])
```

The arithmetic expression  $t$  could now be represented as follows:

```
scala> import Arith.*

scala> val t: Arith[Int] = Plus(Negate(Value(4)), Times(Value(3), Value(7)))
val t: Arith[Int] = Plus(Negate(Value(4)), Times(Value(3), Value(7)))
```

If we want to evaluate such an expression, we have to assume that **A** is in the type class **Numeric**. Then, we can implement an evaluation function using recursion and pattern matching:

```
scala> def eval[A: Numeric](a: Arith[A]): A =
  |   import Arith.*
  |   val num = summon[Numeric[A]]
  |   import num.mkNumericOps
  |   a match
  |     case Value(v) => v
  |     case Plus(a1, a2) => eval(a1) + eval(a2)
  |     case Times(a1, a2) => eval(a1) * eval(a2)
  |     case Negate(a) => -eval(a)

scala> eval(t)
val res: Int = 17
```

Last but not least, we implement a function to compute the *depth* of an arithmetic expression, i.e., the maximum number of nested parentheses.

```
scala> def depth[A](a: Arith[A]): Int =
  |   import Arith.*
  |   a match
  |     case Value(_) => 0
```



```
|      case Plus(a1, a2) => Math.max(depth(a1), depth(a2)) + 1  
|      case Times(a1, a2) => Math.max(depth(a1), depth(a2)) + 1  
|      case Negate(a) => depth(a) + 1
```

```
scala> depth(t)  
val res: Int = 2
```

# Entwurf



## **Object-Oriented Programming and Data Abstraction**

## KAPITEL 13

### Object-Oriented Programming

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP4** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *objektorientierter Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.



We will now go back to the world of imperative programming and see the paradigm of object-oriented programming. For concreteness, we will use Scala. But first, let us explain how imperative programming in Scala works (it is mostly very similar to imperative programming in Python).

#### 13.1 Imperative programming in Scala

**Variables.** In Section 10.2, we saw that every variable in Scala needs to be declared before it can be used. We say that Scala distinguishes between *value declarations* (using the keyword **val**) and *variable declarations* (using the keyword **var**). In the function context, we only used value declarations.

Now, we are back in the imperative world, and we will also need variable declarations. Variable declarations in Scala look the same as value declarations, except that they use the keyword **var** instead of **val**. The main difference is that we can *reassign* a **var**-variable, i.e., we can use a **var**-variable as the left-hand side of assignment statements multiple times. For example:

```
scala> var i: Int = 0
var i: Int = 0

scala> i = i + 1    // reassignment of i
```

```
i: Int = 1

scala> i = 12      // another reassignment of i
i: Int = 12
```

**Attention:** We note that *reassigning* a new value to a variable is not the same as *changing* an existing data object. The latter one can also be done with **val**-variables. We will see an example below when we talk about *arrays*, and also later in Chapter 13 when we talk about *mutable objects*.

**Branches.** Scala's **if**-statement looks very similar to a conditional expression (see Section 10.2). The difference now is that the individual branches are not expressions but *sequences of instructions* (i.e., blocks). The syntax is as follows:

```
if condition1 then
  Block1
else if condition2 then
  Block2
...
else
  Blockn
```

Here is an example:

```
scala> var i: Int = 0
var i: Int = 0

scala> if i > 0 then
|   i = 100
|   println("i was positive.")
| else if i == 0 then
|   i = -1
|   println("i was 0.")
| else
|   i = -100
|   println("i was negative.")
|
i was 0.

scala> i
val res0: Int = -1
```

**While-loops.** The syntax for a **while**-loop in Scala is as follows:

```
while condition do
  Block
```

Here is an example that prints the numbers from 1 to 10:

```
scala> var i: Int = 1
var i: Int = 1

scala> while i <= 10 do
|   println(i)
|   i = i + 1
|

1
2
3
4
5
6
7
8
9
10
```

Moreover, Scala also a *do-while-loop* where the condition is evaluated only *after* the first execution of the loop body. The syntax is as follows:

```
while
  Block
  condition
do ()
```

We will see an example of a do-while-loop below, when we talk about input/output-operations in Scala.

**For-loops.** The syntax of a **for**-loop in Scala is as follows:

```
for variable <- range do
  Block
```

Here, variable is the name of the loop variable, and range is a set of elements over which we can iterate (just as in Python, there are many possibilities; we will see examples later). To iterate over a range of numbers, we can write

```
for variable <- a to b do
  block
```

where a and b are the beginning and the end of the range (in Scala, a and b are both included in the range). For example, to print all numbers from 1 to 10, we write

```
scala> for i <- 1 to 10 do
      |   println(i)
      |
1
2
3
4
5
6
7
8
9
10
```

We can use additional syntax elements to define a step size, to count backwards, to define additional conditions for the loop-elements, etc. We refer to the Scala-documentation for more details.

**Input/output.** To output a string onto the terminal, Scala has two functions: `print` and `println`. The first function just outputs the string, while the second function also starts a new line.

The standard input functionality is provided by `scala.io.StdIn`. The function `StdIn.readInt()` reads an integer from the keyboard. It creates an exception if the input is not a valid integer. There are several other read-functions for different types.

The following function reads an input from the terminal until the user inputs a valid natural number. The function uses a do-while-loop and Scala's exception mechanism to repeat the input until a valid natural number is obtained. We will not say more about Scala's exception handling mechanism, but refer the interested reader to the Scala documentation.

```
import scala.io.StdIn

@main
def func() : Unit =
  var n: Int = -1
  while
    print("Please input a natural number: ")
    try
      n = StdIn.readInt()
    catch
      case e: Exception => n = -1
    n < 0
  do ()
  println(s"The input was ${n}.")
```

Here is an example output of the program:

```
wolfi@home:~$ scala read.scala
Compiling project (Scala 3.6.3, JVM (22))
Compiled project (Scala 3.6.3, JVM (22))
Please input a natural number: -1
Please input a natural number: Hello
Please input a natural number: 10
The input was 10.
```

**Arrays.** A central data structure in Python are *lists* (see Section 3.2.2). A list is a composite data type that allows us to store an arbitrary number of elements. In Scala, we have used functional lists for the same purpose (see Section 11.1). As discussed in Section 11.1, there are three main differences between imperative lists in Python and functional lists in Scala:

- Python lists are *mutable*, while functional lists in Scala are *immutable*;
- Python lists are *random-access*, while functional lists in Scala need to be *traversed*; and
- Python lists can store elements of *different types*, while all elements in a functional list in Scala must have *the same type*.

Scala provides a composite data type that is closer to lists in Python. This composite data type is called *array*, and similar data types exist in most other programming languages. The main properties of arrays are as follows:

- **Arrays are *mutable*.** We can assign new values to the elements of an array, just like with lists in Python.
- **All elements in an array have the same type.** When declaring an array, we need to fix a type for the elements that are stored in the array. This type is enforced by the Scala compiler and cannot be changed.
- **Arrays are *random-access*.** We can access elements at arbitrary positions of an array in time  $O(1)$ . The elements are next stored contiguously in the main memory.
- **Arrays have a fixed size.** When creating an array, we have to commit to a fixed size. This size cannot be changed. If we want to resize an array, we need to create a new array and copy all the elements from the old array to the new one.

In Scala, we can initialize an array in two different ways:

```
var a: Array[Type] = Array[Type](a0, a1, a2, ..., an)
var a: Array[Type] = new Array[Type](n)
```

In the first way, we define an array with  $n + 1$  elements  $a_0, \dots, a_n$ , in this order. The elements are listed explicitly in the definition. In the second way, we define an empty<sup>1</sup> array with exactly  $n$  positions (indexed from 0 to  $n - 1$ ).

We can use the index notation  $a(i)$  to read *and write* the element at position  $i$  of  $a$  (the first index is 0). The number of elements in an array  $a$  is given by  $a.length$ . Here are two examples:

```
scala> val a: Array[Int] = Array[Int](3, 2, 1, 3)
val a: Array[Int] = Array(3, 2, 1, 3)

scala> a(2) = 4

scala> for i <- 0 to a.length - 1 do
|   println(a(i))
|
3
2
4
3
```

and

```
scala> val a: Array[Int] = new Array[Int](4)
val a: Array[Int] = Array(0, 0, 0, 0)

scala> var i: Int = 0
var i: Int = 0

scala> for x <- 2 to 5 do
|   a(i) = x
|   i = i + 1
|

scala> for x <- a do
|   println(x)
|
2
3
4
5
```

The second **for**-loop in this example also shows that we can use an array directly as the range in a **for**-loop. The loop-variable then iterates over the elements of the array, in order.

---

<sup>1</sup>Here, *empty* means that all the positions of the array are filled with a default value that depends on the type of the array.





[Draft] • (None)@(None) • [(None)]

### 13.2 Introduction to OOP

**Motivation of object-oriented programming.** The term *object-oriented programming* is not clearly defined. It is a collection of conceptual ideas and syntactic structures that embody a certain philosophy of how software should be structured. As such, object-oriented programming was developed to address a major problem: “*How should we structure large software projects so that the complexity can be handled?*”

The observation that handling large software projects is difficult has been made very early in the history of Computer Science. Around the year 1968, the term “*software crisis*” was coined to express the fact that large software projects often do not end successfully. Over time, several ideas were developed to address this “software crisis” (however, to this day, the software crisis has never been resolved completely). Before we discuss how object-oriented programming tries to help with designing good software, let us first mention some of the goals of good software design:

- **Modularity.** The program should be structured in separate parts, each with a clear purpose. The parts should interact only through a clearly specified interface, the implementations of the different parts should be independent of each other, possibly done by different teams.
- **Reusability.** It should be possible to reuse existing code without much effort.
- **Extendability.** It should be possible to add new features to existing code without too much effort and without duplication of work.
- **Clarity.** It should be easy to get a quick understanding of the overall structure of the code and how the different parts interact with each other. The structure of the code should follow the situation in the real world as closely as possible.
- **Error prevention.** The code should be written in such a way that programming errors can be detected early and that possible errors can be located quickly.

Object-oriented programming provides a set of language features and concepts that are supposed to make it easier to achieve these goals. There is no single object-oriented paradigm. Nowadays, almost every modern programming language implements some features of object-oriented programming, to different degrees. The golden age of object-oriented programming was in the 1990s, when graphical user interfaces became widespread. In recent years, some aspects of object-oriented programming have become less popular, but the ideas are still ubiquitous. We will now look at some of the main ideas of object-oriented programming, and how they are implemented in Scala.

**Objects.** The central notion of object-oriented programming is the *object*. An object combines data and code that operates on the data into a single whole. This helps us to give a clear structure to our programs and to model entities of the real world directly in our code. This leads to clarity and modularity in our code.

More precisely, an *object* is a collection of data items (called *attributes*, *properties*, *members*, or *fields*) and functions that operate on these data items (called *methods*, *member functions*, or *messages*). An object always has a *state* (the current values of the attributes) and an *identity*. Two different objects will never have the same identity, but they could have the same state.

In Scala, it is possible to directly declare an object in the code, using the keyword **object**.

```
scala> object Maaax:
|   // Attributes:
|   var age: Int = 42
|   var job: String = "Lecturer"
|   // Methods:
|   def getAge(): Int = age
|   def getJob(): String = job
|   def mature(): Unit = age = age + 1
|   def changeJob(newJob: String): Unit = job = newJob
```

This code creates a single object **Maaax**. It has two attributes: (age and job) and four methods (getAge(), getJob(), mature(), and changeJob()). Attributes are declared in the same way as variables: we can use **var** for attributes that can be reassigned, and **val** for attributes that remain fixed. Methods are declared in the same way as usual functions. In the methods, we can use the attributes of the object as local variables. In Scala, it is common practice to capitalize the name of objects.

From outside the object, we can access the attributes and methods using the dot-notation:

```
scala> Maaax.age
val res: Int = 42

scala> Maaax.mature()

scala> Maaax.age = Maaax.age + 1

scala> Maaax.getAge()
val res: Int = 44

scala> Maaax.age
val res: Int = 44
```

The state of the object **Maaax** is given by the values of the attributes. As we see above, the state can be changed by invoking methods and by directly changing the attributes. The object **Maaax** has an identity. If we declare an object **Moritz** with exactly the same attributes and methods, the two objects will be different:

```
scala> object Maaax:
|   // Attributes:
|   var age: Int = 42
|   var job: String = "Lecturer"
|   // Methods:
|   def getAge(): Int = age
|   def getJob(): String = job
|   def mature(): Unit = age = age + 1
|   def changeJob(newJob: String): Unit = job = newJob

scala> object Moritz:
|   // Attributes:
|   var age: Int = 42
|   var job: String = "Lecturer"
|   // Methods:
|   def getAge(): Int = age
|   def getJob(): String = job
|   def mature(): Unit = age = age + 1
|   def changeJob(newJob: String): Unit = job = newJob

scala> Maaax == Moritz
val res: Boolean = false
```

Scala uses *reference semantics* for assignment statements that involve objects (see Section 2.4). This means that if we assign a variable with an object to another variable, these two variables refer to the same object, and changes to the state of the object that are made through one variable are also visible from the other variable:

