Lectures Notes on Data Structures and Algorithms

Florian Rabe

2017

Contents

Part I

Introduction and Foundations

Chapter 1

Meta-Remarks

Important stuff that you should read carefully!

State of these notes I constantly work on my lecture notes. Therefore, keep in mind that:

- I am developing these notes in parallel with the lecture—they can grow or change throughout the semester.
- These notes are neither a subset nor a superset of the material discussed in the lecture.
- Unless mentioned otherwise, all material in these notes is exam-relevant (in addition to all material discussed in the lectures).

Collaboration on these notes I am writing these notes using LaTeX and storing them in a git repository on GitHub at https://github.com/florian-rabe/Teaching. Familiarity with LaTeX as well as Git and GitHub is not part of this lecture. But it is essential skill for you. Ask in the lecture if you have difficulty figuring it out on your own.

As an experiment in teaching, I am inviting all of you to collaborate on these lecture notes with me.

By forking and by submitting pull requests for this repository, you can suggest changes to these notes. For example, you are encouraged to:

- Fix typos and other errors.
- Add examples and diagrams that I develop on the board during lectures.
- Add solutions for the homeworks if I did not provide any (of course, I will only integrate solutions after the deadline).
- Add additional examples, exercises, or explanations that you came up or found in other sources. If you use material from other sources (e.g., by copying an diagram from some website), make sure that you have the license to use it and that you acknowledge sources appropriately!

The TAs and I will review and approve or reject the changes. If you make substantial contributions, I will list you as a contributor (i.e., something you can put in your CV).

Any improvement you make will not only help your fellow students, it will also increase your own understanding of the material. Therefore, I can give you up to 10% bonus credit for such contributions. (Make sure your git commits carry a user name that I can connect to you.) Because this is an experiment, I will have to figure out the details along the way.

Other Advice I maintain a list of useful advice for students at https://svn.kwarc.info/repos/frabe/Teaching/general/advice_for_students.pdf. It is mostly targeted at older students who work in individual projects with me (e.g., students who work on their BSc thesis). But much of it is useful for you already now or will become useful soon. So have a look.

Chapter 2

Basic Concepts

These lecture notes do not follow a particular textbook.
Students interested in additional literature may safely use [CLR10] (available online), one of the most widely used textbooks. Knuth's book series on the Art of Computer Programming [Knu73], although not usually used as a modern textbook, is also interesting as the most famous and historically significant book on the topic.

2.1 What are Data Structures and Algorithms?

Data structures and algorithms are among the most fundamental concepts in computer science.

2.1.1 Static vs. Dynamic

In all areas of life and science, we often find a pair of concept such that one concept captures static and the other one dynamic aspects. This is best understood by example:

area	static	dynamic
in life		
existence	be	become
events	situation	development
food	ingredients	cooking
in science		
mathematics	sets	functions
physics	space	time
chemistry	molecules	reactions
engineering	materials	construction
in computer science		
hardware	memory	processing
abstract machines	states	transitions
programming	types	functions
software design	**data structures**	**algorithms**

The static aspects describes things as they are at one point in time. The dynamic aspects describes how they change over time.
Data structures and algorithms have this role in software design. Data structures are sets of objects (the data) that describe the domain that our software is meant to be used for. Algorithms are operations that describe how the objects in that domain change.

2.1.2 Basic Definition and Examples

Definition 2.1 (Data Structure)**.** Assume some set of effective objects.
A data structure defines a subset of these objects by providing effective methods for determining

- whether an object is in the data structure or not,
- whether two objects are equal.

In practice, a data structure is often bundled with several algorithms for it.

Definition 2.2 (Algorithm)**.** An algorithm consists of

- a data structure that defines the possible input objects
- a data structure that defines the possible output objects
- an effective method for transforming an input object into an output object

These definitions are not very helpful—they define the words "data structure" and "algorithm" by using other not-defined words, namely "effective object" and "effective method". Let us look at some examples before discussing effective objects and methods in Sect. 2.1.3.

Example 2.3 (Natural Numbers)*.* The most important data structure are the natural numbers.
It is defined as follows:

- The string 0 is a natural number.
- If n is a natural number, then the string $s(n)$ is a natural number.
- All natural numbers are obtained by applying the previous step finitely many times, and these are all different.

We immediately define the usual abbreviations $1, 2, \ldots,$. It is also straightforward to define algorithms for the basic functions on natural numbers such as $m + n$, $m - n$, $m * n$, etc.

Example 2.4 (Euclidean Algorithm)*.* The Euclidean algorithm (see also Sect. 2.1.4) computes the greatest common divisor $\gcd(m, n)$ of two natural numbers $m, n \in \mathbb{N}$. It consists of the following components:

- input: $\mathbb{N} \times \mathbb{N}$
- output: $\mathbb{N}$
- effective method:

```
fun gcd(m : ℕ, n : ℕ) : ℕ =
    x := m                                   introduce variables, initialize with input data
    y := n
    while x ≠ y                              repeat as long as gcd(x, y) ≠ x
        if x < y          subtract the smaller number from the bigger one, which does not affect gcd(x, y)
            y := y − x
        else
            x := x − y
    return x                                 now trivially gcd(x, y) = x
```

The algorithm starts by introducing variables x and y and initializes them with the input data m and n. Then it repeatedly subtracts the smaller number from the greater one until both are equal. This works because $\gcd(x, y) = \gcd(x - y, y)$. If x and y are equal, we can return the output because $\gcd(x, x) = x$.

This algorithm has a subtle bug (Can you see it?) that we will fix in Ex. 3.13.

For a simpler example, consider the definition of the factorial $n! = 1 \cdot \ldots n$ for $n \in \mathbb{N}$.

Example 2.5 (Factorial)*.* The factorial can be defined as follows:

- input: $\mathbb{N}$
- output: $\mathbb{N}$

- effective method:

```
fun fact(n : ℕ) : ℕ =
    product := 1
    factor := 1
    while factor ≤ n
        product := product · factor
        factor := factor + 1
    return product
```

  Here the variable *factor* runs through all values from 1 to n and the variable *product* collects the product of those values.

Notation 2.6. It is convenient to give the effective method of an algorithm as a function definition using pseudo-code. That way the input and output do not have to be spelled out separately because they are clear from the data structures used in the header of the function definition.

2.1.3 Effective Objects and Methods

It is now a central task in computer science to define data structures and algorithms that correspond to given sets and functions. This question that was first asked by David Hilbert in 1920, one of the most influential mathematicians at the same time. In modern terminology, he wanted to define data structures for all sets and algorithms for all functions and then machines to mechanize all mathematics.

In the 1930s, several scientist worked on this problem and eventually realized that it cannot be done. These scientists included Alonzo Church, Kurt Gödel, John von Neumann, and Alan Turing. Their work provided partial solutions and theoretical limits to the problem. In retrospect, this was the birth of computer science.

Not every set and not every function can be represented by a data structure or an algorithm (see Sect. 2.1.5 for the reason why not). That limitations bring us back to the question of effective objects and methods:

Definition 2.7 (Effective Object)**.** An effective object is any object that can be stored, manipulated, and communicated by a physical machine.

Here, *physical* means any machine that we can build in the physical world.[1]

Thus, every physical machine defines its own kind of effective objects. All digital machines (which includes all modern computers) use the same effective objects: lists of bits. These are stored in memory or on hard drives, which provide essentially one very, very long list of bits.

Data structures use fragments of these lists to represents sets. For example, the set $\mathbb{Z}_{2^{32}}$ of 32-bit-integers is represented by a list of 32 bits.

Definition 2.8 (Effective Method)**.** An effective method consists of a sequence of instructions such that

- any reasonably intelligent human can carry out the instructions
- and all such humans will carry out the instructions in exactly the same way (in particular reaching the same result).

The first condition makes sure that any prior knowledge needed to understand the instructions is be explicitly stated or referenced. The second conditions makes sure that an effective method has a well-defined result: There may be no ambiguity, randomness, or unspecified choice.

Example 2.9. The third condition excludes for example the following instructions

- "Let x be the factorial of 5.": Different humans could compute the factorial differently because it is not clear which algorithm to use for the factorial.
- "Let x be a random integer.": Randomness is not allowed.
- "Let x be an element of the list l.": It is not specified which element should be chosen.

[1] Sometimes we use hypothetical machines. For example, quantum computers are physical machines that we think we can build but have not been able to build in practice yet (at least not at useful scales).

2.1.4 History

One of the earliest and most famous (arguably *the* earliest) algorithms is Euclid's algorithm for computing the greatest common divisor (see Ex. 2.4). It is given around 300 BC in Euclid's Elements [EucBC, Book VII, Proposition 2], maybe the most influential textbook of all time.

The word *algorithm* is much younger. It is derived from the name of the 9th century scientist al-Khwarizmi. He was one of the most important scientists of his millennium but is relatively unknown in the Western world because he was and wrote in Arabic. Translations of his work on arithmetic in the 12th century spread several new results to the Western world.

This included the use of numbers as abstract objects as opposed to geometric distances that had dominated Europe since the work of the Greek mathematicians (such as Euclid). It also included the positional number system and the base-10 digits that are still in use today. The corresponding arithmetical operations on numbers were named *algorismus* after him in Latin, which developed into the modern word. He also worked on algorithms for solving linear and quadratic equations, and one of his basic operations called *al-jabr* gave rise to the word *algebra*.

The modern *meaning* of the word *algorithm* is even younger: Its formalization was effected by a major development in the 1920s and 1930s that eventually gave to modern computer science itself. Hilbert was the most influential mathematician in the early 20th century. One of his legacies was to call for solutions to certain fundamental problems [Hil00]. Another legacy was his program [Hil26], a call for the formalization of mathematics that (among other things) should yield an algorithm for determining whether any given mathematical formula is a theorem.

Hilbert's program inspired seminal work by (among others) Alonzo Church, Kurt Gödel, and Alan Turing. This led to several concrete definitions of *algorithm*, including Turing-machines and the λ-calculus, from which all modern programming languages are derived. It also led to an understanding of the limits of what algorithms can do (see Sect. 2.1.5), which has led to the modern theory of computation.

2.1.5 The Limits of Data Structures and Algorithms

Countability of Data Structures and Algorithms

We can now see immediately why not all mathematical objects are effective in digital machines: There are only countably many lists of bits. Therefore, there can only be countably many effective objects.

Similarly, any data structure we define must be defined as a list of characters in some language. But there are only countably many such lists. Therefore, there can only be countably many data structures. For the same reason, there can only be countably many algorithms.

Inspecting the sizes of the constructed sets from Sect. A.5, we can observe that

- If all arguments are finite, so is the constructed set—except for lists.
- If all arguments are at most countable, so is the constructed set—except for function and power sets.

Because of these exceptions, we cannot restrict attention to finite or countable sets only—working with them invariably leads to uncountable sets.

Computability

At best, we can hope to give data structures for all countable sets. But not even that is possible. Because countable sets have uncountably many subsets, we cannot give data structures for every subset of every countable set.

Therefore, we give the sets that have data structures a special name:

Definition 2.10 (Decidable)**.** A set is called **decidable** if we can give a data structure for it.

Similarly, at best we can hope to give algorithms for all functions between decidable sets. Again that is not possible. Because countable sets have uncountably many functions between them, we cannot give algorithms for all functions between decidable sets.

Therefore, we give the sets that have data structures a special name:

Definition 2.11 (Computable)**.** A function between decidable sets is called **computable** if we can give an algorithm for it.

At Jacobs University, decidability and computability are discussed in detail in a special course in the 2*nd* year.

The Role of Programming Languages

Vagueness of the Definitions It is not possible to precisely define effective objects and methods—every definition eventually uses not-defined concepts like "machine" or "instruction". Thus, it impossible to precisely define data structures and algorithms are. Instead, we must assume those concepts to exist a priori.

That may seem flawed—but it is actually very normal. We can compare the situation to physics where there is also no precise definition of *space* and *time*. In fact, the question what space and time are is among the difficult of all of physics.[2]

Similarly, the question of what data structures and algorithms are is among the most fundamental of computer science. Every computer and evey programming language give their own answer to the question.

Data Description and Programming Languages To make the definitions of *data structure* and *algorithm* precise, we have to choose a concrete formal language.

Definition 2.12 (Languages)**.** A **data description language** is a formal language for writing objects and data structures.

A **programming language** is a formal language for writing algorithms.

Because algorithms require data structures, every programming language includes a data description language. And because all data structures usually come with specific algorithms, we are usually mostly interested in programming languages.

But there are some languages that are pure data description languages. These are useful when storing data on hard drives or when exchanging data between programs and computers (e.g., on the internet). Examples of pure data description languages are JSON, XML, HTML, and UML.

Types of Programming Languages Programming languages can vary widely in how they represent data structures.

We can distinguish several groups:

- Untyped languages avoid explicit definitions of data structures. Instead, they use algorithms such as *isNat* to check, e.g., if an object is a natural number.
  Examples are Javascript and Python.
- Functional languages focus on using inductive data types.
  Examples are SML and Haskell.
- Object-oriented languages focus on using classes.
  Examples are Java and C++.
- Multi-paradigm languages combine functional and object-oriented features.
  Examples are Scala and F#.

Independence of the Choice of Language Above we have seen that the concrete meaning of *data structure* and *algorithm* seems to depend on the choice of programming language. Thus, it seems that whether a set is decidable or a function computable also depends on the choice of programming language.

One of the most amazing and deepest results of theoretical computer science is that this is not the case:

Theorem 2.13 (Church-Turing Thesis)**.** *All known programming languages (including theoretical ones such as Turing machines)*

- *can define data structures for exactly the same sets,*

[2] For example, even today physicists have no agreed-upon answer to the question why time moves forwards but not backwards.

- *can define algorithms for exactly the same functions.*

Thus, it does not depend on the chosen programming language

- *whether a set is decidable,*
- *whether a function is computable.*

Proof. The proof is very complex. For every program of every language, we must provide an equivalent program in every other language.
However, this can be done (and has been done) for all languages. □

A related (stronger) theorem is that every programming language P allows defining for every programming language Q a program that executes Q-programs.
It is generally believed but impossible to prove that there is no programming language that can define more data structures or algorithms than the known ones.

2.2 Specification vs. Design vs. Implementation

Above we have seen sets and functions as well as data structures and algorithms. Moreover, we have already mentioned and programs.
The following table gives a overview of the relation between these concepts:

Specification	Design/Architecture	Implementation
sets functions	data structures algorithms	types functions

Software development consists of 3 steps:

1. The **specification** describes the intended behavior in terms of mathematical sets and functions.
   It does not prescribe in any way how these sets and functions are realized. The same specification can have multiple different correct realizations differing among others in size, maintainability, or efficiency.

2. The **architecture** makes concrete choices for the data structures and algorithms that realize the needed sets and functions.
   It usually defines many auxiliary data structures and algorithms that are not part of the specification.
   The architecture does not prescribe a programming language. It can be correctly realized in any programming language.

3. The **implementation** chooses a programming languages and then writes a **program** in it that realizes the architecture. The program includes concrete choices for the type and function definitions that realize the needed data structures and algorithms.
   It usually defines many auxiliary types and functions that are not part of the architecture.

Terminology 2.14. *Design* and *architecture* can usually be used synonymously.
The words *specification*, *design*, and *implementation* can refer to both the process and the result. For example, we can say that the result of implementation is one implementation.

It is critical to distinguish the three steps in software development:

- Specification changes are much more expensive than design changes. Changing the specification may completely change, which design is appropriate. Therefore, every single design decision must be revisited and checked for appropriateness.

- Design changes are much more expensive than implementation changes. Changing the design may completely change which components of the implementation are needed and how they interact.
  Therefore, every part of the program must potentially be revisited.
  In particular, whenever the design of component X is changed, we have to revisit every place of the program that uses X. This often introduces bugs.

Typically any specification change entails bigger design changes, and any design change entails bigger implementation changes. Moreover, specification changes require

- re-verification (i.e., checking that the implementation still correctly implements the specification)
- re-certification by regulatory agencies (if applicable to the specific software)
- changes to documentation, manuals, and tutorials, re-training of users, etc.
- distribution of software updates, which confuses and disrupts their workflows
- need for other software projects to adapt to the updated software

An ideal programmer proceeds in the order specification-design-implementation. However, it is often necessary to loop back: The design phase may reveal problems in the specification, and the implementation phase may reveal problems in the design. Therefore, we usually have to work on all 3 parts in parallel—but with a strong preference against changing specification and design.

Many self-taught or not-well-taught programmer do not understand the difference between the 3 steps or do not systematically apply it. There are many such programmers, who never studied CS or got a degree without taking a rigorous foundations course. Their programs are typically awful because:

- They begin programming without writing down the specification. Consequently, they do not realize that they have not actually understood the specification. This results in programs that do not meet the specification, which then leads to retroactive changes to the design. Over time the program becomes (sometimes called "spaghetti code") that is unmaintainable and cannot be understood by other programmers, often not even by the programmer herself.
- They begin programming without consciously choosing a design. Consequently, they end up with a random design that may or may not be appropriate for the task. Over time they change the design multiple times (without being aware that they are changing the design). Each change introduces new bugs and more mess.

Example 2.15 (Greatest Common Divider). The specification of the greatest common divider function gcd is as follows: Given natural numbers m and n, return a natural number g such that

- $g|m$ and $g|n$
- for every number h such that $h|m$ and $h|n$ we have that $h|g$

Before we design an algorithm, we should check whether gcd is indeed a function:

- Consistency: Does such a $g = \gcd(m, n)$ always exists?
- Uniqueness: Could there be more than one such g?

Using mathematics, we can prove that g indeed exists uniquely.

Now we design an algorithm. Let us assume that we have already designed data structures for the natural numbers with the usual operations. There are many reasonable algorithms, among them the one from Ex. 2.4. For the sake of example, we use a different one here:

fun $gcd(m : \mathbb{N}, n : \mathbb{N}) : \mathbb{N} =$
  if $n == 0$
    m
  else
    $\gcd(n, m \bmod n)$

This is a recursive algorithm: The instructions may recursive call the algorithm itself with new input.

Finally, we implement the algorithm. We choose SML as the programming language. First we implement the data structure for natural numbers and the function $mod : nat * nat \to nat$ that were assumed by the specification. Note that this requires some auxiliary functions that were not part of the algorithm:

```
datatype nat = zero | succ of nat

fun leq(m: nat, n: nat): bool = case (m,n) of
  (zero,     zero)     => true
| (zero,     succ(y))  => true
| (succ(x), zero)     => false
| (succ(x), succ(y))  => leq(x,y)
```

```
fun minus(m: nat, n: nat): nat = case (m,n) of
  (zero,     zero)     => zero
| (zero,     succ(y)) => zero (* error case, should not happen *)
| (succ(x), zero)     => succ(x)
| (succ(x), succ(y)) => minus(x,y)

fun mod(m:nat, n:nat):nat =
  if m = n then zero
  else if leq(m,n) then m
  else mod(minus(m,n), n)
```

Then we define

```
fun gcd(m:nat, n: nat): nat = if n = zero then m else gcd(n,mod(m,n))
```

2.3 Stateful Aspects

2.3.1 Immutable vs. Mutable Data Structures

Consider a data structure for the set $\mathbb{N}^*$ of lists of natural number and assume we have a variable $x : \mathbb{N}^*$.

Immutable Data Structures and Call-by-Value

We can always assign a new value to x as a whole. For example, after executing $x := [1, 3, 5]$, we have the following data stored in memory:

variable	type	value
x	$\mathbb{N}^*$	$[1, 3, 5]$

location	value
P	$[1, 3, 5]$

Here the left part shows the variables as seen by the programmer. The right part shows the objects as they are maintained in memory by the programming language. P is the name for the memory location holding the value of x. Importantly, the programmer is completely unaware of the organization of the data in memory and only sees the value of x.

In particular, x is just an abbreviation for the value $[1, 3, 5]$. If we pass x to a function f, there is no difference between saying $f(x)$ and $f([1, 3, 5])$. That is called **call-by-value**.

For example, if we execute the instruction $y = delete(x, 2)$, we obtain:

variable	type	value
x	$\mathbb{N}^*$	$[1, 3, 5]$
y	$\mathbb{N}^*$	$[1, 3]$

location	value
P	$[1, 3, 5]$
Q	$[1, 3]$

All old data is a before. For the new variable m, a new memory location Q has been allocated and filled with the result of the operation. This has the drawback that the entire list was duplicated, and we now use twice as much memory as before.

Immutable data structures and call-by-value are the usual way how functions work in mathematics. Such data structures are closely related their specification and makes writing, understanding, and analyzing algorithms very easy.

Mutable Data Structures and Call-by-Name

If our data structure is mutable, the value of a variable x is just a reference to the memory location where the value is stored.

For example, after executing $x := [1, 3, 5]$, we have the following data stored in memory:

variable	type	value
x	$\mathbb{N}^*$	P

location	value
P	$[1, 3, 5]$

The value of x is now the reference to the memory location. The programmer still cannot see P directly.[3]
But there are two carefully-designed ways how P can be accessed indirectly. Firstly, we can assign new values to each component of x. For example, after $x.1 := 4$, the memory looks like

variable	type	value
x	$\mathbb{N}^*$	P

location	value
P	$[1, 4, 5]$

The old value at location P is gone and has been replaced by the new value.
Secondly, when we pass x to a function f, we pass the reference to the value, not the value itself. This is called **call-by-name** or **call-by-reference**.
For example, after executing $delete(x, 2)$, we have

variable	type	value
x	$\mathbb{N}^*$	P

location	value
P	$[1, 4]$

No additional memory location has been allocated for the result, and copying took place. That makes the operation much more time- and memory-efficient. But from a mathematical perspective, this is very odd: The function call $delete(x, 2)$ *changed* the value x under the hood.