```
scala> Maaax.getAge()
val res: Int = 42

scala> var moritz = Maaax
var moritz: Maaax.type = Maaax$@6b3b4f37

scala> moritz.mature()

scala> Maaax.getAge()
val res: Int = 43
```

Scala is a purely object-oriented programming language: every data item in Scala is an object—even functions are treated internally as objects.

### 13.3 Encapsulation and information hiding

So far, we have seen how to use an object to structure programs. Objects group functions and the data that they operate on into a unit. The next idea from object-oriented programming goes a step further: we can enforce that certain data items and methods can only be accessed from inside an object. This has two advantages: first, by

restricting access to certain attributes, we can ensure that the *integrity* of the data is maintained. It is not possible to put an object into an invalid state by changing the value of an attribute. Second, we can *hide* implementation details of the object from the outside world. This makes it possible to change the details of an implementation without running the risk of having to adapt the code that uses an object. This idea is called *encapsulation*.

As an example, recall the object **Maaax** from the previous section:

```
scala> object Maaax:
|   // Attributes:
|   var age: Int = 42
|   var job: String = "Lecturer"
|   // Methods:
|   def getAge(): Int = age
|   def getJob(): String = job
|   def mature(): Unit = age = age + 1
|   def changeJob(newJob: String): Unit = job = newJob
```

We saw that we can change the attributes of **Maaax** directly:

```
scala> Maaax.age = 23

scala> Maaax.age
val res: Int = 23
```

In this way, nothing prevents us from giving **Maaax** a *negative* age:

```
scala> Maaax.age = -10

scala> Maaax.age
val res1: Int = -10
```

It is very likely that this is not desirable, because other parts of our program could rely on the fact that age is nonnegative. We can use encapsulation to prevent this situation, by making the attribute age accessible only from inside the object. To change the age, we can provide a new method that ensures that the age is set correctly:

```
scala> object Maaax:
|   // Attributes:
|   private var age: Int = 42
|   var job: String = "Lecturer"
|   // Methods:
|   def getAge(): Int = age
|   def setAge(_age: Int): Unit =
|     if _age >= 0 then
|       age = _age
|     else
|       age = 0
```

```
| def getJob(): String = job
| def mature(): Unit = age = age + 1
| def changeJob(newJob: String): Unit = job = newJob
```

The keyword **private** is called an *access-modifier*. It ensures that `age` can only be used from inside the object `Maaax`. An attempt to read or write `age` from the outside will result in an error:

```
scala> Maaax.age
-- [E173] Reference Error: -----
1 | Maaax.age
  | ^^^^^^^^^
  | variable age cannot be accessed as a member of Maaax.type from object rs
  | private variable age can only be accessed from object Maaax.
1 error found

scala> Maaax.age = 10
-- [E173] Reference Error: -----
1 | Maaax.age = 10
  | ^^^^^^^^^
  | variable age cannot be accessed as a member of Maaax.type from object rs
  | private variable age can only be accessed from object Maaax.
1 error found
```

To modify the `age`, we must use the methods `getAge()`, `setAge()`, and `mature()`.

```
scala> Maaax.getAge()
val res: Int = 42

scala> Maaax.setAge(10)

scala> Maaax.getAge()
val res: Int = 10

scala> Maaax.mature()

scala> Maaax.getAge()
val res: Int = 11

scala> Maaax.setAge(-10)

scala> Maaax.getAge()
val res: Int = 0
```

In addition to **private**, there are more access-modifiers in Scala that let us control in more detail who can access certain attributes and methods of an object. We refer to the Scala-documentation for more details.

The underlying principle of encapsulation is called *information hiding*. This is a general principle in Computer Science that states that implementation details should

be hidden from the outside world. There should only be a clearly specified interface that can be used from the outside, the rest should be inaccessible and invisible. This makes it possible to *decouple* different parts of a system and supports the goal of modularity.

**Attention:** It is good practice to make *all* the attributes and auxiliary methods of an object **private**, and to allow access to attributes only through dedicated methods. (Auxiliary methods are methods that should not be visible to the outside world—we will see some examples later.)



## 13.4 Classes

So far, we have only seen how to create a single object. However, most of the time, we do not want to have only one object of a certain type, but *many* objects that have the same attributes and methods, but different identities and different states. For example, consider a scenario where we need hundreds of lamps (e.g., to light a dark floor). We want to represent the lamps in our program. Of course, it is not feasible to declare an individual object for each lamp. Instead, we can use a *class* to provide a template (blueprint) that can be used to create many objects of the same type. A class declares the attributes and the methods the corresponding objects will have. An object is then called an *instance of the class*. Here is an example:

```
scala> class Lamp:
  |   // Attributes
  |   private var power: Int = 0
  |   private var on: Boolean = false
  |   // Methods
  |   def turnOn(): Unit =
  |     on = true
  |   def turnOff(): Unit =
  |     on = false
  |   def isOn(): Boolean = on
  |   def change(_power: Int): Unit =
  |     if _power >= 0 && !on then
  |       power = _power
  |   def getPower(): Int = power
```

This code defines a class **Lamp**. The definition of a class is the same as for an object, except that we use the keyword **class** instead of the keyword **object**. The convention in Scala is that the names of classes are capitalized. The definition of a class does not create any objects. Instead, it creates a new data type that we can use in variable declarations. When initializing a variable of this type, we can create a new object with the keyword **new**, followed by the name of the class:

```
scala> var l1: Lamp = new Lamp()
var l1: Lamp = Lamp@5b88af70

scala> var l2: Lamp = new Lamp()
var l2: Lamp = Lamp@2474df51
```

This code creates two different objects that are instances of the class `Lamp`. These objects have the same attributes and methods, but different states and identities.

```
scala> l1 == l2
val res: Boolean = false

scala> l1.turnOn()

scala> l1.isOn()
val res: Boolean = true

scala> l2.isOn()
val res: Boolean = false
```

We can once again observe that Scala uses reference semantics for assignments of variables that refer to objects:

```
scala> var l3: Lamp = l1
var l3: Lamp = Lamp@5b88af70

scala> l1.isOn()
val res: Boolean = true

scala> l3.turnOff()

scala> l1.isOn()
val res: Boolean = false
```

We can also see that `Lamp` is a mutable object (because it has a `var`-attribute that can be changed through the methods of `Lamp`). This fact is independent of whether we define a `Lamp`-variable with `var` or `val`:

```
scala> val l1: Lamp = new Lamp()
val l1: Lamp = Lamp@52d59507

scala> l1.isOn()
val res: Boolean = false

scala> l1.turnOn()

scala> l1.isOn()
val res: Boolean = true
```



```
scala> l1 = new Lamp()
-- [E052] Type Error: -----
1 | l1 = new Lamp()
  | ^^^^^^^^^^^^^^^
  | Reassignment to val l1
1 error found
```

**Attention:** We note that the access-modifiers of Scala work at the class level and not at the object level. For example, the following code is valid:

```
scala> class Lamp:
  |   // Attributes
  |   private var power: Int = 0
  |   private var on: Boolean = false
  |   // Methods
  |   def turnOn(): Unit =
  |     on = true
  |   def turnOff(): Unit =
  |     on = false
  |   //turn on another lamp
  |   def turnMeOn(l: Lamp): Unit =
  |     l.on = true
  |   def isOn(): Boolean = on
  |   def change(_power: Int): Unit =
  |     if _power >= 0 && !on then
  |       power = _power
  |   def getPower(): Int = power

scala> var l1: Lamp = new Lamp()
var l1: Lamp = Lamp@507f47f9

scala> var l2: Lamp = new Lamp()
var l2: Lamp = Lamp@22002459

scala> l2.isOn()
val res: Boolean = false

scala> l1.turnMeOn(l2)
scala> l2.isOn()

val res: Boolean = true
```

Within the method `turnMeOn`, we can access the private attribute `on` of a different `Lamp`-object that is passed as a parameter. This means that the lamp `l1` can turn on the lamp `l2`.

**Constructors.** As we just saw, new instances of a class are created with the keyword `new`. This is called the *construction* of the instance. It is possible to define certain code that is executed whenever an instance of a class is constructed. This code is

called the *constructor* of the class. The constructor can be used to initialize the state of the instance and to set up the environment. A constructor may have parameters, e.g., to provide the initial values of certain attributes. In Scala, the parameters of the constructor are written in parentheses behind the name of the class, and the code of the constructor is written directly in the body of the class:

```
scala> class Lamp(_power: Int):  
  | // Constructor  
  | println(s"Hello. This is a new lamp with power ${_power}.")  
  | // Attributes  
  | private var power: Int = _power  
  | private var on: Boolean = false  
  | // Methods  
  | def turnOn(): Unit =  
  |   on = true  
  | def turnOff(): Unit =  
  |   on = false  
  | def isOn(): Boolean = on  
  | def change(_power: Int): Unit =  
  |   if _power >= 0 && !on then  
  |     power = _power  
  | def getPower(): Int = power  
  
scala> var l1: Lamp = new Lamp(100)  
Hello. This is a new lamp with power 100.  
var l1: Lamp = Lamp@2cc97e47  
  
scala> var l2: Lamp = new Lamp(50)  
Hello. This is a new lamp with power 50.  
var l2: Lamp = Lamp@7378c4a4
```

This code equips the class **Lamp** with a constructor that has one parameter: `_power`. This parameter is used to initialize the power-attribute of **Lamp**. In addition, the constructor consists of a `println`-instruction that is executed whenever an instance of **Lamp** is constructed.

In Scala, a parameter to the constructor is called a *class parameter*. In the example above, the class parameter is used only to initialize an attribute of **Lamp**. For this situation, Scala provides a special shortcut: we can directly define an attribute of a class in the parameter list of the constructor, using the keywords **var** or **val**. In this case, we can also add an access-modifier to control the visibility of this attribute:

```
scala> class Lamp(private var power: Int):  
  | println(s"Hello. This is a new lamp with power ${power}.")  
  | // Attributes  
  | private var on: Boolean = false  
  | // Methods  
  | def turnOn(): Unit =  
  |   on = true
```

```
|     def turnOff(): Unit =
|         on = false
|     def isOn(): Boolean = on
|     def change(_power: Int): Unit =
|         if _power >= 0 && !on then
|             power = _power
|     def getPower(): Int = power

scala> var l1: Lamp = new Lamp(100)
Hello. This is a new lamp with power 100.
var l1: Lamp = Lamp@22f1a340

scala> var l2: Lamp = new Lamp(50)
Hello. This is a new lamp with power 50.
var l2: Lamp = Lamp@10efb806
```

This code declares a **private var**-attribute `power` of the class `Lamp` as a parameter to the constructor. Whenever an instance of `Lamp` is constructed, the attribute `power` is initialized directly with the actual parameter of the constructor. An attribute that is defined in this way is called a *parametric attribute* of a class.

**Secondary constructors.** We can use *ad-hoc-polymorphism* (see Section 11.2.1) to equip a class with multiple constructors that have different signatures. In this case, the constructor of a class is *overloaded*: when a new instance of the class is constructed, the compiler uses the number and types of the actual parameters to determine which constructor should be used. In Scala, we distinguish between the *primary* constructor and the *secondary* constructors. A class has always one primary constructor. As we saw above, the parameter list of the primary constructor is given behind the class name and the instructions of the primary constructor appear directly in the body of the class.

In addition to the primary constructor, there can be an arbitrary number of secondary constructors. A secondary constructor is defined in the same way as a regular method in the body of a class, but it must have the special name `this` and no return type. In the function body, the secondary constructor must first invoke another constructor—usually the primary constructor. In the next example, we extend the `Lamp`-class with several secondary constructors:

```
scala> class Lamp(private var power: Int):
|     println(s"Hello. This is a new lamp with power ${power}.")
|     // Attributes
|     private var on: Boolean = false
|     // Secondary constructors
|     def this() =
|         this(100)
|         println("Hello from the first secondary constructor.")
|     def this(_power: Int, _on: Boolean) =
```

```
|         this(_power)
|         on = on
|         println("Hello from the second secondary constructor.")
|     // Methods
|     def turnOn(): Unit =
|         on = true
|     def turnOff(): Unit =
|         on = false
|     def isOn(): Boolean = on
|     def change(_power: Int): Unit =
|         if _power >= 0 && !on then
|             power = _power
|     def getPower(): Int = power

scala> var l1: Lamp = new Lamp(100)
Hello. This is a new lamp with power 100.
var l1: Lamp = Lamp@61b0af9f

scala> var l2: Lamp = new Lamp()
Hello. This is a new lamp with power 100.
Hello from the first secondary constructor.
var l2: Lamp = Lamp@336070ab

scala> var l3: Lamp = new Lamp(50, true)
Hello. This is a new lamp with power 50.
Hello from the second secondary constructor.
var l3: Lamp = Lamp@7bfcc108
```

**Garbage collection and destructors.** We have seen how to construct objects as instances of a class. But what do we do if we no longer need an object? In Scala, we do not need to do anything. As soon as an object is no longer associated with any variable in the program, the object will be automatically removed from memory. No intervention is necessary from our side. This process is called *garbage collection*. Garbage collection is not only provided by Scala, but also by other programming languages like Java or Python. However, in other programming languages, e.g., in C++, there is no garbage collection. Here, we need to manually delete an object when it is no longer needed. For this, objects provide a special method, the *destructor*. The destructor is called whenever an object is supposed to be removed from memory. We refer the interested reader to the literature on C++ and related languages to learn more about manual memory management and destructors.

**Requirements.** Scala offers a special instruction `require` that can be used to ensure that certain conditions are fulfilled. The instruction `require` checks a Boolean expression. If this expression evaluates to **false**, then `require` throws an exception. This is useful for checking that an object is constructed with valid parameters.