In many programming languages (in particular object-oriented ones), mutable data structures are called *classes*. Some of the related function will make use of mutability, some will not. This must be part of the specification of the function.

2.3.2 Environments and Side Effects

So far we have said that algorithms realize mathematical functions. That makes algorithms very close to the specification and makes writing, understanding, and analyzing them very easy. But it is not the whole picture in computer science—computer science needs a generalization:

Definition 2.16 (Stateful Functions)**.** Let E be the set of environments. An **effectful function** from A to B is a function $A \times E \to B \times E$.

Again this is a vague definition because the word "environment" is not defined. That is normal—there is no universally recognized definition for it. Intuitively, an object $e \in E$ represents the state of the environment. e contains all information that is visible from the outside of our algorithms and that can be acted on by the algorithm. These usually include the global variables, all kinds of input/output, and exceptions.
An effectful function f from A to B can do two things besides returning a result of type B:

- It can use the environment (because E occurs in its input type). Thus, calling f twice on the same $a \in A$ may return different result if the environment has changed in between.
  Formally, if $f(a, e_1) = (b_1, e_1')$ and $f(a, e_2) = (b_2, e_2')$ always implies $b_1 = b_2$, we say that f is **environment-independent**.
- It can change the environment (because E occurs in its output type). Thus, programmers must be careful when to call f and how often to call f because every call may have an effect that can be observed by the user.
  Formally, if $f(a, e) = (b, e')$ always implies $e = e'$, we say that f is **side-effect-free**.

If f is both environment-independent and side-effect-free, f is called **pure**. In that case, we always have $f(a, e) = (g(a), e)$ for some function $g : A \to B$, i.e., we can ignore environments entirely. Thus, pure functions are essentially the same as the usual mathematical functions.
An environment $e \in E$ is usually a big tuple containing among others

- the current values of all accessible variables
- console input/output:
  - the list of characters to be printed out to the user
  - the list of characters typed by the user that are available for reading
- file and peripheral network input/output: for every open file, network connection or similar
  - the list of data to be written to the connection

[3]Some programming languages allow explicitly creating and manipulating these references. The most notable example is *C* (where the references are called *pointers*). With very few caveats, that can be considered a design flaw in the programming language.

  - the list of data that is are available for reading
- information about exceptions
  - by depending on this aspect of the environment, effectful functions can handle exceptions
  - by effecting this aspect of the environment, effectful functions can raise exceptions
- additional components depending on the features of the respective programming language

Environment-dependency and side effects are important. Without input/output side effect, the user could never provide input for algorithms and could never find out what the output is. Moreover, computer could not be used to read sensor data or control peripheral devices.

But they also present major challenges to algorithm design. Because the precise definition of E depends on the details of the programming language, it is very difficult to precisely specify effectful functions. And without a precise specification, the programmer never knows whether an algorithm is designed and implemented correctly. Therefore, some programming languages such as Haskell try to systematically restrict them as much as possible.

Chapter 3

Design Goals

3.1 Correctness

3.1.1 General Definition

The most important goal of design is *correctness*:

Definition 3.1. We say that:

- A data structure D is correct for a set S if the objects of D correspond exactly to the elements of S.
- An algorithm A is correct for a function F if for every possible input x the result of running A on x has output $F(x)$.

Obviously, an incorrect algorithm is simply a bug.[1]
However, incorrect data structures are often used.

Example 3.2. The data structure *int* is not correct for the sets $\mathbb{N}$ or the $\mathbb{Z}$. In both cases, *int* has not enough objects. *int* even has objects that are not in $\mathbb{N}$ at all (namely negative numbers).
However, *int* is routinely used in practice as if it were a correct data structure for $\mathbb{N}$ and $\mathbb{Z}$. If *int* uses 32 bits, it only covers the numbers between -2^{31} and $2^{31}-1$. As long as all involved numbers are between -2^{31} and 2^{31}, this is no problem.
It is possible to define correct data structure for $\mathbb{N}$ and $\mathbb{Z}$. But that can be disadvantageous because

- operations on *int* are much faster,
- interfacing with other program components may be difficult if they use different data structures.

Example 3.3. There is no data structure that is correct for $\mathbb{R}$.
Therefore, the data structure *float* used in practice as if it were a correct data structure for $\mathbb{R}$. This always leads to rounding errors so that all results about are only approximate.
float is often also used as if it were a correct data structure for $\mathbb{Q}$. That is a bad habit because computations on *float* are only approximate even if the inputs are exact. For example, there is no guarantee that 1.0/2.0 returns 0.5 and not 0.49999999999.

Example 3.4. Object-oriented languages use class types. Because of the *null* pointer, a class A that implements a set S actually implements the set $S^?$: A value of type A can be *null* or an instance of A.
Therefore, many good programmers systematically avoid ever using *null*. Still, the use of *null* is wide-spread in practice.

[1] However, there are advanced areas of computer science that study approximation algorithms. For example, we may want to use a fast algorithm that is almost correct for a function for which no fast algorithm exists.

Example 3.5. Assume we have a correct data structure for A.

Then we can give a correct data structure for $\{x \in A | P(x)\}$ if $P \in \mathbb{B}^A$ is computable. However, because the set of computable functions is itself not decidable, programming languages usually do not allow defining data structures for $\{x \in A | P(x)\}$.

We cannot in general give a correct data structure for $\{F(x) : x \in A\}$ even if F is computable. Similarly, we cannot in general give a correct data structure for A/r even if $r \in \mathbb{B}^{A \times A}$ is computable.

Verification The process of making sure that an algorithm is correct is called *verification*. Verification is very difficult. In particular, the function that determines whether a data structure or algorithm is correct is itself not computable. Therefore, we have to prove the correctness of each data structure or algorithm individually.
Good programmers design algorithms that are close to the specification. That makes it easier to verify the design.
Verification often splits the correctness of an algorithm into two parts as follows:

Definition 3.6. An algorithm **terminates** for inputs I if the execution of its instructions takes only finite time. An algorithm is **partially correct** if it is correct for all inputs for which it terminates.

Theorem 3.7. *An algorithm is correct iff it is partially correct and terminates for all inputs.*

Partial correctness and termination are often proved separately. Sect. 3.1.2 and 3.1.3 describe the most important techniques.

3.1.2 Loop Invariant

Many algorithms use while-loops. Verifying the correctness of while-loops is notoriously difficult.
Therefore, many good programmers try to avoid while-loops altogether. Instead, they prefer operations on lists (like *map*, *fold*, and *foreach*) or recursive algorithms.

The central method for verifying the correctness of a while-loop is the *loop invariant*:

Definition 3.8 (Loop Invariant)**.** Consider a loop of the form $while\,(C(\vec{x}))\{code\}$. Here $\vec{x} = (x_1, \ldots, x_n)$ are the names that are in scope before entering the loop (i.e., excluding any names declared only in *code*).
A formula $F(\vec{x})$ is a loop invariant if F is preserved by the loop, i.e., if it holds before an iteration of the loop, it also afterwards. Specifically, for all $\vec{v}$, the following must hold

$$C(\vec{v}) \text{ and } F(\vec{v}) \qquad \text{implies} \qquad F(code(\vec{v}))$$

where $code(v) = (v'_1, \ldots, v'_n)$ contains the values of the x_i after executing $x_1 := v_1; \ldots; x_n := v_n; code$.

If we have a loop-invariant, we can use a loop invariant as follows:

Theorem 3.9. *Consider a while-loop* $while\,(C(\vec{x}))\{code\}$ *and a loop-invariant* $F(\vec{(x)})$.
Assume that $F(\vec{v})$ *holds where* v_i *is the value of* x_i *before executing the while-loop.*

Then $!C(\vec{(x)})\&\&F(\vec{x})$ *holds if the while-loop has been executed.*

Proof. After the while-loop $C(\vec{x})$ cannot hold—otherwise, the while-loop would continue. Because $F(\vec{x})$ held before executing the loop and is preserved by *code*, it also holds after executing the loop. □

Note that Thm. 3.9 talks about *if* and not *when* the while-loop has been executed. That is because it is not guaranteed that a while-loop after terminates. We still have to prove that separately.

Example 3.10 (Euclid's Algorithm). Consider the algorithm from Ex. 2.4. We proceed statement-by-statement.

The first two statements are easy to handle: Their effect is that $x == m$ and $y == n$.

But now we reach a while-loop. We have $\vec{x} = (m, n, x, y)$ and $C(m, n, x, y) = x \neq y$. A loop invariant is given by $F(m, n, x, y) = \gcd(m, n) == \gcd(x, y)$. The intuition of this loop-invariant is that we only apply operations to x and y that do not change their gcd.

To work with the while-loop, we prove that F is a loop invariant:

- Before execution of the loop, we have $x == m$ and $y == n$. Thus, immediately $gcd(m, n) == gcd(x, y)$.
- Let us assume that $C(m, n, x, y)$ holds, i.e., $x \neq y$ (i).
  Moreover, let us assume that $F(m, n, x, y)$ holds, i.e., $\gcd(m, n) == \gcd(x, y)$ (ii).
  Let $code(m, n, x, y) = (m', n', x', y')$.
  We have to prove $F(m', n', x', y')$, i.e., $\gcd(m, n) = \gcd(x', y')$.
  To do that, we have to distinguish two cases according to the if-statement:
  - $x < y$: Then $(m', n', x', y') = (m, n, x, y - x)$. Thus we have to prove that $\gcd(m, n) = \gcd(x, y - x)$. Because of (ii), it is sufficient to prove $\gcd(x, y) = \gcd(y - x)$. That follows from the mathematical properties of gcd.
  - $y < x$: Then $(m', n', x', y') = (m, n, x - y, y)$. We have to prove that $\gcd(m, n) = \gcd(x - y, x)$. That follows in the same way as in the first case.
  - We do not need a case for $x == y$ because that is excluded by the condition of the loop.

Now we can continue. The next statement is **return** x. Using Thm. 3.9, we obtain that $!C(m, n, x, y) \&\& F(m, n, x, y)$ holds, i.e., $!x \neq y \&\& \gcd(m, n) == \gcd(x, y)$. That yields $x == y$ and therefore $\gcd(m, n) == \gcd(x, x) == x$. Thus, the returned value is indeed $\gcd(m, n)$.
To complete the correctness proof, we still have to show that the while-loop terminates, which we do in Ex. 3.13

3.1.3 Termination Orderings

Verifying the termination of an algorithm is also very hard. The halting function is the function that takes as input an algorithm A and an object I and returns as output the following boolean: *true* if A terminates with input I and *false* otherwise. The halting function is not computable.