```
scala> class Lamp(private var power: Int):  
  | // Check that power has a valid value.  
  | require(power >= 0)  
  | println(s"Hello. This is a new lamp with power ${power}.")  
  | // Attributes  
  | private var on: Boolean = false  
  | // Methods  
  | def turnOn(): Unit =  
  |   on = true  
  | def turnOff(): Unit =  
  |   on = false  
  | def isOn(): Boolean = on  
  | def change(_power: Int): Unit =  
  |   if _power >= 0 && !on then  
  |     power = _power  
  | def getPower(): Int = power  
  
scala> var l1: Lamp = new Lamp(50)  
Hello. This is a new lamp with power 50.  
var l1: Lamp = Lamp@4df13dd0  
  
scala> l1 = new Lamp(-50)  
java.lang.IllegalArgumentException: requirement failed
```

**The keyword `this`.** The keyword `this` plays a special role in the definition of a class. Let us mention three typical scenarios:

- As we saw above, we can use `this` to define *secondary constructors* of a class, using ad-hoc polymorphism and overloading.
- Within a secondary constructor, we can use `this` to call another constructor (typically the primary constructor).
- Within a method, we can use `this` to refer to the current object. This is usually not necessary, but can sometimes increase the readability of the code.

```
scala> class Lamp(private var power: Int):  
  | println(s"Hello. This is a new lamp with power ${power}.")  
  | // Attributes  
  | private var on: Boolean = false  
  | // Methods  
  | def turnOn(): Unit =  
  |   on = true  
  | def turnOff(): Unit =  
  |   on = false  
  | def isOn(): Boolean = on  
  | // Copy the on-state from another lamp  
  | def copyOn(l: Lamp): Unit =  
  |   this.on = l.on  
  | def change(_power: Int): Unit =
```

```
|         if _power >= 0 && !on then
|             power = _power
|         def getPower(): Int = power

scala> var l1: Lamp = new Lamp(10)
Hello. This is a new lamp with power 10.
var l1: Lamp = Lamp@5831989d

scala> var l2: Lamp = new Lamp(20)
Hello. This is a new lamp with power 20.
var l2: Lamp = Lamp@5fd4e67f

scala> l2.turnOn()

scala> l1.isOn()
val res: Boolean = false

scala> l1.copyOn(l2)

scala> l1.isOn()
val res: Boolean = true
```

In the method `copyOn`, we use `this` to distinguish the `on`-attribute of the current object from the `on`-attribute of the parameter object `l`.

### 13.5 Sharing state between instances of a class

Sometimes, we would like to share a common state between different instances of the same class. For example, consider the following class that is used to model the entity **Student**.

```
scala> class Student(private val name: String, private val matr: Int):
|     // Attributes
|     private val subject: String = "Bio-Informatics"
|     private var ects: Int = 0
|     // Methods
|     def study(topic: String): Unit =
|         println(s"${this.name} reads a book about ${topic}.")
|         this.ects = this.ects + 1
|     def isDone(): Boolean = this.ects >= 180
|     def getMatr(): Int = this.matr
|     def getName(): String = this.name
```

Usually, when students join a university, they are assigned a *student id-number* (matriculation number). These id-numbers are given in ascending order. With our definition of **Student**, the id-number is passed to the constructor as a parametric attribute. It is up to the user of the class to ensure that the id-numbers are valid and that each student receives a unique id-number. Of course, this is not a very good idea.

To solve this problem, it would be useful if all the instances of **Student** could share a common counter that can be used to create unique ascending student id-numbers. To support encapsulation, it would also be nice if this counter could be accessed only from inside the class **Student**.

In Scala, this can be realized as follows: whenever we define a class, the compiler automatically creates a *singleton object* that has the same name as the class. This singleton object is called the *companion object* of a class. All instances of the class are able to access the private attributes and methods that are defined in the companion object of the class. To share a common state between all instances of a class, we can implement our own companion object, and we can define the common attributes and methods in this companion object.

Coming back to our example, we define a companion object **Student** for the class **Student**. In this companion object, we maintain a counter `matCounter` that indicates the next free student id-number, as a private attribute. In the constructor of class **Student**, we access `matCounter` to assign an id-number to the new instance, and we increment the counter by one. To access the attribute `matCounter` of the companion object from the class, we use the syntax **Student**.`matCounter`. The implementation looks as follows:

```
class Student(private val name: String):  
  // Attributes  
  private val subject: String = "Bio*Informatics"  
  private var ects: Int = 0  
  // initialize id-number using the companion object  
  private var matr: Int = Student.matCounter  
  Student.matCounter = Student.matCounter + 1  
  // Methods  
  def study(topic: String): Unit =  
    println(s"${this.name} reads a book about ${topic}.")  
    this.ects = this.ects + 1  
  def isDone(): Boolean = this.ects >= 180  
  def getMatr(): Int = this.matr  
  def getName(): String = this.name  
  
  // the companion object  
  object Student:  
    // the counter for the id-number  
    private var matCounter: Int = 100  
  
    def resetMatCounter(newCounter : Int): Unit =  
      matCounter = newCounter  
  
@main  
def test(): Unit =  
  val s1: Student = new Student("Max")  
  val s2: Student = new Student("Moritz")  
  val s3: Student = new Student("Katharina")  
  val s4: Student = new Student("Michaela")
```

```
println(s"${s1.getName()} has id-number ${s1.getMatr()}.")
println(s"${s2.getName()} has id-number ${s2.getMatr()}.")
println(s"${s3.getName()} has id-number ${s3.getMatr()}.")
println(s"${s4.getName()} has id-number ${s4.getMatr()}.")
```

**Attention:** This example does not work in the REPL. To make the connection between a class and its companion object, the two structures need to be written in a single file and compiled with the Scala compiler.

The output of the program looks as follows:

```
wolfi@work:~$ scala student.scala
Compiling project (Scala 3.6.3, JVM (22))
Compiled project (Scala 3.6.3, JVM (22))
Max has id-number 100.
Moritz has id-number 101.
Katharina has id-number 102.
Michaela has id-number 103.
```

An attribute that is shared between all instances of a class is also called a *class variable*. The role of class variables is similar to the role of global variables in an imperative program, but the scope and usage of a class variable is more restricted. The notion of a companion object is very specific to Scala. However, almost all object-oriented programming languages support some notion of class variables, with different realizations in the language design.

## 13.6 Inheritance

In object-oriented design a *subclass* extends a *super class*. The subclass *inherits* from the super class, i.e., the subclass takes over the attributes and methods of the upper class and can add further attributes and methods. Subclasses can then be super classes of other classes. This leads to a *class hierarchy*. The *root* of this hierarchy – the highest upper class – is called **Any**. Every class in Scala automatically inherits from this class **Any**. Thus, every class automatically has the `toString`-function, since this function is a method of the class **Any**.

As a simple example we can look at two classes **Prof** and **Student** that inherit from the super class **Person**. First of all, we have the super class **Person**, which is implemented as follows:

```
class Person(private val name: String, private var age: Int):
  def isGrownUp : Boolean = age >= 18
  def work() : Unit = println("Work, work")
```

It has two private attributes and two methods. Next, the class **Student** inherits from **Person** and works as follows:



```
class Student(name: String, age: Int, private val mat: Int) extends Person(name, age):  
  def mat_nr : Int = mat  
  override def work() : Unit = println("Study, study")
```

This class has another private attribute (the enrolment number). Writing the expression `extends Person(name, age)` means, that the constructor of `Student` first calls the constructor of its super class `Person` with the two given parameters. Moreover, since `work()` has the same signature in both classes, we need to **override** the function in the sub class. An object of the class `Student` now has a different work-method than an object of class `Person`.

Finally, to increase the size of our class hierarchy we can also implement a third class:

```
class Prof(name: String, age: Int, private var sal: Int) extends Person(name, age):  
  def salary : Int = sal  
  override def work() : Unit = println("Hold monologue")
```

**Inclusion polymorphism.** Let us consider the following Scala-line, which works perfectly fine:

```
var p : Person = Student("Anton", 19, 621637)
```

We assigned an object of type `Student` to a variable of type `Person`. This works, because `Student` is a subclass of `Person`. We call this phenomenon *inclusion polymorphism*. We can reassign as follows:

```
p = Prof("Mulzer", 34, 42)
```

The type of the object changed from `Student` to `Prof`. The type of the corresponding object that is stored in `p` is called *dynamic type*, because it can change over time. However, the program still interprets the object as `Person`, because the type of declaration – the *static type* – is still `Person`. Therefore, `p.mat_nr` or `p.salary` crashes, since the two functions are not members of the class `Person`.

Inclusion polymorphism is the reason why we can write code like this:

```
val a : Array[Any] = Array[Any](9, 4.5, "Hallo", true)
```

All elements are interpreted as objects of type `Any`, because `Int`, `Double`, `String` and `Boolean` are subclasses of `Any`.

## KAPITEL 14

### Queues and Stacks

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP4** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *objektorientierter Programmierkonzepte*.
- KdP5** Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und -paradigmen.
- KdP9** Du beschreibst unterschiedliche Implementierungen *abstrakter Datentypen* und vergleichst diese miteinander.



#### 14.1 Traits

An **abstract class** is a class that has at least one unimplemented method. Although abstract classes have constructors and destructors like concrete classes we are not able to create instances of abstract classes. Instead abstract classes provide templates for subclasses. If a class inherits from an abstract class, it either must implement the abstract methods or must be declared as abstract as well. The signature of the implementation in the subclass must fit to the signature in the abstract super class.

For example the class **Person** should be implemented as abstract, since the method `work()` depends on the concrete subclasses: students work differently than lecturers, and both work differently than physicians etc.

```
abstract class Person(val name: String, var age: Int):  
  def isGrownUp : Boolean = age >= 18  
  def work() : Unit      // abstract method  
  
class Studi(name : String, age : Int, val matr : Int) extends Person(name, age):  
  def mat_nr : Int = matr  
  def work() : Unit = println("Study, study") // "override" not needed anymore
```

```
class Prof(name : String, age : Int, val sal : Int) extends Person(name, age):  
  def salary : Int = sal  
  def work() : Unit = println("Hold monologue") // "override" not needed anymore
```

Now, by inclusion polymorphism we can still write the following assignment:

```
var s : Person = Studi("Maaax", 37, 4164581)  
// var p : Person = Person("Maaax", 37) does not compile work Person is abstract
```

However, since Java und Scala do not allow multiple inheritance we cannot extend classes from more than one (abstract) class. Let us for example consider the following situation:

- Students and Lecturers are both persons.
- Students are persons that learn.
- Lecturers are persons that learn (by researching) but also work (by giving lectures) – they earn money.
- Not all students earn money by working. Hence, not all of them work.

If we want to model this behaviour we need a new concept in object-oriented programming: the **trait**. A trait can be seen as abstract class – it usually has abstract methods – but it is more seen as a *property* that we *mix in* different classes. For example we can mix the properties *working person* and *learning person* into lecturers. Like (abstract) classes, traits define a new type, e.g., **Ordering** is implemented as trait<sup>1</sup>. In most of the cases, traits only have unimplemented methods. Nevertheless, it is technically possible to implement attributes, concrete methods and even constructors within traits. Moreover, traits can inherit from other traits. A class/trait should only inherit multiple traits if the traits do not have concrete methods of the same signature, otherwise, we would not be able to decide which implementation is the one we need.<sup>2</sup> Concerning our example from above, we can do as follows:

```
trait WorkingPerson:  
  def work(): Unit  
  def salary: Int  
  
trait LearningPerson:  
  def learn(topic: String): Unit
```

These are two different traits: the working person and the learning person. Moreover, we have a class that can also be used as super class:

<sup>1</sup><https://dotty.epfl.ch/api/scala/math/Ordering.html>

<sup>2</sup>Technically, it is possible to inherit from different traits that have implemented methods of the same signature, but then we have to override this method in the class/trait that mixes in the traits. However, this is bad programming style and should be avoided.

```
class Person(private val name: String, private var age: Int):  
  def isGrownUp : Boolean = age >= 18  
  override def toString : String = name
```

Now, a student *is a* person having the *property* to be a learning person. Hence, **Studi** inherits from **Person** and mixes the trait **LearningPerson** in. Syntactically, it looks like this:

```
class Studi(name: String, age: Int, mat_nr: Int)  
  extends Person(name, age), LearningPerson:  
  def matrikel_nr: Int = mat_nr  
  def learn(topic: String): Unit = println("I learn about " + topic)
```

Finally, the class for the lecturer looks like this:

```
class Prof(name: String, age: Int, _salary: Int)  
  extends Person(name, age), WorkingPerson, LearningPerson:  
  def salary: Int = _salary  
  def learn(topic: String): Unit = println("I research about " + topic)  
  def work(): Unit = println("Hold monologue.")
```

A small test looks as follows:

```
var p : LearningPerson = Studi("Wolfgang", 22, 3426167)  
p.learn("Traits")  
p = Prof("Lena", 42, 12)  
p.learn("Super-Traits")
```

## 14.2 Abstract Data Types

An *abstract data type* is used to describe the external functionality and the externally visible properties of certain objects and abstracts from the concrete way in which this functionality is implemented. The implementation of an abstract data type is then called data structure. An abstract data type is given by its *specification*:

- (i) specification of the stored data
- (ii) indication of the operations to be performed on the data and their specification

The great advantage of this approach is the abstraction. The user uses the interface (the specification) without knowing how it works internally. The developer can provide any implementations and subsequently provide the user with better implementations who does not have to change his own program.

In Scala, we can use the concept **trait** to describe the abstract data types. The different implementations will then mix in this trait.

In our class we will consider four different abstract data types and discuss the different implementations: Stack, Queue, PriorityQueue, and Dictionary/Map. The four data types can be implemented either with *arrays* or with so called *linked nodes*. We will then analyse and discuss the advantages and disadvantages of these implementations.

### 14.3 Stacks

First, we consider the abstract data type *Stack*. In a stack we can store elements of *arbitrary* types. We can insert and remove elements. The stack is a so called *Last-In-First-Out*-storage (LIFO-principle), i.e., we always remove the most recent element of the stack. The specification looks as follows:

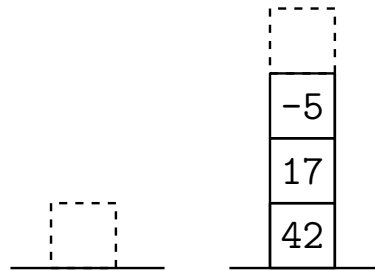
```
trait MyStack[A]:  
  // Objects of an arbitrary but fixed type A are stored in a stack.  
  
  // Precondition: None  
  // Result: None  
  // Effect: x is now the most recent element in the stack.  
  def push(x : A) : Unit  
  
  // Precondition: Stack is not empty.  
  // Result: The most recent element is returned.  
  // Effect: The most recent element is removed from the stack.  
  def pop() : A  
  
  // Precondition: Stack is not empty.  
  // Result: The most recent element is returned.  
  // Effect: None  
  def top : A  
  
  // Precondition: None  
  // Effect: None  
  // Result: true is returned if and only if the stack has no elements.  
  def isEmpty : Boolean  
  
  // Precondition: None  
  // Effect: None  
  // Result: The number of elements in the stack is returned.  
  def size : Int
```

We can imagine a deck of cards or a stack of books: we can put elements on the top of the stack or remove the top element, but we are not allowed to remove (or even read) an element somewhere in the middle of the stack.

Stacks are used for recursive programs: whenever we call a function, we put this function onto the stack. In primitive recursion this stack grows until we reach a recursion anchor, then the top function is removed. Another example is the *Back*-button on your internet browser. You go back to the site you have visited before. In

graph theory we use stacks as data structures to implement depth-first-search – a very important algorithm<sup>3</sup>.

This, is how we can imagine an empty stack or a stack with 3 elements:



### 14.3.1 LinkedNodes implementation

First of all, we want to consider an implementation of the stack using *linked nodes*. Elements are stored in *nodes*, which also contain a reference/link to the node of the next element. In Scala, these nodes are represented by an *inner class* called **Node** – a private class that is only visible within the class **LinkedNodesStack**.

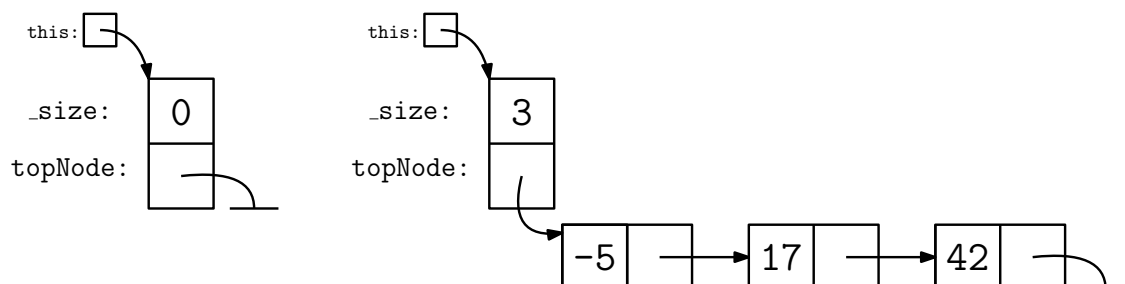
```
private class Node(val item: A, val next: Node)
```

Moreover, in the *header* we store the actual size of the stack as well as a reference to the highest (i.e., recent) element. The header corresponds to the private attributes of **LinkedNodesStack**.

```
class LinkedNodesStack[A] extends MyStack[A]:
  // The inner class:
  private class Node(val item: A, val next: Node)

  // Header:
  private var topNode : Node = null
  private var _size : Int = 0
  // [...]
```

Here we can see how an empty stack or a stack with three elements would look like:



Hence, we can now easily implement the two `isEmpty` and `size` as follows:

<sup>3</sup>Spoiler alert: You learn this in „Discrete Structures“ and „Algorithms and Data Structures“

```
class LinkedNodesStack[A] extends MyStack[A]:  
  // [...]  
  def isEmpty : Boolean = topNode == null  
  def size : Int = _size  
  // [...]
```

In the push-method we create a new Node object that stores the item and put the new object at the front of the linked list. The node that was previously referenced by the head now becomes the second node in the linked list. Finally, we increment the size.

```
class LinkedNodesStack[A] extends MyStack[A]:  
  // [...]  
  def push(x : A) : Unit =  
    topNode = Node(x, topNode)  
    _size = _size + 1  
  // [...]
```

In the pop- and top-method we first check whether the stack is non-empty. If yes, we temporarily store the element in the top node and return it at the end. In the pop-function we update the variable topNode to the next node in the order and decrement the size.

```
class LinkedNodesStack[A] extends MyStack[A]:  
  // [...]  
  def pop() : A =  
    if !isEmpty then  
      val result: A = topNode.item  
      // If the top node was simultaneously the last node  
      // in the chain, topNode becomes null.  
      topNode = topNode.next  
      _size = _size - 1  
      result  
    else throw Exception("Stack is empty")  
  def top : A =  
    if !isEmpty then topNode.item  
    else throw Exception("Stack is empty")
```

### 14.3.2 Array implementation

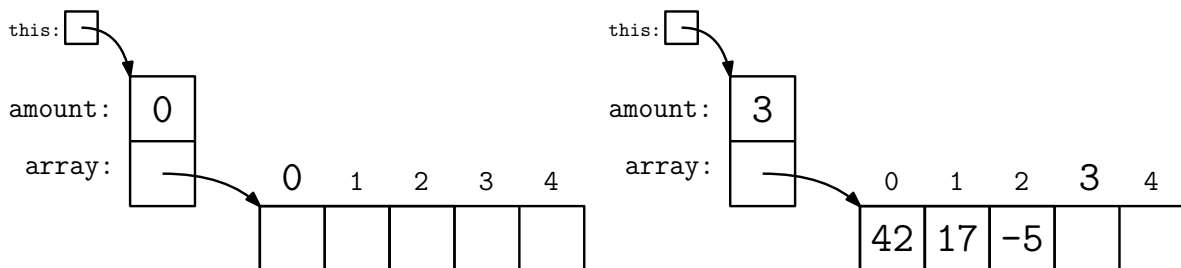
Next, we consider an implementation of the stack using *arrays of static size*. The header of the implementation contains two different variables. The elements are stored in an array. The array contains the elements of the stack in order from the oldest element to the most recent element (the oldest element is stored at index 0, the second most oldest element is at index, and so on). The other variable, called amount stores the number of elements that are stored in the stack. Hence, if the stack has at least one element, the most recent element is stored at position amount-1.

Since we use an array of static size, the stack has an upper bound of elements, which is called capacity. This parameter has to be passed to the constructor of the **ArrayStack**.

```
class ArrayStack[A : ClassTag](capacity: Int) extends MyStack[A]:
  // Header:
  private val n : Int = if capacity < 1 then 1 else capacity
  private val array : Array[A] = new Array[A](n)
  private var amount : Int = 0
  // [...]
```

You might notice, that we declared the type **A** to be an instance of **ClassTag**. The reason for that lies deep in the implementation of Scala, Java and the JVM and coincides with the fact that we created an array with elements of arbitrary type. We do not want to go into detail here. Instead we notice that, whenever we want to create an array of arbitrary type **A**, this type has to be an instance of **ClassTag** and we have to import `scala.reflect.ClassTag`. If you are curious about you might consult your favourite AI or the Scala documentation<sup>4</sup>.

Here we can see how an empty stack or a stack with three elements would look like:



In the push-method we first check, whether have not reached the maximum capacity. If not, we store the element at the correct position and increase the amount.

```
class ArrayStack[A : ClassTag](capacity : Int) extends MyStack[A]:
  // [...]
  def push(x : A) : Unit =
    if amount < array.length then
      array(amount) = x
      amount = amount + 1
    else throw new Exception("The stack is full")
  // [...]
```

In the pop-method we first check whether the stack is non-empty. If yes, we temporarily store the element at position `amount-1` and return it at the end. After that we write **null** to the position of the most recent element<sup>5</sup> and decrease the amount afterwards.

<sup>4</sup> <https://dotty.epfl.ch/api/scala/reflect/ClassTag.html>

<https://users.scala-lang.org/t/classtag-for-parametrized-array/3894>

<sup>5</sup> Otherwise the garbage collection would not remove the object.



```
class ArrayStack[A : ClassTag](capacity : Int) extends MyStack[A]:  
  // [...]  
  def pop() : A =  
    if !isEmpty then  
      val result : A = array(amount - 1)  
      array(amount - 1) = null.asInstanceOf[A]  
      amount = amount - 1  
      result  
    else throw new Exception("Stack is empty")  
  
  def top : A =  
    if !isEmpty then array(amount - 1)  
    else throw new Exception("Stack is empty")
```

### 14.3.3 Discussion

Next, we want to compare the two implementations of a stack. First, we briefly have to think about different criteria on how to compare different implementations. Here, we have some important issues:

- Running time of the operations. (The faster, the better.) This usually can be measured in terms of the O-notation, but there could be other (technical) problems that might influence the running time.
- Memory size. (The less, the better.)
- Complexity of implementation. Of course, this can be very subjective. The number of source code lines or the usage of complicated concepts are good parameters for a discussion.
- Restrictions. For example can we use the implementation only for specific data types, are there memory or time restrictions, etc.

Thus, we have good criteria for a comparison of the two implementations of the stack.

**Running times.** In both implementations all methods work in constant time in terms of the O-notation. We do not have any loops or recursions or expensive calls of functions. However, if we take a closer look into the technical issues we will observe that arrays have a much better *caching performance*<sup>6</sup> as linked nodes. The reason is that elements stored in an array are always next to each and can be accessed faster than linked nodes that are spread among the entire program memory. In terms of running time the array implementation is better.

---

<sup>6</sup>You will learn about this in higher classes.

**Memory.** The memory of the linked nodes implementation only depends on the number of elements in the stack. Moreover, the memory size changes dynamically with the number of elements: the more elements in the stack the higher the memory consumption and vice versa, because for each element in the stack we only store a reference to this element and a reference to a next node. The memory of the array implementation does not depend on the number of elements, it depends only on the capacity of the underlying array. This results in two problems. First, the ratio between the array size and the number of elements could be quite large: for example our stack could store up to 10,000,000 elements but the application only needs 10 – 15 – a lot of wasted space. The second problem is discussed under restrictions. In terms of memory the linked nodes implementation is better.

**Complexity of implementations.** In the complexity of implementation the two implementations are not much different. They neither use complex concepts or functions and they have both more or less a comparable number of source code lines.

**Restrictions.** The linked nodes implementation does not have any restrictions. The array implementation, however, does. The stack has a fixed capacity. Hence, the stack is not able to grow/shrink dynamically.

### 14.3.4 Arrays of dynamic size

Although the running time of the array implementation is better due to the better caching performance, it has the big problem that it cannot shrink or grow dynamically – the array has *static size*. We can get rid of this problem by using an array of *dynamic size*. For this, we consider the two values `amount` and `array.length` of the original implementation. During the whole life time of a stack with arrays of dynamic size we maintain the following invariant:

```
array.length/4 <= amount < array.length
```

Whenever we call `push` or `pop` we check whether the invariant has been violated. If this is the case, we create a new array whose size is either half the size of the old array (if `array.length/4 > amount`) or double the size of the old array (if `amount >= array.length`), copy all elements from the old to the new array and remove the old array. We achieve this by a private help function `resize()`, which is then used in the functions `push` and `pop`. The process can be seen in the following figure, where the shaded regions illustrate the amount of elements that are actually stored in the array.

# Entwurf

## Kapitel 14 Queues and Stacks

amount >= array.length

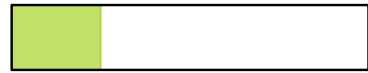


resize()



amount < array.length

array.length/4 > amount



resize()



array.length/4 <= amount

```
class DynamicArrayStack[A : ClassTag] extends MyStack[A]:
  // Header:
  private var array : Array[A] = new Array[A](1)
  private var amount : Int = 0
  // Precondition: None
  // Result: None
  // Effect: (array.length/4 <= amount < array.length) holds.
  private def resize() : Unit =
    val cap : Int = array.length
    if cap/4 <= amount && amount < cap then return // do nothing if INV holds
    // allocate a new array with half or double the size of the old array:
    val newArray : Array[A] = new Array[A](if cap/4 > amount then cap/2 else cap*2)
    // Copy all elements to new array:
    for i <- 0 to amount-1 do
      newArray(i) = array(i)
    array = newArray // the old array is now removed
  // [...]
```

Now, we can implement the other two methods:

```
class DynamicArrayStack[A : ClassTag] extends MyStack[A]:
  // [...]
  def push(x : A) : Unit =
    array(amount) = x
    amount = amount + 1
    resize() // Array too small? ... Resize!
  def pop() : A =
    if !isEmpty then
      val result : A = array(amount - 1)
      array(amount - 1) = null.asInstanceOf[A]
      amount = amount - 1
      resize() // Array too large? ... Resize!
      result
    else throw new Exception("Stack is empty")
```

**Discussion.** Of course, we should now ask, what did we get from this feature and how does it compare to the linked nodes implementation. First of all, we removed

any restrictions on the size of the stack. Hence, our stack is now able to grow and shrink dynamically. Moreover, the complexity of the implementation increased due to the help function `resize()`. Furthermore, since we always check the invariant, we can guarantee that the ratio between the number of elements in the stack and the size of the allocated arrays is always between 0.25 and 1 which is very good and comparable to the linked nodes implementation. Moreover, since we use arrays the caching performance is much better than for the linked nodes of course.

However, a potential disadvantage is that whenever the invariant is violated, we have to copy ALL elements from position to another. Hence, sometimes the push- and pop-operations have worst-case running time  $O(n)$ . BUT: The larger an array is, the more unlikely it is that we have to do an expensive resize. For example, if we made the array larger and copied 1 Million elements, then there will be 1 Million preceding operations, where we do not have to do an expensive resize. We say that the *amortized* running time<sup>7</sup> of push and pop is  $O(1)$ . Thus, concerning the running time in terms of O-notation, the dynamic array implementation is not worse than the linked nodes implementation.

## 14.4 Queues

Next, we consider the abstract data type *Queue*. In a queue we can store elements of *arbitrary* types. We can insert and remove elements. The queue is a so called *First-In-First-Out*-storage (FIFO-principle), i.e., we always remove the oldest element of the stack. The specification looks as follows:

```
trait MyQueue[A]:
  // Objects of an arbitrary but fixed type A are stored in a queue.

  // Precondition: None
  // Result: None
  // Effect: x is now the most recent element in the queue.
  def enqueue(x : A) : Unit

  // Precondition: Queue is not empty.
  // Result: The oldest element is returned.
  // Effect: The oldest element is removed from the queue.
  def dequeue() : A

  // Precondition: None
  // Effect: None
  // Result: true is returned if and only if the queue has no elements.
  def isEmpty : Boolean

  // Precondition: None
  // Effect: None
  // Result: The number of elements in the queue is returned.
  def size : Int
```

<sup>7</sup>This is a much better measure than the worst-case analysis. You will learn about this in higher classes.

We can imagine a queue in a well-structured supermarket: the customers that enqueue first at the cash desk will be handled at first. Customers arriving later at the cash desk will be handled later.

The operating system makes intensive use of queues, for example to schedule different processes or for printer spooling.

In graph theory we use queues as data structures to implement breadth-first-search – a very important algorithm<sup>8</sup> that is used to compute shortest paths between two vertices.

This is how we imagine an empty queue or a queue with three elements:



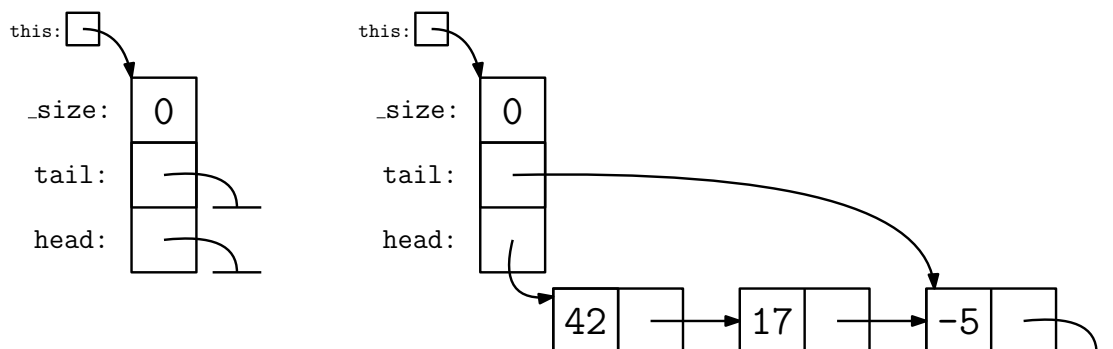
### 14.4.1 LinkedNodes implementation

Again, we start with an implementation of the queues using linked nodes. Our class is then called **LinkedNodesQueue**. As before, we have an inner class called **Node** that stores an element as well as a reference to the next element in the queue. Moreover, in the header we store the actual size of the stack as well as two references: the head and the tail of the queue, they are initially **null**.

```
class LinkedNodesQueue[A] extends MyQueue[A]:
  // The inner class:
  private class Node(val item : A, var next : Node)

  // Header:
  private var head : Node = null
  private var tail : Node = null
  private var _size : Int = 0
  // [...]
```

Here is an example how an empty queue or a queue with three elements might look like:



<sup>8</sup>Spoiler alert: You learn this in „Discrete Structures“ and „Algorithms and Data Structures“

We skip the implementation of the methods `isEmpty` and `size` since they are straightforward. Let us first focus on the inserting-method. In the `enqueue`-method we must distinguish two cases: either the queue is empty or not. If the queue is empty we create a new node with the enqueued element and `null` as successor and let `head` and `tail` reference this node. If otherwise, the queue is non-empty, we enqueue the new node next to the `tail` (which cannot be `null`). After that, we update `tail` to the new node. Finally, we increment the size.

```
class LinkedNodesQueue[A] extends MyQueue[A]:  
  // [...]  
  def enqueue(item: A): Unit =  
    if !isEmpty then  
      tail.next = Node(item, null)  
      tail = tail.next  
    else  
      tail = Node(item, null)  
      head = tail  
      _size = _size + 1  
  // [...]
```

In the `dequeue()`-operation we first check, whether the queue is empty or not. If not, we throw an exception as usual. If not, we store the item of the head in a temporary variable which is later returned. After that, we update `head` to `head.next`. If this was the last element of the queue (i.e., if `head` is now `null`), we have to set `tail` to `null`. Finally, we decrement the size.

```
class LinkedNodesQueue[A] extends MyQueue[A]:  
  // [...]  
  def dequeue(): A =  
    if !isEmpty then  
      val result: A = head.item  
      head = head.next  
      if head == null then  
        tail = null  
      _size = _size - 1  
      result  
    else throw Exception("Queue is empty")  
  // [...]
```

### 14.4.2 Array implementation

Last but not least, we consider an implementation of the queue using arrays of static size. The header of the implementation contains three different variables: we have an array (that stores the elements) and two indices `front` and `back`. The index of the frontmost/first element in the queue is `array(front)`. The elements of the queue are stored in the order of their arrival in the queue, with a possible wrap-around technique. That means the successor of the element in the last position of the array is stored at

the first position of the array. Moreover, back is the index of the first free slot in the array after the elements of the queue.

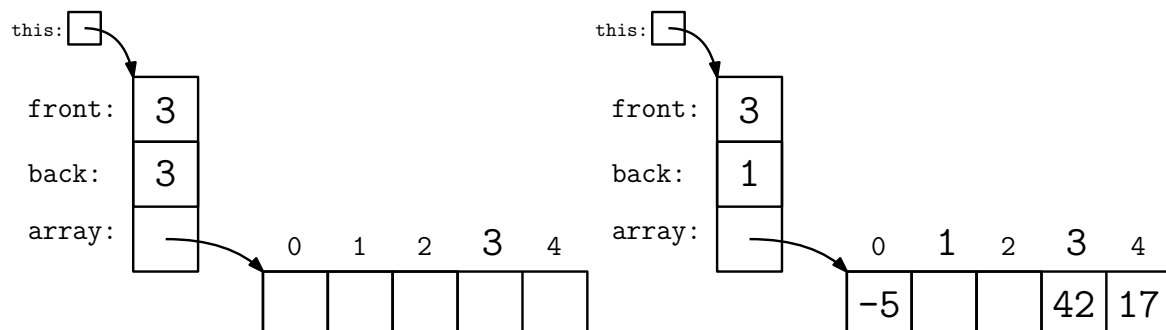
Since we use an array of static size, the queue has an upper bound of elements, which is called capacity. This parameter has to be passed to the constructor of the **ArrayStack**.

```
class ArrayQueue[A : ClassTag](capacity: Int) extends MyQueue[A]:
  // The header:
  private val n : Int = 1 + (if capacity < 1 then 1 else capacity)
  private val array: Array[A] = new Array[A](n)
  private var front: Int = 0
  private var back: Int = 0
  // [...]
```

Here, we store `array.length` in the variable `n`, since it will make the code a bit more clear. Thus, the array will have the positions  $0, \dots, n-1$  and to ensure that `front` and `back` are always between 0 and  $n-1$  we compute modulo  $n$ . The queue is interpreted as being empty if and only if `front == back`. In order to avoid ambiguities, we interpret the queue as full if and only if  $(back + 1) \% n == front$ .

However, this means that `back` always indicates an empty slot in the array, so that there must always exist an empty position in the array. This implies that the capacity of the queue is one less than the number of positions in the array. Such a construction is called *ring buffer* because we imagine that the elements are arranged in a cyclic order. Initially, we have `front == back == 0`, so the queue is empty.

Here is an example how an empty queue or a queue with three elements might look like:



Next, we consider the `enqueue`-method. First, we check whether the queue is full, i.e., whether  $(back + 1) \% n == front$ . If not, we store the element in the array at position `back` and update the value `back` to the next position in the cyclic ordering.

```
class ArrayQueue[A : ClassTag](capacity: Int) extends MyQueue[A]:
  // [...]
  def enqueue(x : A): Unit =
    // The next position after back
    // in the cycling ordering:
    val newBack: Int = (back + 1) \% n
```

```
if newBack != front then // stack is not full??
    array(back) = x
    back = newBack
else throw new Exception("The queue is full")
// [...]
```

For the dequeue-method we must first check, whether the queue is non-empty. If yes, we store the first element temporarily and return it later. We remove the object from the array by setting the content to **null**. Finally, we move the index front one step further in the cycling ordering.

```
class ArrayQueue[A : ClassTag](capacity: Int) extends MyQueue[A]:
  // [...]
  def dequeue() : A =
    if !isEmpty then
      val result : A = array(front)
      array(front) = null.asInstanceOf[A]
      // one position further in the cyclic ordering
      front = (front + 1) % n
      result
    else throw new Exception("The queue is empty")
  // [...]
```

**Remarks.** Finally, both implementations can be compared according to the known criteria. Furthermore, it is again possible to use an array of dynamic size to implement the queue. Here, one should be careful with the `resize()`-method. We cannot copy it one to one, because the wrap-around technique could make some trouble here.



## KAPITEL 15

### Priority Queue

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP4** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *objektorientierter Programmierkonzepte*.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP8** Du analysierst *Algorithmen*, indem du *Korrektheit* (auf Grundlage der Spezifikation), *Speicherplatz* und *Laufzeit* angibst und begründest.
- KdP9** Du beschreibst unterschiedliche Implementierungen *abstrakter Datentypen* und vergleichst diese miteinander.



#### 15.1 ADT PrioQueue

In this unit we want to consider an abstract data type called *Priority Queue* (short: PrioQueue). The PrioQueue works quite similar to the simple queue, but this time an element has a *priority*, i.e., it is a value of totally ordered universe **K**. The smaller this value, the more important it is, and hence, the earlier it has to be dequeued from the queue. Thus, we have again an inserting-method and a removing-method, like the simple queue has. But this time the removing-method always removes the smallest element from the queue. Here, we have the specification of the abstract data type<sup>1</sup>

```
trait MyPrioQueue[K : Ordering]:  
  // In a PrioQueue we store elements from a totally  
  // ordered universe.
```

<sup>1</sup>If you want you can of course add the `size`-method.