Thus, even if do not care what our algorithm actually does and only want to know if it terminates, all we can do is prove it manually for each input.

Termination is trivial for assignment, if-statement, and the return-statement. Only while-loops and recursion are tricky. The most important technique to prove termination is to use a termination ordering.

While-Loops

Definition 3.11 (Termination Ordering)**.** Consider a while-loop of the form $while\,(C(\vec{x}))\{code\}$.
A termination ordering is a function $T(\vec{x}) \in \mathbb{N}$ such that for all $\vec{v}$ we have that $C(\vec{v})$ implies $T(\vec{v}) > T(code(\vec{v}))$.

Theorem 3.12 (Termination Ordering)**.** *Consider a while-loop* $while\,(C(\vec{x}))\{code\}$ *and a termination ordering* $T(\vec{x})$.
Then the while-loop terminates for all initial values $\vec{v}$ *of* $\vec{x}$.

Proof. We define a sequence $\vec{v}^0, \vec{v}^1, \ldots$ such that $\vec{v}^i$ contains the values of $\vec{x}$ after executing *code* i times:

$$\vec{v}^0 = \vec{v}$$

$$\vec{v}^{i+1} = code(\vec{v}^i) \qquad \text{for } i > 0$$

We use an indirect proof: We assume the while-loop does not terminate and show a contradiction.
If the loop does not terminate, the condition must always be true, i.e., $C(\vec{v}^i)$ for all $i \in \mathbb{N}$.

Then the termination ordering yields $T(\vec{v}^i) > T(\vec{v}^{i+1})$ for all $i \in \mathbb{N}$.
That yields an infinite sequence $T(\vec{v}^0) > T(\vec{v}^1) > \ldots$ of natural numbers.
But such a sequence cannot exist, which yields the needed contradiction. □

Example 3.13 (Euclid's Algorithm). We prove that the algorithm from Ex. 2.4 terminates for all inputs. Only the while-loop presents a problem.
A termination ordering for the while-loop is given by $T(m, n, x, y) = x + y$. The intuition of this termination ordering is that the loop makes x or y smaller. Therefore, it makes their sum smaller.

We show that T is indeed a termination ordering.
As when proving the loop-invariant, we put $(m', n', x', y') = code(m, n, x, y)$.
We have to show that $T(m, n, x, y) > T(m', n', x', y')$, i.e., $x + y > x' + y'$.
We again distinguish two cases according to the if-statement:

- $x < y$ and thus $(m', n', x', y') = (m, n, x, y - x)$: We have to show $x + y > x + y - x$.
- $x > y$ and thus $(m', n', x', y') = (m, n, x - y, y)$: We have to show $x + y > x - y + y$.

Both cases are trivially true for all $x, y \in \mathbb{N} \setminus \{0\}$.
But what happens if $x == 0$ or $y == 0$? Indeed, the proof of the termination ordering property does not go through.
Inspecting the algorithm again, we realize that we have found a bug: If exactly one of the two inputs is 0, the algorithm never terminates.
We can fix the algorithm in two ways:

- We change the specification to match the behavior of the algorithm. That means to change the input data structure such that $m, n \in \mathbb{N} \setminus \{0\}$.
- We change the algorithm to match the specification. We can do that by adding the lines

  if $(x == 0)$ {**return** y}
  if $(y == 0)$ {**return** x}

  Now the loop can be analyzed with the assumption that $x \neq 0$ and $y \neq 0$.

Recursion

Termination orderings for recursion work in essentially the same way. But the precise definition is a little bit trickier.

Definition 3.14 (Termination Ordering for Recursion)**.** Consider a recursive function $f(\vec{x})$.
A termination ordering for f is a function $T(\vec{x}) \in \mathbb{N}$ such that: whenever f is called on arguments $\vec{v}$ and recursively calls itself with arguments $\vec{v'}$, then $T(\vec{v}) > T(\vec{v'})$.

Definition 3.15 (Relative Termination)**.** Consider a recursive function $f(\vec{x})$.
We say that f terminates relatively if the following holds: f terminates for all arguments under the assumption that all recursive calls return.

Theorem 3.16 (Termination Ordering for Recursion)**.** *Consider a recursive function $f(\vec{x})$ with a termination ordering T for it.*
If f terminates relatively, then it terminates for all arguments.

Proof. This is proved in the same way as for while-loops. □

Example 3.17 (Recursive Euclidean Algorithm). Consider the recursive algorithm from Ex. 2.15.
It is easy to see that the arguments never get bigger during the recursion. So we might try $T(m, n) = m + n$ as a termination ordering. But that does not work because if $m < n$, the recursive call is to $gcd(n, m)$, which just flips the arguments. In that case, $T(m, n) = m + n$ does not become strictly smaller.
It becomes easier to show termination if we expand the recursive call once. That yields the equivalent function:

```
fun gcd(m : ℕ, n : ℕ) : ℕ =
    if n == 0
        m
    else
        if m mod n == 0
            n
        else
            gcd(m mod n, n mod(m mod n))
```

Relative termination trivial either way: Under the assumption that the recursive call returns, the function consists only of if-statements and terminates immediately.
And for the expanded function, $T(m, n) = m+n$ is a termination ordering. We have to prove $m+n > (m \bmod n)+(n \bmod(m \bmod n))$, which is easy to see.

3.2 Efficiency

3.2.1 Exact Complexity

While termination describes whether an algorithm A terminates at all, the complexity describes how long it takes to terminate. The complexity of A is a function $C : \mathbb{N} \to \mathbb{N}$ such that $C(n)$ is the number of steps needed until A terminates for input of size n.

An algorithm is efficient if its complexity is low and vice versa.

The definition of the number of steps and the sizes of inputs depend on the programming language and the physical machine that is used. Below we give a typical definition as an example.
At Jacobs University, decidability and computability are discussed in detail in a special course in the $2nd$ year.

Example 3.18 (Computing Exact Complexity). For a typical programming language implemented on a digital machine, the following definition is roughly right:

For the execution of a statement:

- $\text{Steps}(C; D) = \text{Steps}(C) + \text{Steps}(D)$
- $\text{Steps}(x := E) = \text{Steps}(E) + 1$
  - $\text{Steps}(E)$ steps to evaluate the expression E
  - 1 step to make the assignment
- $\text{Steps}(\textbf{return } E) = \text{Steps}(E) + 1$
  - $\text{Steps}(E)$ steps to evaluate the expression E
  - 1 step to return
- $\text{Steps}(\textbf{if } (C) \ \{T\} \textbf{ else } \{E\}) = \text{Steps}(C) + 1 + \begin{cases} \text{Steps}(T) & \text{if } C == \textit{true} \\ \text{Steps}(E) & \text{if } C == \textit{false} \end{cases}$
  - $\text{Steps}(C)$ steps to evaluate the condition
  - 1 step to branch
  - $\text{Steps}(T)$ or $\text{Steps}(E)$ steps depending on the branch
- $\text{Steps}(\textbf{while } \ C \ \{B\}) = (n + 1) * \text{Steps}(C) + n * \text{Steps}(B)$ where n is the number of times that the loop is repeated
  - $\text{Steps}(C)$ steps to evaluate the condition $n + 1$ times
  - 1 step to branch after each evaluation of the condition
  - $\text{Steps}(B)$ steps to execute the body

For the evaluation of an expression:

- Retrieving a variable: Steps$x = 1$
- Applying built-in operators O such as + or &&: Steps$O(E_1, \ldots, E_n = \text{Steps}E_1 + \ldots + \text{Steps}E_n + 1$
  - Steps(E_i) steps to evaluate the arguments
  - 1 step to apply the operator
- Calling a function: Steps$(f(E_1, \ldots, E_n)) = \text{Steps}(E_1) + \ldots + \text{Steps}(E_n) + 1 + n$
  - Steps(E_i) steps to evaluate the arguments
  - 1 step to create jump into the definition of f
  - 1 step each to pass the arguments to f

The size of an object depends on the data structure:

- For *int*, *float*, *char*, and $\mathbb{B}$, the size is 1.
- For *string*, the size is the length of the string.
- For lists, the size is the sum of the sizes of the elements plus 1 more for each element. The "1 more" is needed because each element needs a pointer to the next element of the list.

Problems with Exact Complexity In actuality however, a number of subtleties about the implementation of the programming language, its compiler, and the physical machine can affect the run-time of a program. For example:

- We usually assume that all arithmetic operations take 1 step. But actually, that only applies to arithmetic operations on the type *int* of 32 or 64-bit integers.
  - Any arithmetic operation that can handle arbitrarily large numbers takes longer for larger numbers. Most such arithmetic operations have complexity closely related to the number of digits needed to represent the arguments. That number is logarithmic in the size of the arguments.
  - Multiplication and related operations usually take longer than addition and related operations. Similarly, exponentiation usually takes longer than multiplication.
  - Any operation not built into the hardware must be implemented using software, which makes it take longer. Operations on larger numbers may take longer even if they are of type *int*.
- Floating point operations may take more than 1 step.
- The programming language may provide built-in operations that are actually just abbreviations for non-trivial functions. For example, concatenation of strings usually require copying one or both of the strings, which takes at least 1 step for each character. In that case, concatenating longer strings takes longer.
- The programming language's compiler may perform arbitrary optimizations in order to make execution faster. For example, we may have Steps(**if** (*false*) $\{E\}) = 0$ because the compiler removes the statement entirely. On the other hand, optimization may occasionally use a bad trade-off and make execution slower.
- A smart compiler may generate code that is optimize for multi-core machines, such that, e.g., 2 steps are executed in 1 step.
- Calling a function may take much more than 1 step to jump to the function. Usually, it requires memory allocation, which can be a complex operation.
- For advanced operations, like instantiating a class, it is essentially unpredictable how many steps are required.
- From a complexity perspective, IO-operations (printing, networking, file access, etc.) take as many steps as the size of the sent data. But they take much more time than anything else.

The dependency of exact complexity on programming language, implementation, and physical machine is awkward because it precludes analyzing an algorithm independent of its implementation. Therefore, it is common to work with asymptotic complexity instead.

The ideas is these dependencies are usually harmless in the sense that they can be "rounded away". For example, it does not matter much whether Steps$(x := E) = \text{Steps}(E) + 1$ or Steps$(x := E) = \text{Steps}(E) + 2$. It just means that every program takes a little longer. It would matter more if Steps$(x := E) = 2 * \text{Steps}(E) + 1$, which is unlikely.

We introduce the formal definitions in Sect. 3.2.2 and apply them in Sect. 3.2.3.

3.2.2 Asymptotic Notation

The field of complexity theory usually works with with BachmannLandau notations.[2] The basic idea is to focus on the rough shape of the function $C(n)$ instead of its details. For example, $C(n) = an + b$ is linear, and $C(n) = 2^{an+b}$ is exponential. The distinction linear vs. exponential is often much more important than the distinction $an + b$ vs. $a'n + b'$.

Therefore, we define classes of functions like linear, exponential, etc.:

Definition 3.19 (O-Notation)**.** Let $\mathbb{R}^+$ be the set of positive-or-zero real numbers.

We define a relation on functions $f, g : \mathbb{N} \to \mathbb{R}^+$ by

$$f \lessdot g \quad \text{iff} \quad \exists N \in \mathbb{N}. \exists k > 0. \forall n > N. f(n) \leq k \cdot g(n)$$

If $f \lessdot g$, we say that f is **asymptotically smaller** than g.

We write $f \eqdot g$ if $f \lessdot g$ and $g \lessdot f$.

Moreover, for a function $g : \mathbb{N} \to \mathbb{R}^+$, we define the following sets of functions

$$O(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid f \lessdot g\}$$

$$\Omega(g) = \{h : \mathbb{N} \to \mathbb{R}^+ \mid g \lessdot h\}$$

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid f \eqdot g\} = O(g) \cap \Omega(g)$$

Intuitively, $f \lessdot g$ means that f is essentially smaller than g. More precisely, f is smaller than g *for sufficiently large arguments* and *up to a constant factor*. The other definitions are straightforward: $O(g)$ is the set of everything smaller than g, $\Omega(g)$ is the set of everything larger than g, and $\Theta(g)$ is the set of everything essentially as great as g (i.e., both smaller and larger).

Remark 3.20 (A Slightly Simpler Definition). The following statement is not true in general. However, it is easier to remember and true for all functions that come up when analyzing algorithms: $f \lessdot g$ iff $\exists a > 0, b > 0. \forall n. f(n) \leq a \cdot g(n) + b$.

We can verbalize that condition as "f is smaller than g except for a constant factor and a constant summand". Those are the two aspects of run time that we can typically make up for by building faster machines.

Example 3.21. Now we can easily define some important classes of functions grouped by their rough shape:

- $\Theta(1)$ is the set of $(*)$ constant functions
- $\Theta(n)$ is the set of $(*)$ linear functions
- $\Theta(n^2)$ is the set of $(*)$ quadratic functions
- and so on

Technically, we should always insert "asymptotically" at $(*)$. For example, $\Theta(n)$ contains not only the linear functions but also all functions whose shape is similar to linear. But that word is often omitted for brevity.

If we use O instead of Θ, we obtain the sets of *at most* constant/linear/quadratic/etc. functions. For example, $O(n)$ includes the constant functions whereas $\Theta(n)$ does not.

Similarly, if we use Ω instead of Θ, we obtain the sets of *at least* constant/linear/quadratic/etc. functions. For example, $\Omega(n)$ includes the quadratic functions whereas $\Theta(n)$ does not.

Of particular importance in complexity analysis is the set of polynomial functions: . It includes all all functions whose shape is similar to a polynomial.

The following list introduces a few more classes and arranges them by increasing size:

[2] In the definition below, only O, Ω, and Θ are the standard BachmannLandau notations. The symbols $\lessdot$ and $\eqdot$ are specific to these lecture notes.

$O(1)$	constant
$O(\log_c \log_c n)$	doubly logarithmic
$O(\log_c n)$	logarithmic
$O(n)$	linear
$O(n \log_c n)$	quasi-linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$\vdots$	$\vdots$
$Poly = \bigcup_{k \in \mathbb{N}} O(n^k)$	polynomial
$Exp = \bigcup_{f \in Poly} O(c^{p(n)})$	exponential
$\bigcup_{f \in Exp} O(c^{f(n)})$	doubly exponential

Here $c > 1$ is arbitrary—all choices yield the same classes of functions.

We also say sub-X for strictly lower and super-X for strictly greater complexity than X. For a $\log_c n$ is sub-linear, and n^2 is super-linear.

The following theorem collects the basic properties of these concepts:

Theorem 3.22 (O-Notation)**.** *We have the following properties for all f, g, h, f', g':*

- $\lesssim$ *is*
  - *reflexive: $f \lesssim f$*
  - *transitive: if $f \lesssim g$ and $g \lesssim h$, then $f \lesssim h$*

  Thus, it is a preorder.
- *If $f \lesssim f'$ and $g \lesssim g'$, then $\lesssim$ is preserved by*
  - *addition: $f + g \lesssim f' + g'$*
  - *multiplication: $f \cdot g \lesssim f' \cdot g'$*
- $\eqsim$ *is*
  - *reflexive: $f \eqsim f$*
  - *transitive: if $f \eqsim g$ and $g \eqsim h$, then $f \eqsim h$*
  - *symmetric: if $f \eqsim g$, then $f \eqsim g$*

  Thus, it is an equivalence relation.
- *The following are equivalent:*
  - $f \lesssim g$
  - $O(f) \subseteq O(g)$
  - $\Omega(f) \supseteq \Omega(g)$
  - $f \in O(g)$
  - $g \in \Omega(f)$

  All statements express that f is essentially smaller than g.
- *The following are equivalent:*
  - $f \in \Theta(g)$
  - $g \in \Theta(f)$
  - $\Theta(f) = \Theta(g)$

  All statements express that f is essentially as great as g.

Proof. Exercise. □

Notation 3.23. The community has gotten used to using $O(f(n))$ as if it were a function. If $f(n) - g(n) \in O(r(n))$, it is common to write $f(n) = g(n) + O(r(n))$. The intuition is that f arises by adding some function in $O(r(n))$ to g. This is usually when r is smaller than g, i.e., r is a rest that can be discarded.

Similarly, people often write $f = O(r(n))$ instead of $f \in O(r(n))$ to express that f is equal to some function in $O(r(n))$.

These notations are not technically correct and should generally be avoided. But they are often convenient.

Example 3.24. Using Not. 3.23, we can write $2^n + 5n^2 + 3 = 2^n + O(n^2)$. This expresses that 2^n is the dominating term and the polynomial rest can be rounded away.
Or we can write $6n^3 + 5n^2 + \log n = O(n^3)$.

Remark 3.25 (Other Notations). There are a few more notations like O, Ω, and Θ. They include o and ω. They are less important and are omitted here to avoid confusing the reader.

3.2.3 Asymptotic Analysis

Equipped with asymptotic notations, we can now compute the run time of algorithms in a way that is mostly independent of the implementation and the machine.

Example 3.26. . Consider the algorithm Ex. 2.5. Let $C(n)$ be the number of steps it takes with input n.
Because we are only interested in the complexity class of C, this is quite simple:

1. The while-loop must be repeated n-times. So the algorithm is at least linear.
2. Each iteration of the while-loop requires one comparison, one multiplication, and two assignments. These operations take a constant number c of steps.[3]
   So the entire loop takes $c \cdot n$ steps. The value of c does not matter because we can ignore all constant factors. Thus, the entire loop takes $\Theta(n)$ steps.
3. The assignments in the first two lines and the return statement take constant time each. Because $C(n)$ is at least linear, we can ignore them entirely.
4. Thus, we obtain $C(n) \in \Theta(n)$ as the complexity class of the algorithm.

Note how all the subtleties described in Sect. **??** are rounded away by looking at Θ-classes.

There are some subtle ambiguities when analyzing complexity:

- In $C(n)$, we usually say that n is the size of the input. But it is not always clear what the size is:
  - Is n the size of a number $n \in N$? Or is it $\log n$, which is the number of bits needed to represent n?
  - If the input is a list, is n just the length of the list? Or does it matter how big the elements of the list are?
  - If there are multiple inputs, do we simply add their sizes?
- Sometimes the run time depends on the exact value, not just on its size. For example, Ex. 2.4 happens to terminate immediately if $m = n$, no matter what the size is.
  Thus, we have to distinguish between:
  - worst-case complexity: This is the maximal possible number of steps. If there is no information, this is usually what the author means.
  - average-case complexity: This may be more useful in practice. However, it is more difficult because we need a probabilistic analysis.
  - best-case complexity: This is rarely useful but occasionally helps put a lower bound on the complexity.

There are no universal answers to these questions. Instead, we have to consider the context to understand what the author means.

Example 3.27. (Euclid's Algorithm] Consider the algorithm from Ex. 2.4. Let $n = \max(a, b)$ and let $C(n)$ be the worst-case number of steps the algorithm takes for input a, b (i.e., we use the maximum value of the inputs as the argument of the complexity function).
It is not that easy to see what the worst case actually is. But we can immediately see that the loop is repeated at most n times. Each iteration requires one comparison, one subtraction, and one assignment, which we can sum up to a constant factor.[4] Thus, the critical question is how often the loop can be repeated.
We can answer that question by going backwards. Because x and y are constantly decreased but stay positive, the worst case must arise if they are both decreased all the way down to 1. Then computing through the loop backwards, we obtain $1, 1, 2, 3, 5, 8, 13$ as the previous values, i.e., the Fibonacci numbers.
Indeed, the worst-case of the Euclidean algorithm arises if m and n are consecutive Fibonacci numbers. By applying some general math (see Sect. 4.2), we obtain that $Fib(k) \in \Theta(2^k)$. Thus, if n is a Fibonacci number, the number of repetitions of the loop is in $\Theta(\log n)$.
Thus, $C(n) \in \Theta(\log n)$.

The following is an (highly simplified variant of an) example from the author's research:

Example 3.28 (String Processing). Consider the following function in the Scala programming language

```
def processString(s: String) {
  var rest = s
  while (s != "") {
    if (rest.startsWith("foo")) {
      // does not matter, assume this takes constant time
    } else {
      // also does not matter
    }
    rest = s.substring(1)
  }
}
```

Here startsWith and substring are methods on strings from the Java library (which Scala can call with only constant-time overhead).

Let $C(n)$ be the run time of this function where n is the length of the input string. What is the complexity class of $C(n)$?

You can the program yourself on increasingly large input to find out. In the author's case, the surprising effect was noticeable when $n > 10^7$, e.g., when reading 10 MB text file.

3.2.4 Discussion

Asymptotic Analysis

Asymptotic analysis is the dominant form of assessing the complexity of algorithms. It has the huge advantages that it

- is mostly largely independent of the implementation and the physical machine,
- abstract away from minor details that do not significantly affect the quality of the algorithms.

But it has some disadvantages. Most importantly, the terms that it ignores can be huge. For example, $n+2^{(2^{10000})} \in O(n)$ is linear. But the constant term is so huge that an algorithm with that complexity will never terminate in practice.

More formally, $f \lesssim g$ only means that f is smaller than g for *sufficiently large* input. Thus, $f \lesssim g$ does not mean that f is better than g. It only means that f is better than g if we need the results for sufficiently large inputs.

Judging Complexity

Θ-classes for complexity are usually a very reliable indicator of the performance of an algorithm. If two algorithms were designed naturally without extreme focus on complexity, we can usually assume that:

- For small inputs, they are both fast, and it does not matter which one we use.
- For large inputs, the one in the smaller complexity class will heavily outperform the other.