```
// Precondition: None
// Effect: The element key is added to the priority queue.
// Result: None
def insert(key : K) : Unit

// Precondition: The queue is non-empty.
// Effect: A smallest element is removed from the priority queue.
// Result: A smallest element of the priority queue is returned.
def extractMin() : K

// Precondition: None
// Effect: None
// Result: true is returned if and only if
// the priority queue has no elements.
def isEmpty : Boolean
```

To address the issue that the elements have to be of a totally ordered universe, we require **K** to be an instance of the class **Ordering**. However, it is also possible to have key-value-pairs in a priority queue – we call this variant **PairPrioQueue**. In this case the signatures (and surely the specifications) slightly change to

```
def insert(key : K, value : V) : Unit
def extractMin() : V
```

The method `extractMin()` will then remove a value `v` whose key is minimal among all other keys. However, we will consider the simpler case where the keys are the values.

**Applications.** Here are some applications of priority queues:

- In operating systems priority queues are used to organize processes based on their urgency and importance. This helps the operating system efficiently schedule and execute processes.
- When transmitting data packets over a network, priority queues can be used to give higher priority to specific types of traffic (e.g., voice or video data) and ensure that these data are transmitted with higher precedence.
- In database management systems, priority queues are employed to process queries or transactions with higher priority more quickly.
- In various algorithms priority queues can be used to make the selection and processing of elements more efficient. For example the famous shortest-path algorithm of Dijkstra uses a priority queue<sup>2</sup>.
- In printing systems priority queues are utilized to organize print jobs with different levels of urgency, ensuring that important documents are printed first.

<sup>2</sup>Spoiler alert: You will learn this in the lecture „Algorithms and Data Structures“

- In many applications and operating systems, priority queues are used to schedule and execute tasks or threads with different priorities<sup>3</sup>.

### 15.1.1 Sorting with priority queues

Let us consider the following algorithm:

```
def prioQueueSort[K : Ordering](array : Array[K]) : Unit =  
  val n = array.length  
  val pq : MyPrioQueue[K] = PrioQueueImplementation[K]()  
  for i <- 0 to n-1 do pq.insert(array(i))  
  for i <- 0 to n-1 do array(i) = pq.extractMin()
```

This algorithm takes an array as input and first, creates an empty priority queue (using an arbitrary implementation). Then, all elements of the input array are inserted in the priority queue. Finally, the elements are then removed step-by-step from the priority queue and inserted in increasing order in the array. Hence, we have a new general sorting algorithm. We can use this algorithm to test the implementations of a priority queue.

### 15.1.2 LinkedNodes Implementation

Let us first focus on an implementation of the priority queue using a linear ordering of linked nodes. We have already seen this when implementing stacks and simple queues. However, we can implement the priority queue with linked nodes in two different ways.

**Variant 1:** The elements are stored in increasing order. This will make the method `extractMin()` quite easy, because the smallest element will be in the first node and we can simply remove and return it. This costs  $O(1)$  time. On the other way, the inserting method might cost a lot of time, because in the worst, we have to walk through a large amount of nodes to insert a new node. Thus, in the worst-case this costs  $O(n)$  time. Later, we will take a closer look at this implementation.

**Variant 2:** Here, the elements are stored in an arbitrary order. This will then make the inserting method quite easy, because we can simply add a new node to the chain without walking to any position. It costs  $O(1)$  time. However, the `extractMin()` method is now the main bottleneck, because since the elements are not ordered we have to walk through the entire chain to find the smallest element – this costs  $O(n)$  time. We will not take a closer look at this implementation.

---

<sup>3</sup>Spoiler alert: You will learn this in the lectures of the technical computer science

**The dummy-node technique.** We want to implement the first variant, where it is easy to extract the smallest element but hard to insert a new element. But first, we briefly discuss a technique that is called *dummy-node technique*. Recall that we used references to nodes called *head* or *tail* for the implementations of the stack or the simple queue. Then we said that the queue/stack is empty, if the references of *head* and *tail* point to **null**. In the implementation of the stack this was not a problem, but in the implementation of the queue we had to consider several cases – this made the implementation harder to understand and more prone to implementation mistakes.

This problem will occur in the implementation of the priority queue. We solve this by introducing a dummy node which does not store an element but of the same node-type is the other nodes. The header of the implementation does not have a *head* or a *tail*, instead it is called *anchor* and stores the reference to the dummy node. Thus, the representation of the empty priority queue has one node – the dummy node – and the priority queue is empty if and only if the next node after this anchor is **null**. We remove and insert elements after the dummy node. The dummy node itself is never removed. This seems to be cumbersome but it will make the implementation itself much more easier because we do not need to consider any special cases.

Here, we have the first lines of the implementation using the dummy-node technique:

```
class LinkedSortedNodes[K : Ordering]() extends MyPrioQueue[K]:
  private val ord = summon[Ordering[K]]
  import ord.mkOrderingOps
  // The inner class:
  private class Node(val item : K, var next : Node)

  // Header:
  private var anchor : Node = Node(null.asInstanceOf[K], null) // the dummy-node

  def isEmpty : Boolean = anchor.next == null
  // [...]
```

Having this idea in mind we can now have a look at the easy-to-implement `extractMin()` method. Of course, we first check – as usual – whether the prioqueue is empty or not. If not, the smallest key must be in the node *next* to the anchor. We store this element (and return it later). Everything we have to do now is to update `anchor.next`.

```
class LinkedSortedNodes[K : Ordering]() extends MyPrioQueue[K]:
  // [...]
  def extractMin() : K =
    if isEmpty then throw Exception("Queue is empty")
    val result : K = anchor.next.item
    anchor.next = anchor.next.next
    result
  // [...]
```

The implementation is easy in the sense that we do not consider any special cases like what happens if we only have one element, i.e., if `anchor.next.next == null`. Now, for the `insert`-method we have to walk through the chain of nodes until we find the right position for our element. With `temp = anchor` we start at the anchor and use `temp = temp.next` to make a step to a new node. This step is repeated until we reached the end of the chain (`temp.next == null`) or we found an element that is larger than the new element (`temp.next.item > key`). Once we found the correct position in the chain, we insert a new node with the new element and corresponding reference to the next node.

```
class LinkedSortedNodes[K : Ordering]() extends MyPrioQueue[K]:
  // [...]
  def insert(key : K): Unit =
    var temp : Node = anchor
    while temp.next != null && temp.next.item <= key do
      temp = temp.next
    temp.next = Node(key, temp.next)
```

Again, the implementation is straightforward and we do not need to consider any special cases for example the prioqueue is empty or there is only one element or the new element is the smallest/largest. All these special cases work fine with the implementation.

## 15.2 Array Implementation

The linked nodes implementations had the great advantage that they are easy to implement (once we use the dummy-node technique). However, the implementation either have a very expensive `insert`-method or a very expensive `remove`-method. Moreover, the caching behaviour is bad.

In this section, we describe (and implement) the prioqueue with arrays of static size<sup>4</sup>. The implementation has the disadvantage to be much more complex. Nevertheless, we get better running times of the important methods (and better caching behaviour).

In the array implementation we will interpret the array as *binary heap*. Before we can give a definition of the notion we have to introduce the concept of *binary tree*.

### 15.2.1 Binary Trees

A *binary tree* is a non-linear structure that consists of nodes that can store elements. Each node has a *parent* node and at most two *child* nodes<sup>5</sup> – a *left* and a *right* child. The node whose parent is `null` is called *root of the tree*. Nodes that have no children are called *leafs*, and nodes that have at least one child are called *inner nodes*. A node represented as inner class in Scala could look as follows:

<sup>4</sup>We have already learned how to extend this to arrays of dynamic size.

<sup>5</sup>In *ternary trees* nodes have at most *three* child nodes, and so on ...

```
private class Node(parent : Node, elem : A, lchild : Node, rchild : Node)
```

Let  $T$  be a binary tree with root  $r$  and let  $v$  be any node in the tree. The number of steps from  $v$  to the root  $r$  is called the *depth* of  $v$ . The depth of the root is 0 because we need 0 steps from the root to itself. The children of the root have depth 1, and so on. *Level  $i$*  of the tree  $T$  is the set of all nodes of depth  $i$ . The *height* of  $T$  is the largest depth among all nodes, i.e., it is the length of the longest path from the root to a leaf. The *size* of  $T$  is the number of nodes in  $T$ .

A binary tree that has no nodes is called *empty tree*. It has size 0 and height  $-1$ .

### 15.2.2 Binary Heaps

A *binary min-heap*  $T$  is a binary tree whose elements are from a totally ordered universe and which satisfies the following two properties<sup>6</sup>

- (i) **Ordering property:** The values of the nodes are smaller or equal to the values of the corresponding child nodes.
- (ii) **Completeness property:** Let  $h$  be the height of  $T$ . Then the levels  $0 \dots h-1$  are completely filled and the leafs in level  $h$  are left-aligned.

Here, we have some important facts on min-heaps.

**Lemma 15.1.** *The minimum of the elements of a min-heap is stored in the root.*

**Lemma 15.2.** *Let  $n$  be the size and  $h$  the height of a min-heap. Then we have  $h \in O(\log n)$ .*

**Array-representation.** The most important property of a binary heap is that it can be represented as array. A position 0 we have the root, at positions 1–2 we have the children of the root, at positions 3–6 we have the children of the children of the root (from left to right), and so on. In general, the elements of level  $i$  are at positions  $2^i - 1$  to  $2^{i+1} - 2$  for  $0 < i < h$ . Since the elements of the level  $h$  are left-aligned (completeness property), they are stored positions  $2^h - 1$  until the end.

Let  $v$  be an element in the heap that is located in the corresponding array at position  $i$  then we can compute the positions of its parent, left child, and right child as follows:

```
private def parent(i : Int) : Int = (i - 1) / 2
private def lchild(i : Int) : Int = 2 * i + 1
private def rchild(i : Int) : Int = 2 * i + 2
```

Therefore, although binary heaps can be implemented using linked nodes, we use an array representation.

<sup>6</sup>In a similar way we can define *binary max-heaps*. It is also possible to define *ternary min/max-heaps*.

### 15.2.3 Header

The header of the heap-implementation looks as follows:

```
class BinaryHeap[K : Ordering : ClassTag](capacity: Int) extends MyPrioQueue[K]:
  private val ord = summon[Ordering[K]]
  import ord.mkOrderingOps

  private val array : Array[K] = new Array[K](if capacity < 2 then 2 else capacity)
  private var last : Int = -1

  private def parent(i : Int) : Int = (i - 1) / 2
  private def lchild(i : Int) : Int = 2 * i + 1
  private def rchild(i : Int) : Int = 2 * i + 2

  private def swap(i : Int, j : Int) : Unit =
    val temp = array(i)
    array(i) = array(j)
    array(j) = temp

  def isEmpty : Boolean = last == -1
```

We allocate an array of a fixed size as usual. Moreover, we need an index that we call *last*, it is the index of the right-most element in level  $h$ . The heap is empty, if and only if there is no element, i.e., if `last == -1`. Finally, the support method `swap(i, j)` swaps the elements at positions  $i$  and  $j$  in the array.

### 15.2.4 Insertion

For the insertion of a new element we assume that the array is not full yet. We insert the element to the right of the last position. That means, the element is inserted to the right of the right-most element of level  $h$ , or, if level  $h$  is completely full, the new element is now the left-most element in a new level  $h + 1$ .

```
def insert(key : K) : Unit =
  if last == array.size - 1 then throw Exception("The heap is full.")
  last = last + 1
  array(last) = key
  bubbleUp(last)
```

The completeness property is still satisfied but the ordering property might now be violated – in the worst-case the new element is the smallest among all elements and have to be in the root. To repair the heap, we finally call a method called `bubbleUp` that *bubbles* the new element upwards until the ordering property is fulfilled, i.e., until the parent of the new element is smaller or equal.

Let us consider the implementation of `bubbleUp`:

```
private def bubbleUp(i : Int) : Unit =
  if array(parent(i)) > array(i) then
    swap(i, parent(i))
    bubbleUp(parent(i))
```

If the ordering property is violated, i.e., if  $\text{array}(\text{parent}(i)) > \text{array}(i)$ , we swap the element at position  $i$  with its parent and recursively bubble the upwards. If we reach the case  $i=0$  then we are at the root and have  $\text{parent}(0)=0$ , and thus, the procedure terminates as well<sup>7</sup>.

**Running time.** The crucial part of the insertion is the bubble-up process. In the worst-case the inserted element is the new minimum and we have to bubble it upwards from a leaf to the root. Thus, the running time is in the worst-case  $O(h)$ , where  $h$  is the height of the heap. By Lemma 2, we know  $h \in O(\log n)$  where  $n$  is the size of the heap. In conclusion, the running time of the insertion is  $O(\log n)$ .

### 15.2.5 Deletion

For the deletion of an element we first store  $\text{array}(0)$  temporarily (and return it later). Then we remove it from the array. Thus, we have a hole in the root, which has to be filled. For this we use the last element in the array and swap it to the root.