Note that large inputs are usually not encountered by the programmer: the programmer often only tests his programs with small test cases and examples. Instead, large input is encountered by users. Therefore, complexity analysis is an important tool for the programmer to judge algorithms. Most of the time this boils down to relatively simple rules of thumb:

- Avoid doing something linearly if you do it logarithmically or in constant time.
- Avoid doing something quadratically if you do it quasi-linearly or linearly.
- Avoid doing something exponentially if you can do it polynomially.

The distinction between exponential and polynomial has received particularly much attention in complexity theory. For example, in cryptography, as a rule of thumb, polynomial is considered easy in the sense that anything that takes only polynomial amount of time to hack is considered insecure. Exponential on the other hand is considered hard and therefore secure. For example, the time needed to break a password through brute force is exponential in the length of the password. So increasing the length and variety of characters from time to time is enough to stay ahead of brute force attacks.

Algorithm Complexity vs. Specification Complexity

Note that we have only considered the complexity of *algorithms* here.

We can also define the complexity of a specification: Given a mathematical function f, its complexity is that of the most efficient correct algorithm A for it. In this context, f is usually called the problem and A a solution.

It is generally much harder to analyze the complexity of a problem than that of an algorithm. It is easy to establish an upper bound for the complexity of a problem: Every algorithm for f defines an upper bound for the complexity of f. But to give a lower bound, we have to prove that there is no better algorithm for f. Proving the absence of something is generally quite difficult.

An example is the $P \neq NP$ conjecture, which is the most famous open question in computer science. To prove it, one has to show that there is no polynomial algorithm for any one of a certain large class of problems.

Algorithm Complexity vs. Implementation Complexity

The complexity of an implementation is its actual run-time. It is usually assumed that this corresponds to the complexity of an algorithm.

But occasionally, the subtleties discussed in see Ex. 3.18 have to be considered because they do not get rounded away. This subtleties can usually not make the implementation less complex than the algorithm, but they may it more complex. Most importantly, when analyzing the complexity of algorithms, we often assume that arithmetic operations can be performed in $O(1)$. In practice, that is only true for numbers within the limits of the type *int*. If we implement the data structures for numbers correctly, the complexity of the implementation will increase.

More generally, when analyzing algorithm complexity, we must make assumptions about the complexity of the primitive operations used in the algorithm. Then the complexity of the implementation is equal to complexity of the algorithm if the implementation of the primitive operations satisfies these assumptions.

Example 3.29 (Euclidean Algorithm). The implementation in Ex. 2.15 uses a very inefficient implementation for the data structure $\mathbb{N}$. It does not satisfy the assumption that arithmetic operations are done in $O(1)$. In fact, already the function implementing $\leq$ is in $\Theta(n)$. Consequently, the complexity of this particular implementation of gcd is higher than $\Theta(n)$.

But there are efficient correct implementations of $\mathbb{N}$, which we could use instead. For example, if we use base-2 representation, we can implement natural numbers as lists of bits. Because the number of bits of n is $\Theta(\log_2 n)$, most arithmetic operations end up being $O(p(\log_2 n))$ for a polynomial p. For example, addition and subtraction take time linear in the number of bits. Multiplication and related operations such as mod take a bit more than linear. That is more than $O(1)$ but still small enough to often be inessential.

Then the implementation of gcd, which uses $\Theta(\log_2 n)$ steps and a mod at every step, has a complexity somewhat bigger than $O((\log_2 n)^2)$. The details depend on how we implement mod.

3.3 Simplicity

An important and often under-estimated design goal is simplicity.

An algorithm should be elegant in the sense that it is very close to its mathematical specification. That makes it easy to understand, verify, document, and maintain.

Often simplicity is much more important than efficiency. The enemy of simplicity is optimization: Optimization increases efficiency usually at the cost of simplicity.

In practice, programmers must balance these two conflicting goals carefully.

Example 3.30 (Building a List). A frequent problem is to read a bunch of values and store them in a list. This usually requires appending every value to the end of the list as in:

```
data = []
while moreData
    d = getData
    data = append(data, d)
return data
```

But appending to *data* may take linear time in the length of the list. This is because *data* points to the beginning of the list, and the append operation must traverse the entire list to reach the end. Thus, traversal takes 1 step for the first element that is appended, 2 for the second, and so on. The total time for appending n elements in a row is $1 + 2 + \ldots + n = n(n+1)/2 \in \Theta(n^2)$. Thus, we implement a linear problem with a quadratic algorithm.

A common solution is the following:

```
data = []
while moreData
    d = getData
    data = prepend(d, data)
return reverse(data)
```

This *prepends* all elements to the list. Because no traversal is required, each prepend operation takes constant time. So the whole loop takes $O(n)$ steps.

But we build the list in the wrong order. Therefore, we revert it before returning it. Reversal must traverse and copy the entire list once, which takes linear time again.

Thus, the second algorithm runs in $O(n)$ overall.

But it requires an additional function call, i.e., it is less simple. In a very large program, it is possible that the calls to *prepend* and *reverse* occur in two different program locations that are far away from each other. A programmer who joins the project may not realize that these two calls are related and may introduce a bug.

It is non-obvious which algorithm should be preferred. The decision has to be made on a case-by-case basis keeping all goals in mind. For example, if the data is ultimately read from or written to a hard drive, that will be linear. But it will be much slower than building the list in memory, no matter if the list is built in linear or quadratic time.

3.4 Advanced Goals

There are a number of additional properties that algorithms should have. These can be formally part of the specification, in which case they are subsumed by the correctness properties. But often they are deliberately or accidentally ignored when writing the specification.

Reliability An algorithm is **reliable** if it minimizes the damage that can be caused by external factors. For example, power outages, network failures, user error, available memory and CPU, communication with peripherals (printers, hard drive, etc.) can all introduce problems even if all data structures and algorithms are correct.

Safety A system is safe if it cannot cause any harm to property or humans. For example, an algorithm governing a self-driving car must make sure not to hit a human.

Often safety involves interpreting signals received from and sending signals to external devices that operate in the real world, e.g., the cameras and the engine of the car. This introduces additional uncertainty (not to mention the other cars and pedestrians) that can be difficult to anticipate in the specification.

Security A system is secure if it cannot be maliciously influenced from the outside. This includes all defenses against hacking.

Security is often not part of the specification. In fact, attacking a system often requires intentionally violating the specification in order to call algorithms with input that the programmer did not anticipate.

Secure algorithms must catch all such invalid data.

Privacy Privacy is the requirement that only the output of an algorithm is visible to the user. Perfect privacy is impossible to realize because all computation leaks some information other than the output: This reaches from runtime and resource use to obscure effects like the development of heat due to CPU activity.

More critically, badly designed systems may expose intermediate data that occurred during execution but is not specified to be part of the output. For example, when choosing a password, the output should only the cryptographic hash of the password, not the password itself.

Additionally, a system may behave according to its specification, but the user may be unaware of it. For example, a user may not be aware that her word document stored its previous revision, thus accidentally exposing an early draft.

Chapter 4

Arithmetic Examples

4.1 Exponentiation

4.1.1 Specification

The function $exp(x \in \mathbb{Z}, n \in \mathbb{N}) \in \mathbb{N}$ returns the n-the power of x defined by

$$x^0 = 1$$

$$x^n = x \cdot x^{n-1} \qquad \text{if } n > 0$$

By induction on n, we show this indeed specifies a unique function.

4.1.2 Naive Algorithm

It is straightforward to give an algorithm for exponentiation. For example,

```
fun power(x : ℤ, n : ℕ) : ℕ =
    if x == 0
        1
    else
        x · power(x, n − 1)
```

Correctness The correctness of this algorithm is immediate because it follows the specification literally.

Complexity Assuming that all multiplications take $O(1)$ no matter how big x is, the complexity of this algorithm is $\Theta(n)$ because we need n multiplications and recursive calls.

4.1.3 Square-and-Multiply Algorithm

It is easy to think that $\Theta(n)$ is also the complexity of the specification, i.e., that there is no sub-linear algorithm for it. But that is not true.
Consider the square-and-multiply algorithm:

```
fun sqmult(x : ℤ, n : ℕ) : ℕ =
    if n == 0
        1
    else
        r := sqmult(x, n div 2)
        sq := r · r
        if (n mod 2 == 0) {sq} else {x · sq}
```

Correctness To prove the correctness of this algorithm, we note that

$$x^{2i+0} = (x^i)^2$$

$$x^{2i+1} = x \cdot (x^i)^2$$

Moreover, we know that $n = 2(n \operatorname{div} 2) + (n \bmod 2)$. Partial correctness of *sqmult* follows immediately.
To prove termination, we observe that $T(x, n) = n$ is a termination ordering: $n \operatorname{div} 2$ always decreases (because $n \neq 0$) and remains positive.

Complexity Computing the run time of a recursive function often leads to a recurrence relation: The function occurs on both sides with different arguments. In this case, we get:

$$C(n) = C(n \operatorname{div} 2) + c$$

where $c \in O(1)$ is the constant-time effort needed in each iteration. We systematically expand this further

$$C(n) = C(n \operatorname{div} 2) + c = C(n \operatorname{div} 2 \operatorname{div} 2) + 2 \cdot c = \ldots = C(n \overbrace{\operatorname{div} 2 \ldots \operatorname{div} 2}^{k+1 \text{ times}}) + (k+1) \cdot c$$

Now let $n = (b_k \ \ldots \ b_0)_2$ be the binary representation of the exponent. We know that $k = \lfloor \log_2 n \rfloor$ and $n \overbrace{\operatorname{div} 2 \ldots \operatorname{div} 2}^{k+1 \text{ times}} = 0$. Moreover, we know from the base case that $C(0) = 1$.
Substituting these above yield

$$C(n) \in O(1) + \Theta(\log_2 n) \cdot O(1) = \Theta(\log_2 n)$$

Thus, we can compute power in logarithmic time.

4.2 Fibonacci Numbers

4.2.1 Specification

The Fibonacci numbers $Fib(n \in \mathbb{N}) \in \mathbb{N}$ are defined by

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2) \qquad \text{if } n > 1$$

By induction on n, we prove that this indeed specifies a unique function.
Moreover, we can prove the non-obvious result that

$$fib(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \qquad \text{for } \varphi = \frac{1+\sqrt{5}}{2}$$

(φ is also called the golden ratio.) That can be further simplified to

$$fib(n) = round\left(\frac{\varphi^n}{\sqrt{5}}\right)$$

where we round to the nearest integer.

4.2.2 Naive Algorithm

It is straightforward to give an algorithm for computing Fibonacci numbers. For example:

```
fun fib(n : ℕ) : ℕ =
    if x ≤ 1
        x
    else
        fib(n − 1) + fib(n − 2)
```

Correctness The correctness of this algorithm is immediate because it follows the specification literally.

Complexity We obtain the recurrence relation $C(n) = C(n-1)+C(n-2)+c$ where $c \in O(1)$ is the constant-time effort of the recursion. That is the same recurrence as for the definition of the Fibonacci numbers themselves, thus $C(n) \in O(fib(n)) = Exp$.
This naive approach is exponential because every functions spawns 2 further calls. Each time n is reduced only by 1 or 2, so we have to double the number of calls about n times to $\Theta(2^n)$ calls.

4.2.3 Linear Algorithm

It is straightforward to improve on the naive algorithm turning an exponential into a liner solution. For example:

```
fun fib(n : ℕ) : ℕ =
    if x ≤ 1
        x
    else
        prev := 0
        current := 1
        i = 1
        while i < n
            next := current + prev
            prev := current
            current := next
            i := i + 1
        return current
```

Correctness As a loop invariant, we use

$$F(prev, current, i) = prev == fib(i-1) \;\&\&\; current == fib(i)$$

which is straightforward to verify. After the loop, we have $i == n$ and thus $current = fib(n)$, which yields partial correctness.
As a termination ordering, we use $T(prev, current, i) = n - i$. Again this is straightforward to verify.

Complexity Both the code before and inside the loop take $O(1)$, and the loop is repeated $n-1$ times. Thus, the complexity is $O(n)$.

4.2.4 Inexact Algorithm

It is tempting to compute $fib(n)$ directly using $fib(n) = round(\varphi^n/\sqrt{5})$. Because we can precompute $1/\sqrt{5}$, that requires $n+1$ floating point multiplications, i.e., also $O(n)$.
However, it is next to impossible verify the correctness of the algorithm. We know that the formula $fib(n) = round(\varphi^n/\sqrt{5})$ is true, but that has no immediate to floating point arithmetic. Rounding errors will accumulate over time and may eventually lead to a false result.

4.2.5 Sublinear Algorithm

Maybe surprisingly, we can still do better. Inspecting the body of the while loop in the linear algorithm, we see that we can rewrite the assignments as

$$(current, prev) := (current + prev, current)$$

which we can write in matrix form as

$$(current, prev) := (current, prev) \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Thus, we obtain

$$(fib(n), fib(n-1)) = (1,0) \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \qquad \text{for } n > 0$$

The n-th power of a matrix can be computed in $O(\log n)$ in the same way as in Sect. 4.1.3. Thus, we can compute $fib(n)$ with logarithmic complexity.

4.3 Matrices

4.3.1 Specification

We write $\mathbb{Z}^{mn}$ for the set $(\mathbb{Z}^n)^m$ of vectors over vectors (i.e., matrices) over integers.
We define two operations on matrices:

- Addition: For of $x, y \in \mathbb{Z}^{mn}$, we define $x + y \in \mathbb{Z}^{mn}$ by

$$(x + y)_{ij} = x_{ij} + y_{ij}$$

- Multiplication: For $x \in \mathbb{Z}^{lm}$ and $y \in \mathbb{Z}^{mn}$, we define $x \cdot y \in \mathbb{Z}^{ln}$ by

$$(x \cdot y)_{ij} = x_{i1} \cdot y_{1j} + \ldots + x_{im} \cdot y_{mj}$$

4.3.2 Naive Algorithms

Vectors and matrices are best stored using arrays. We assume that

- Mat is the data structure of arrays of arrays of the same length of integers,
- if x is an object of Mat, then $x.rows$ is the length of the array and $x.columns$ is the length of the inner arrays,
- **new** $Mat(m, n)$ produces a new array of length m of arrays of length n in which all fields are initialized as 0.

Then we have the straightforward algorithms

```
fun add(x : Mat, y : Mat) : Mat =
    r = new Mat(x.rows, x.columns)
    for i from 1 to x.rows
        for j from 1 to x.columns
            r.i.j := x.i.j + y.i.j
    return r

fun mult(x : Mat, y : Mat) : Mat =
    r = new Mat(x.rows, y.columns)
    for i from 1 to x.rows
        for j from 1 to y.columns
            for k from 1 to x.columns
                r.i.j := r.i.j + x.i.k · y.k.j
    return r
```

Correctness The algorithms directly implement the definitions. Thus, correctness—seemingly—obvious.

But there is one subtlety: The functions take two arbitrary matrices—there is no way to force the user to pass matrices of the correct dimensions. Therefore, we have to state correctness a bit more carefully:

- for $z := add(x, y)$
  precondition: $x.rows == y.rows$ and $x.columns == y.columns$,
  postcondition: $z == x + y$ and $z.rows == x.rows$ and $z.columns == x.columns$.
- for $z := mult(x, y)$
  precondition: $x.rows == y.columns$
  postcondition: $z := mult(x, y)$ is $x \cdot y$ and $z.rows == x.rows$ and $z.columns == y.columns$

Then we can easily show that *add* and *mult* are correct in the sense that the precondition implies the postcondition.

Complexity Assuming that all additions and multiplications take constant time, the complexity is easy to analyze. For addition it is $\Theta(mn)$ and for multiplication $\Theta(lmn)$ where l, m, and n are the dimensions of the respective matrices.

For addition, we can immediately see that we cannot improve on $\Theta(mn)$: Just creating the new array and returning it already takes $\Theta(mn)$ steps. Thus, $\Theta(mn)$ is the complexity of the specification, and the naive algorithm is optimal.

This is not obvious for multiplication. Using the same argument, we can say that the complexity of multiplication is $\Omega(ln)$. But there cannot be an $\Theta(ln)$-algorithm because m must matter—if m increases, it must take longer.

4.3.3 Strassen's Algorithm

Inspecting the definition of matrix multiplication, we see that we can split up matrices into rectangular areas of submatrices, for example, like so:

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} & \begin{pmatrix} x_{13} & x_{14} \\ x_{23} & x_{24} \end{pmatrix} \\ \begin{pmatrix} x_{31} & x_{32} \\ x_{41} & x_{42} \end{pmatrix} & \begin{pmatrix} x_{33} & x_{34} \\ x_{43} & x_{44} \end{pmatrix} \end{pmatrix}$$

Moreover, if matrices are split up like that, we can still obtain their product in the same way using recursive matrix multiplication:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} = \begin{pmatrix} p & q \\ r & s \end{pmatrix}$$

Strassen's algorithm works in the general. But for simplicity, we only consider the case $l = m = n$, i.e., we are multiplying square matrices. Then the naive algorithm has complexity $\Theta(n^3)$, and we know the specification has complexity $\Omega(n^2)$. The question is to find a solution in between.

We further simplify to $n = 2^k$, i.e., we can recursively subdivide our 2^k-matrices 4 2^{k-1}-matrices. Then we can design a recursive algorithm that only needs k nested recursions.

The complexity depends on the details of the implementation. Naively, computing p, q, r, s requires 8 recursive calls to multiplications and 4 additions of 2^{k-1}-matrices. That yields

$$C(n) = 8 \cdot C(n/2) + \Theta(n^2) = \ldots = 8^k \cdot C(1) + \Theta(n^2)$$

Because $k = \log_2 n$ and $C(1) \in O(1)$, that yields $C(n) \in \Theta(n^{\log_2 8} = \Theta(n^3)$.

However, Strassen observed that we can do better. With some fiddling around, we can replace the 8 multiplications and 4 additions with 7 multiplications and 18 additions. The extra additions do not harm because they are $\Theta(n^2)$. But turning the 8 into a 7 yields $C(n) = \Theta(n^{\log_2 7}$. Thus, Strassen's algorithm reduces n^3 to $n^{2.81\ldots}$, which can yield practically relevant improvements for relatively small n, e.g., $n \approx 30$.

Even more efficient algorithms are found regularly. The current record is $\Theta(n^{2.37\ldots})$. However, the sufficiently large n for which these are actually faster than Strassen's algorithm is so large that they have no practical relevance at the moment.

Chapter 5

Example: Lists and Sorting

5.1 Specification

Lists are the most important non-primitive data structure in computer science.

5.1.1 Lists

For a set A, the set A^* contains all lists $[a_0, \ldots, a_{l-1}]$ with elements $a_i \in A$ for some $l \in \mathbb{N}$. l is called the length of the list.
The following table specifies the most important functions involving lists:

function	returns	abbreviation
$nil \in A^*$	$[]$	
$range(m \in \mathbb{N}, n \in \mathbb{N}) \in \mathbb{N}^*$	$[m, \ldots, n-1]$ or $[]$ if $m \geq n$	
below, let $l \in A^*$ be of the form $[a_0, \ldots, a_{l-1}]$ and assume $n < l$		
$length(x \in A^*) \in \mathbb{N}$	l	
$get(x \in A^*, n \in \mathbb{N}) \in A^*$	a_n	x_n or $x[n]$
$append(x \in A^*, y \in A*) \in A^*$	$[a_0, \ldots, a_{l-1}, b_0, \ldots, b_{k-1}]$ if $y = [b_0, \ldots, b_{k-1}]$	$x + y$
$map(x \in A^*, f \in A \to B) \in B^*$	$[f(a_0), \ldots, f(a_{l-1})]$	$l\ map\ f$
$fold(x \in A^*, b \in B, f \in A * B \to B) \in B$	$f(a_1, f(a_2, \ldots, f(a_n, b)) \ldots)$	
$delete(x \in A^*, n \in \mathbb{N}) \in A^*$	$[a_0, \ldots, a_{n-1}, a_{n+1}, \ldots, a_{l-1}]$	
$insert(x \in A^*, a \in A, n \in \mathbb{N}) \in A^*$	$[a_0, \ldots, a_{n-1}, a, a_n, a_{n+1}, \ldots, a_{l-1}]$	
$update(x \in A^*, a \in A, n \in \mathbb{N}) \in A^*$	$[a_0, \ldots, a_{n-1}, a, a_{n+1}, \ldots, a_{l-1}]$	

These are split into three groups:

- The first group contains functions to create new lists. These are important to have any lists.
- The second group contains functions that take a list $l \in A^*$ as their first argument and return data about l or use l to build new data.
- The third group also takes a list $l \in A^*$ but also returns an element of A^*. This distinction is irrelevant in mathematics but critical in computer science: These functions may be implemented using in-place-updates. With in-place update, the list l is changed to become the intended result. The original value of l is lost in the process. If this is the case, we speak of *mutable* lists.

The following table specifies the most important functions on mutable lists. Instead of returning a new list, they have the effect of assigning a new value to the first argument.

function	returns	effect	abbreviation
below, let $l \in A^*$ be of the form $[a_0, \ldots, a_{l-1}]$ and assume $n < l$			
$delete(x \in A^*, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a_{n+1}, \ldots, a_{l-1}]$	
$insert(x \in A^*, a \in A, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a, a_n, a_{n+1}, \ldots, a_{l-1}]$	
$update(x \in A^*, a \in A, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a, a_{n+1}, \ldots, a_{l-1}]$	$x_n := a$ or $x[n] := a$

5.1.2 Sorting

Sorting a list is intuitively straightforward. We need a function that takes a list and returns a list with the same elements in a different order, namely such that all elements occur according to their size.