```
def extractMin() : K =
  if isEmpty then throw Exception("The heap is empty.")
  val result = array(0)
  array(0) = null.asInstanceOf[K]
  swap(0, last)
  last = last - 1
  bubbleDown(0)
  result
```

The completeness property is satisfied but the ordering might be violated: the new root could now be the largest element. We use a bubble-down technique that moves the large element downwards in the tree as long as there is at least one smaller child.

The implementation of `bubbleDown` looks a bit more complex. This comes from the fact that we always have to check both children, whereas in the bubble-up process we only had to consider the parent node.

```
private def bubbleDown(i : Int) : Unit =
  if lchild(i) <= last && array(lchild(i)) < array(i) ||
    rchild(i) <= last && array(rchild(i)) < array(i)
  then
    val child = if rchild(i) <= last && array(rchild(i)) < array(lchild(i))
                  then rchild(i) else lchild(i)
```

<sup>7</sup>This works in Scala because  $(-1)/2$  is evaluated to 0



```
swap(i, child)
bubbleDown(child)
```

First, we check whether there is at least one child, that smaller. If not, the method terminates. Otherwise, we have to check which child is smaller, because we have to swap with the smaller child to satisfy the ordering property. After the swap we recurse on the child node. The statements `lchild(i) <= last` and `rchild(i) <= last` are used to check whether there are any children.

**Running time.** The crucial part of `extractMin()` is the bubble-down process. In the worst-case we have to bubble the element down to a leaf. Thus, the running time is in the worst-case  $O(h)$ , where  $h$  is the height of the heap. By Lemma 2, we know  $h \in O(\log n)$  where  $n$  is the size of the heap. In conclusion, the running time of `extractMin()` is  $O(\log n)$ , as well.

### 15.3 Sorting with a PrioQueue

Let us recall the sorting algorithm we have discussed earlier:

```
def prioQueueSort[K : Ordering](array : Array[K]) : Unit =
  val n = array.length
  val pq : MyPrioQueue[K] = PrioQueueImplementation[K]()
  for i <- 0 to n-1 do pq.insert(array(i))
  for i <- 0 to n-1 do array(i) = pq.extractMin()
```

What is the running time of this sorting algorithm? This surely depends on the concrete implementation. But we can make a general worst-case analysis. Let  $T_{in}(n)$  be the running time of the insert-method and let  $T_{out}(n)$  be the running time of the remove-method when there are  $n$  elements in the priority queue. Then we can determine the running time  $T(n)$  of the sorting algorithm easily as follows:

$$\begin{aligned} T(n) &= T_{in}(0) + T_{in}(1) + \dots + T_{in}(n-2) + T_{in}(n-1) \\ &\quad + T_{out}(n) + T_{out}(n-1) + \dots + T_{out}(2) + T_{out}(1) \\ &= \sum_{i=0}^{n-1} T_{in}(i) + \sum_{i=1}^n T_{out}(i) = \sum_{i=1}^n (T_{in}(i-1) + T_{out}(i)) \leq n \cdot (T_{in}(n) + T_{out}(n)) \end{aligned}$$

Hence, in conclusion we can say

$$T(n) \leq n \cdot (T_{in}(n) + T_{out}(n)).$$

#### 15.3.1 Insertionsort and Selectionsort

Let us consider the first variant of the linked-nodes implementation. We already discussed that in this case  $T_{in}(n) \in O(n)$  and  $T_{out}(n) \in O(1)$ . We apply the inequality

and conclude that in this implementation we have  $T(n) \in O(n^2)$ . Moreover, taking a closer look we might notice that this algorithm is the *insertion sort* algorithm (in some sense), because we always insert new elements in an ordered sequence.

On the other way, if we consider the second variant of the linked-nodes implementation we get  $T_{in}(n) \in O(1)$  and  $T_{out}(n) \in O(n)$  and hence,  $T(n) \in O(n^2)$  again. Taking a closer look we notice that this algorithm is the *selection sort* algorithm (in some sense), because we always select the smallest element in an unsorted sequence of elements.

### 15.3.2 Heapsort

Finally, we consider the implementation of the priority queue using a binary heap. This algorithm has the name *heapsort*, because it sorts elements by using a binary heap. We know that both insertion and deletion have running times  $O(\log n)$ . Thus, we can conclude the following property.

**Satz 15.3.** *The sorting algorithm heapsort has worst-case running time  $O(n \log n)$ .*

Let us have a closer look at the other important properties associated to sorting algorithms:

- Memory usage: For the general sorting algorithm we used a priority queue as additional storage. Thus, we use a linear amount of space and therefore, this implementation of heapsort is out-of-place.
- Heapsort is not stable, because the heap-structure removes any ordering of elements of the same value.<sup>8</sup>

### 15.3.3 In-place Heapsort

There is a chance to get rid off the huge storage amount and to find an in-place implementation. The main problem of the previous implementation is that we used the priority queue as blackbox. Thus, without knowing, the elements are first copied from one array to another. Instead, we will only use the two bubbling processes to work directly on the input array.

This is the important part of the implementation of the in-place heapsort algorithm:

```
def heapsort[K : Ordering](array : Array[K]) : Unit =
  // [...]
  val n = array.length
  for last <- 1 to n-1 do
    bubbleUp(last)
  for last <- n-1 to 1 by -1 do
    swap(0, last)
    bubbleDown(0, last-1)
```

<sup>8</sup>I would like to invite you to find a concrete example that shows that heapsort is not stable.

The process works as follows: We interpret the input array step-by-step as heap (`last` increases) and in each step we retrieve the ordering property (`bubbleUp(last)`) until the entire array is a heap. In the second phase we successively swap the minimum of the heap with the last element of the heap (`swap(0, last)`) and retrieve the ordering property of the remaining heap (`bubbleDown(0, last-1)`). Thus, the `bubbleDown`-method now also needs as input up to which position the array shall be interpreted as heap. In this phase, the size of the heap decreases in every step (`last` decreases). Finally, we achieve an unstable, in-place sorting algorithm of worst-case running time  $O(n \log n)$ .

### Dictionaries and Binary Search Trees

#### Lernziele

- KdP1** Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.
- KdP4** Du implementierst gut strukturierte *Scala-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *objektorientierter Programmierkonzepte*.
- KdP6** Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.
- KdP8** Du analysierst *Algorithmen*, indem du *Korrektheit* (auf Grundlage der Spezifikation), *Speicherplatz* und *Laufzeit* angibst und begründest.
- KdP9** Du beschreibst unterschiedliche Implementierungen *abstrakter Datentypen* und vergleichst diese miteinander.



#### 16.1 Abstract Data Type

In this last chapter we consider a last abstract data type that has lots of applications: the *dictionary*. In a time before things like internet, Wikipedia and AIs, dictionaries were books having for different words (the *keys*) different entries (the *values*) like for example translations into other languages or further explanations. Wikipedia in this sense is still a dictionary: for different keywords you find corresponding articles which are the values. In computer science, a dictionary has for each key at most one value, i.e., the same key will not occur more than time.

We have at least three types of methods on dictionaries. First of all, we must be able to *search* a given key and check whether it is included and if so what its value is. Next, we need some method to *remove* a given key and its associated value. Finally, the other way round, we need a method to *insert* a new key-value-pair into the dictionary. This is the specification of the abstract data type we want to consider:

```
trait MyDict[K, V]:  
  // In a dictionary we store pairs of keys of data type K and  
  // values of data type V. For each key there is at most one value.  
  
  // Precondition: None  
  // Effect: The dictionary contains key with the given value.  
  def put(key : K, value: V): Unit  
  
  // Precondition: None  
  // Effect: The dictionary does not contain key.  
  def remove(key : K) : Unit  
  
  // Precondition: The dictionary contains key.  
  // Result: The corresponding value of key in the dictionary is returned.  
  def get(key : K): V  
  
  // Precondition: None  
  // Result: True is returned, if and only if the dictionary contains key.  
  def contains(key : K) : Boolean  
  
  // Precondition: None  
  // Result: True is returned, if and only if the dictionary is empty.  
  def isEmpty : Boolean
```

**Applications.** Let us consider some applications of dictionaries:

- Dictionaries are used in *translation services* to translate words or sentences from one language to another. They serve as the basis for mapping words between different languages.
- In *compiler development*, dictionaries are used to manage symbols and tokens. Symbols can be associated with tokens, facilitating the process of translating source code into machine code.
- In *databases*, dictionaries are often used for fast access to records. Primary keys serve as unique identifiers for the records.
- In *word processing programs*, dictionaries are used for spell checking and autocorrection. Words and their correct spellings are stored, enabling automatic error detection and correction.
- Dictionaries are used in *software configuration* to manage configuration parameters along with their corresponding values. This can occur in various contexts such as application or system configurations.
- In *network communication*, dictionaries can be used to manage protocol headers and parameters. They facilitate the interpretation of network messages and contribute to efficient communication between systems.

## 16.1.1 Implementation with linked nodes

A dictionary can be implemented with linearly linked nodes like we did with queues or stacks. A node could be represented as follows:

```
private class Node(key : K, value : V, next : Node)
```

We will not give the complete implementation, but instead present the overall idea and analyse it. In the header, we have an anchor reference to one of the nodes to get access to all other nodes in the chain.

**get(key) and contains(key).** To find a given key is in the dictionary, we have to check every node in the worst-case. Once we have found the key in one of the nodes, we can either return **true** or the associated value. If no node contains the key, we return **false** or throw an exception. This last case costs time  $O(n)$  in the worst-case, assuming that  $n$  is the number of key-value-pairs in the dictionary.

**remove(key).** To remove a key and its associated value we walk through the linked nodes to find the node. Once we have found it, we update the links of the nodes to remove the node with the key. If there is no such node, we do not have to update anything, but still we would have a look into each node. Thus, the worst-case running time is again  $O(n)$ .

**put(key, value).** Finally, for the put-method, we proceed very similar. We scan all the nodes to check whether the key is already contained in the dictionary. If yes, we update the corresponding value. If not, we insert a new node at the end of the chain and store the key-value-pair in the new node. However, in this worst-case the method has running time  $O(n)$ .

## 16.1.2 Implementation with arrays

A dictionary can naively be implemented with an array. For this, each entry of the array is a key-value-pair and we use the counter  $n$  to save the current number of key-value-pairs. Of course, the array can be of static or dynamic size. Again, we do not give the complete implementation for the methods.

**get(key) and contains(key).** To find a given key is in the dictionary, we have to check all  $n$  entries in the worst-case. Once we have found the key in one of the entries, we can either return **true** or the associated value. If no entry contains the key, we return **false** or throw an exception. This last case costs time  $O(n)$  in the worst-case.

**remove(key).** To remove a key and its associated value we check the array entries to find the key. Once we have found the key at a position  $i$ , we swap the pairs at positions  $i$  and  $n - 1$  and remove the pair at position  $n - 1$ . Finally, we decrement  $n$ . If there is no such entry, we do not have to update anything, but still we would have a look at  $n$  positions of the array. Thus, the worst-case running time is again  $O(n)$ .

**put(key, value).** Finally, for the put-method, we proceed very similar. We scan all the positions to check whether the key is already contained in the dictionary. If yes, we update the corresponding value. If not, we insert the key-value pair at position  $n$  and increment the counter. Of course, if the array is full, we either resize or throw an exception. However, in the worst-case the method has running time  $O(n)$ .

### 16.1.3 Conclusion

Using simple ideas for the implementation of dictionaries leads to very bad running times of the methods – all methods have worst-case running times  $O(n)$ . However, it is possible to get much better running times with array implementations. But then, we have to consider a technique called *hashing*, which we do not discuss in this class. Hashing allows to have amortized running times  $O(1)$ . But this technique comes with a lot of problems that have to be solved.

Instead we will restrict the set of keys and use an implementation with non-linear linked nodes.

## 16.2 Binary Search Trees

From now on, we consider the key set  $K$  to be an instance of **Ordering**[ $K$ ]. Thus, we consider  $K$  to be a totally ordered universe. At first sight, this is a technical restriction, but on the other hand in practise most of the key sets have a well-known ordering.

**Attention:** In the following, we discuss the theory of binary search trees. For simplicity, we forget the values, because they are unimportant in theory. Nevertheless, we keep in mind that each key arrives with a corresponding value.



### 16.2.1 Definition

Let  $T$  be a binary tree whose nodes contain elements of a totally ordered universe  $K$ . A *binary search tree* is defined as follows:

- (i) If  $T$  is the empty tree, then  $T$  is a binary search tree.
- (ii) If  $k$  is the root of  $T$  and  $T_l$  and  $T_r$  are the left and the right subtree of  $T$ , respectively. Then,  $T$  is a binary search tree, if all elements in  $T_l$  are smaller than  $k$ , all elements in  $T_r$  are larger than  $k$ , and  $T_l$  and  $T_r$  are themselves two binary search trees.

This definition of binary search trees is *inductive*, since we assume  $T_l$   $T_r$  to be binary search trees to conclude that the entire tree is a binary search tree.

**Attention:** Binary search trees and heaps are totally different concepts.



Here, we can see a potential private class that might be used to represent the nodes of a binary search tree in Scala:

```
private class Node(key : K, left : Node, right : Node)
```

It might be possible to add references to the parent node. However, this will not be necessary in our implementation. Moreover, in this representation we do not have values which we need for the dictionary implementation. When using binary search trees for dictionaries we could adapt the representation as follows:

```
private class Node(key : K, value : V, left : Node, right : Node)
```

Hence, the header of a corresponding class looks as follows:

```
class BinarySearchTree[K : Ordering, V] extends MyDict[K,V]:  
  private val ord = summon[Ordering[K]]  
  import ord.mkOrderingOps  
  
  private class Node(val key : K, var value : V, var left : Node, var right : Node)  
  private var root : Node = null  
  
  // empty tree:  
  def isEmpty : Boolean = root == null  
  // [...]
```

First, we import the ordering and its corresponding syntax to use it in the further implementation. Moreover, in the empty tree, the root is **null**.

A binary search tree has three main operations: we must be able to *search* a key, to *remove* a key (if it is contained in the tree), and to *insert* a key in the tree. These three operations can then be used to implement the dictionary methods. Additionally, since a binary search tree is defined inductively, we can use recursive implementations of the methods.

### 16.2.2 Searching

The general idea for finding a search-key in a binary search tree works as follows:

- (a) If the tree is empty, then the search-key is obviously not contained in the tree.
- (b) If the tree is not empty we consider root-key. Then three cases can occur:



- i If the root-key *matches* the search-key we are done and have found the search-key.
- ii If the root-key is *larger* than the search-key, the search-key can only occur in the left subtree, since all elements in the right subtree are larger than the root-key and therefore, larger than the search-key. We recursively search in the left subtree.
- iii If the root-key is *smaller* than the search-key, the search-key can only occur in the right subtree, since all elements in the left subtree are smaller than the root-key and therefore, smaller than the search-key. We recursively search in the right subtree.

Having this idea in mind, we can implement the *search*-method of a binary search tree as follows:

```
// Precondition: None
// Result: The node containing key is returned,
// if key is in the binary search tree rooted at curr.
// If key is not in the tree, null is returned.
private def search(curr : Node, key : K) : Node =
  if curr == null || curr.key == key then
    curr
  else if curr.key > key then
    search(curr.left, key) // go to the left subtree
  else // curr.key < key
    search(curr.right, key) // go to the right subtree
```

The first case in the implementation corresponds to the cases (a) and (b.i), the second case corresponds to (b.ii) and the third case corresponds to (b.iii). The implementation works for an arbitrary node and searches the key in the subtree rooted at *curr*. To find the key in the entire binary search tree, we start the method with the root. Thus, the two dictionary methods are implemented as follows:

```
def get(key : K) : V = search(root, key).value
def contains(key : K) : Boolean = search(root, key) != null
```

If *contains*(key) returns false, then *get*(key) will raise an exception, since *null*.value fails.

**Running time.** We make a constant number of operations on one node. In the best case, we simply stop, but in the worst-case we walk downwards the tree. The set of visited nodes in this process is called *search path*. The search path always starts at the root and in the worst-case ends in a leaf. Thus, in the worst-case the running time is proportional to length of the longest root-to-leaf-path, which is exactly the height of the tree. In conclusion, if *h* is the height of the tree, then the search-method needs running time  $O(h)$ .

## 16.2.3 Insertion

The insertion of an element in the tree works quite similar to the searching process: we compare the element with the keys of the nodes and either walk downwards the left or the right subtree if the new element does not match the keys. However, if we reach a leaf we add a new node with the element to left or the right of the leaf.

What happens if the element is already contained in the tree? In the context of dictionaries we simply update the associated value to the key and stop (see implementation). However, there is another possibility to handle this case. We can slightly change the inductive definition of the binary search tree by demanding that the elements of the left subtree are smaller *or equal* to the root. Then, whenever the key of a node matches the element that has to be inserted, we recursively walk to the left subtree, because in the left subtree we allow element to be equal to the root.

Nevertheless, we want to have a look at the implementation for the original definition and how it is used for the dictionary-method `put`.

```
// Precondition: Let curr be the root of a binary search tree T.
// Effect: If T contains a node with key, then its value is updated.
// If T does not contain key, a new node in T is inserted with
// content (key, value).
// Result: The root of the updated binary search tree is returned.
private def insert(curr : Node, key : K, value : V) : Node =
  if curr == null then // key is not contained: new leaf
    Node(key, value, null, null)
  else if curr.key == key then // key is contained: update
    curr.value = value
    curr
  else if curr.key > key then // insert in left subtree
    curr.left = insert(curr.left, key, value)
    curr
  else // curr.key < key // insert in right subtree
    curr.right = insert(curr.right, key, value)
    curr
```

We have the same four cases as before. If `curr` is `null`, a new node is inserted. If the key has been found we update its value. If the key does not match we either insert to the left or to the right subtree. For technical reasons, the root of the current tree is returned, which is either `curr` or the new node. Again, the method works for arbitrary nodes and their subtrees. To insert a new key-value-pair into the entire tree we have to start the method on the root.

```
def put(key : K, value : V) : Unit = { root = insert(root, key, value) }
```

**Running time.** During the insertion process we again walk along the search path of the key and maybe end in a leaf to include a new node to its left or right. Again, on each node we do only a constant number of steps. Thus, the running time of the insertion process is again  $O(h)$ , where  $h$  is the height of the tree.

**Height of a binary search tree.** As we have seen, the height of the tree plays an important role in the running time of the search- and insert-method. For heaps, we have seen that the height is in  $O(\log n)$  when  $n$  is the number of nodes. A binary search tree is in general no heap and we cannot argue like this. Moreover, if we consider the empty binary search tree and insert the number  $1, \dots, n$  in this order, the height of the tree will be  $n - 1$ , the resulting tree is called *degenerate*. Thus, the running times of the search- and insert-methods are still  $O(n)$  in the worst case. That is embarrassing since it seems the binary search trees are not much better than the naive implementations of the dictionary.

However, there are two good news: First, in practise it barely happens that the tree is degenerate. Secondly and most importantly, there are methods to restructure the trees, such that their height will not be higher than  $O(\log n)$ . There are many different approaches to restructure the tree<sup>1</sup>.

### 16.2.4 Removing

Finally, to give a full implementation of the dictionary we have to discuss how to remove elements from a binary search tree. For this, we first want to consider a special case that is then used for the general case.

**Removing the maximum.** Let  $T$  be a non-empty binary search tree. We want to consider the case of removing the node with the largest element. The idea here is quite simple: If a node has a non-empty right subtree then the maximum has to be in this subtree, since it contains elements that is larger than the element of the considered node. Moreover, if a node has an empty right subtree, then the element of the node has to be the maximum, since in the left subtree there are no larger elements. Thus, the approach of finding the largest element is: Go as long possible in the right subtree until there is no right subtree and output the element of the node we have stopped at.

The implementation in Scala will be recursive. It needs a node as input, which is considered to be the root. The function outputs the root of the tree in which the maximum has been removed. Moreover, for the dictionary implementation, the function will also output the largest key and its corresponding value.

```
// Precondition: Let curr be the root of a non-empty binary search tree T.
// Effect: The node with the largest key in T is removed.
// Result: The root of the updated binary search tree is returned.
// The largest key and its corresponding value is returned.
private def removeMax(curr : Node) : (K, V, Node) =
  if curr.right == null then // no right subtree => root = maximum
    (curr.key, curr.value, curr.left)
  else // otherwise find maximum in right subtree
    val (k, v, n) = removeMax(curr.right)
    curr.right = n
    (k, v, curr)
```

<sup>1</sup>Spoiler alert: You learn about this in „Algorithms and Data Structures“.

The implementation shows the two cases that have been discussed so far. In the first case – the right subtree is empty – the root of the left subtree is now the new root. For the second case, we apply the function recursively to `curr.right` to find the maximum here. This application returns the new root of the updated right subtree. Finally, we return the key, the value, and the original root of the entire tree.

**General case.** Let us next consider the general case. We are given the root of the tree and a search-key which is then deleted from the tree. The key can only occur at the corresponding search-path. Hence, we will walk along this search-path recursively: if the search-key is smaller than the root-key, go right (case 2), and if the search-key is larger than the root-key, go left (case 3). If the search ends in `null`, then the search-key is not contained in the tree, and nothing happens, since nothing has to be removed.

The more interesting case is what happens, if we have found the node that contains the search-key which has to be removed. If the node has no left subtree, we can simply replace it by the root of the right subtree and hence, the node is removed. This was the easy case. For the more complicated we assume that there is a right subtree. We cannot remove the node, instead we have to find a suitable key-value-pair that fits into this node, we override the key-value-pair that has to be removed. Since the left subtree is non-empty, we can remove and get the largest key (and its value) in the left subtree. This key is then used for the overriding. Furthermore, the largest element in the left subtree is a perfect candidate, since the binary search tree property is now fulfilled.

```
// Precondition: Let curr be the root of a binary search tree T.
// Effect: The binary search tree does not contain a node with key.
// Result: The root of the updated binary search tree is returned.
private def remove(curr : Node, key : K) : Node =
  if curr == null then // key is not in the tree
    curr
  else if curr.key > key then // remove in left subtree
    curr.left = remove(curr.left, key)
    curr
  else if curr.key < key then // remove in right subtree
    curr.right = remove(curr.right, key)
    curr
  else if curr.left == null then // curr.key == key
    curr.right
  else // curr.left != null && curr.key == key
    // remove largest element in left subtree to update the
    // current node, that originally contains key
    val (k, v, left): (K, V, Node) = removeMax(curr.left)
    Node(k, v, left, curr.right)
```

The last two lines of the function are important. First, we get the key-value-pair of the largest element in the left-subtree. Furthermore, this pair is removed from the left-subtree and its root is returned from `removeMax`. Finally, `Node(k,v,left,curr.right)` is the new root with updated references.

The remove function is of course started with the root of the tree:

```
def remove(key : K) : Unit = { root = remove(root, key) }
```

**Running time.** The first method – removing the maximum – works in running time  $O(h)$ , because in the worst case, the maximum is in deepest leaf and we walk along a root-to-leaf-path. The second method works in  $O(h)$  as well, because first we walk along a search-path that has length  $O(h)$  until we find the node (or the element is not contained). Finally, we apply one invocation of `removeMax`, which runs in  $O(h)$ .

### 16.2.5 Traversing a tree

Trees are non-linear structures. Sometimes, it is easier to work with linear structures like lists or arrays. A *traversing of a tree* is a function that computes a list of the elements of the tree. There are different approaches on how to do this. We will consider two variants: the *levelorder* and the *preorder*. The first one is also called *breadth first search* (bfs) while the second is a special variant of *depth first search* (dfs).

**Levelorder.** In the levelorder traversing we first visit the root, then we visit the nodes at level 1, from left to right, then we visit the nodes at level 2, from left to right, and so on. The implementation uses a queue:

```
// Precondition: None
// Result: A list of all keys of the tree is returned.
def bfs : List[K] =
  val queue : MyQueue[Node] = MyQueueImpl[Node]()
  queue.enqueue(root)
  var result : List[K] = List()
  while !queue.isEmpty do
    val curr = queue.dequeue()
    if curr != null then
      // Store key of current node in the list ...
      result = result :: List(curr.key)
      // ... and enqueue the two children of the node.
      queue.enqueue(curr.left)
      queue.enqueue(curr.right)
  result
```

We start by visiting the root. Whenever we visit a node with `val curr = queue.dequeue()` we enqueue its two child nodes in the queue. The key of the visited is stored in a result list. Since the left child is always the first element in the queue, the implementation ensures that the nodes are visited from left to right in one level.

**Preorder.** The preordering of the elements is much easier, since it works recursively. First, we put the element of the root into the list, then we recursively traverse all the elements in the left subtree and finally, we traverse all elements in the right subtree.

```
// Precondition: None
// Result: A list of all keys of the tree is returned.
def dfs : List[K] =
  def preorder(curr : Node) : List[K] =
    if curr == null then Nil
    else List(curr.key) ::: preorder(curr.left) ::: preorder(curr.right)
  // Use preorder starting at root
  preorder(root)
```

The depth first search algorithm can also be implemented by using two different approaches: for the *inordering* we use the line

```
else inorder(curr.left) ::: List(curr.key) ::: inorder(curr.right)
```

and for the *postordering* we use the line

```
else postorder(curr.left) ::: postorder(curr.right) ::: List(curr.key)
```

This gives different orderings but still lists of the elements.