Example 5.1. Consider $x = [4, 6, 5, 3, 5, 0] \in \mathbb{N}^*$. Then $sort(x)$ must yield $[0, 3, 4, 5, 5, 6]$.
Here we made the implicit assumption that we want to sort with respect to the $\leq$-order on $\mathbb{N}$. We could also use the $\geq$-order. Then $sort(x)$ should return $[6, 5, 5, 4, 3, 0]$.
Thus, sorting always depends on the chosen order.

Definition 5.2 (Sorting)**.** Fix a set A and a total order $\leq$ on A.
A list $x = [a_0, \ldots, a_l] \in A^*$ is called $\leq$-**sorted** if $a_0 \leq a_1 \leq \ldots \leq a_{l-1} \leq a_l$.
Let $count(x \in A^*, a \in A) \in \mathbb{N}$ be the number of times that a occurs in x. Two list $x, y \in A^*$ are a **permutation** of each other if $count(x, a) = count(y, a)$ for all $a \in A$.
$sort : A^* \to A^*$ is called a $\leq$-**sorting** function if for all $x \in A^*$, the list $sort(x)$ is a $\leq$-sorted permutation of x.

As usual we check that the specification indeed defines a function:

Theorem 5.3 (Uniqueness)**.** *The function sort from Def.* **??** *exists uniquely.*

Proof. Because $\leq$ is assumed to be total, every list x has a unique least element, which must occur first in $sort(x)$. By induction on the length of x, we show that all elements of $sort(x)$ are determined. □

For immutable lists, the above definition is all the specification we need. For mutable lists, we specify an alternative sorting function that does not create a new list:

Definition 5.4 (In-place Sorting)**.** An effectful function $sort$ that takes an argument $x \in A^*$ and has the side-effect of modifying the value v of x to v' is called an **in-place** $\leq$-**sorting** function if $v' = s(v)$ for a $\leq$-sorting function s.

5.1.3 Sorting by a Property

Often we do not have a total order on A, and we want to sort according to a certain property. The property must be given by a function $p : A \to P$ such that we have a total order $\leq$ on P.
For example, we may want to sort a list of students by age. Then $A = Student$, $P = \mathbb{N}$, and $p : (s \in Student) \mapsto age(s)$.
However, there may be ties: A list may contain multiple different elements that agree in the value of p. To break, we require that the order in the original list should be preserved. Formally:

Definition 5.5 (Sorting by Property)**.** Fix sets A and P, a function $p : A \to P$, and a total order $\leq$ on P.
Given a list $x \in A^*$, we define a total order $\leq^p$ on the elements of x as follows:

$$x_i \leq^p x_j \quad \text{iff} \quad p(x_i) < p(x_j) \quad \text{or} \quad p(x_i) = p(x_j) \text{ and } i \leq j$$

$sort : A^* \to A^*$ is called a **stable sorting** function for p and $\leq$ if it is a sort function for $\leq^p$.

Note that normal sorting becomes a special case of sorting by property using $P = A$ and $p(a) = a$.

5.1.4 Why Do We Care About Sorting?

Thus, a good, modern programmer might respond as follows:

1. How do you implement sorting a list? — I call the *sort* function of my programming language's basic library.
2. OK, but what if there is no *sort* function? — I import a library that provides it.

3. OK, but what if there is no such library? — I use a different programming language.
4. OK, but what if circumstances beyond your control prevent you from using third-party libraries? — I copy-paste a definition from the internet.[1]

Thus, for most people the only realistic situations in which to implement sorting algorithms is in exams, job interviews, or similar situations. Then the question is never actually about sorting—it just uses sorting as an example to see whether the programmer understands how to design algorithms, analyze their complexity, and verify their correctness.

In any case, sorting is an extremely good subject for an introductory computer science class because it

- is an elementary problem that is easy to understand for students,
- is complex enough to exhibit many important general principles in interesting ways,
- is simple enough for all analysis to be doable manually,
- has multiple solutions, none of which is better than all the others,
- is extremely well-studied,
- is widely taught so that the internet is full of good visualizations that help learners.

5.2 Design: Data Structures for Lists

Besides natural numbers, the most important examples of a data structure are lists. There are many different data structures for lists that differ subtly in how simply and/or efficiently the various functions can be implemented.

5.2.1 Immutable Lists

For immutable lists, functions like *delete*, *insert*, and *update* (see Sect. A.5.4) always return new lists. That may require copying, which takes more time and memory.

Functional Style: Lists as an Inductive Type

Functional languages usually implement lists an in inductive data type:

$$\textbf{data}\ List[A] = nil \mid cons(head : A,\ tail : List[A])$$

Now the list $[1, 2, 3]$ is built as $cons(1, cons(2, cons(3, nil)))$.

Then functions on lists are implemented using recursion and pattern-matching. For example:

$$\begin{array}{l}\textbf{fun}\ map(x : List[A],\ f : A \to B) : List[B] = \\ \quad \textbf{match}\ x \\ \qquad nil \mapsto nil \\ \qquad cons(h, t) \mapsto cons(f(h), map(t, f))\end{array}$$

Object-Oriented Style: Linked Lists

Every inductive data type can also be systematically realized in an object-oriented language. The correspondence is as follows:

inductive type	class	example: lists
name of the type	abstract class	$List$
parameters of the type	parameters of the class	A
constructor	concrete subclass	e.g., *cons*
constructor arguments	constructor arguments	$head : A, tail : List[A]$

A basic realization looks as follows:

[1] Nowadays an internet search for elementary problems almost always finds a solution for every programming language, usually on http://www.stackexchange.org.

```
abstract class List[A]()
class nil[A]() extends List[A]()
class cons[A](head : A, tail : List[A]) extends List[A]()
```

Now the list $[1, 2, 3]$ is built as **new** $cons(1,$ **new** $cons(2,$ **new** $cons(3,$ **new** $nil())))$.
Instead of pattern-matching, we have to use instance-checking to split cases. For example:

```
fun map(x : List[A], f : A → B) : List[B] =
    if x isInstanceOf nil
        new nil()
    else
        xc := x asInstanceOf cons
        new cons(f(xc.head), map(x.tail, f))
```

Complexity

Most operations on lists are linear because the algorithm must traverse the whole list. For example, the straightforward implementation of $length$ takes $O(n)$.
Similarly, $get(x, i)$ takes i steps to find the element. This is n in the worst case and $n/2$ on average. So it also takes $O(n)$.
In general, immutable lists require copying the list, whenever we insert, delete, or update elements. These algorithms must traverse the list. Therefore, they usually take $O(n)$ time where n is the length of the list.
In the case of $map(x, f)$ and $fold(x, a, f)$, the complexity depends on the passed function f. However, because the run time of f does not depend on the length of the list, it takes constant time c. Thus, the overall run time is $O(cn) = O(n)$.
However, there is one exception: prepending an element takes $O(1)$. This is because we can prepend to x simply by calling $cons(a, x)$. Similarly, removing the first element takes $O(1)$.

5.2.2 Mutable Lists

Mutable lists allow assignments to the individual elements of the list. This allows updating an element without copying the list.
Because we can update the list in place, it becomes critical how exactly the list is stored in memory. Three cases are of great importance:

data structure	memory layout	remark
array	all in a row	easy to find elements but difficult to extend length
linked list	every element points to next one	easy to change but traversal needed
doubly-linked list	every element points to next and previous	traversal in both directions, more overhead
growable array	linked list of arrays	compromise between the above

Arrays

In an array all elements are stored in a row in memory.
For example, the list $x = [1, 2, 5]$ is stored in 3 consecutive memory locations:

variable	type	value
x	$\mathbb{N}^*$	P

location	value
P	1
$P + 1$	3
$P + 2$	5

That allows implementing *get* and *update* in $O(1)$. $get(x, n)$ is evaluated by retrieving the element in memory location $P + n$. That takes one step to retrieve x, one step for the addition, and one step to retrieve the element at $P + n$. $update(x, a, n)$ works accordingly.
Inserting and deleting elements still takes $O(n)$. For example, we can implement deleting by:

```
fun delete(x : List[A], n : ℕ) =
    for i from n to length(x) − 1
        x_n := x_{n+1}
```

Inserting an element into an array is difficult though: The memory location behind the array may not be available because it was already used for something else. Therefore, arrays are often realized in such a way that the programmer chooses a priority the maximal length of the array. Thus, technically this data structure does not realize the set A^* but the set A^n for some length n.

Linked Lists

Mutable linked list consist of a reference to the first element. Each element consists of a value and a reference to its successor.

```
class List[A](head : Elem[A])
class Elem[A](value : A, next : Elem[A])
```

Technically, *head* and *next* should have the type $Elem(A)^?$ to allow for empty lists and the end of the list, respectively. However, object-oriented programmers usually use a dirty trick where the built-in value *null* is used those cases.
Now the list $[1, 2, 5]$ is built as $x :=$ **new** $List($**new** $Elem(1,$ **new** $Elem(2,$ **new** $Elem(5, null))))$. It is stored in memory as

variable	type	value
x	$\mathbb{N}^*$	P

location	value
$P.head$	Q
$Q.value$	1
$Q.next$	R
$R.value$	2
$R.next$	S
$S.value$	5
$S.next$	$null$

Deletion can now be realized in-place as follows

```
fun delete(x : List[A], n : ℕ) =
    if n == 0
        x.head := x.head.next
    else
        previous := x.head
        current := x.head.next
        for i from 1 to n − 1
            previous := current
            current := current.next
            previous.next := current.next
```

Like immutable lists, linked-lists take $O(n)$ for most operations. However, they still perform better because changes can be done in-place. Moreover, they require $O(1)$ memory whereas immutable lists require $O(n)$ memory to copy the list.
We can also define a constant-time variant of *insert*. Instead of taking the position n at which to insert (which takes linear time to find), we take the element after which to insert:

```
fun insert(x : List[A], after : Elem[A], a : A) =
    after.next := new Elem(a, after.next)
```

The same trick does not work for *delete*: Even if we have the element that we want to delete, we still need to search for predecessor to update the list.

Doubly-Linked Lists

Doubly-linked linked list are the same as linked lists except that each element also knows its predecessor (*null* for the first element). Moreover, the list knows its first and last element. Thus, a doubly-linked list can be traversed in both directions.

```
class List[A](head : Elem[A], last : Elem[A])
class Elem[A](value : A, previous : Elem[A], next : Elem[A])
```

Now the list $x = [1, 2, 5]$ is stored in memory as

variable	type	value
x	$\mathbb{N}^*$	P

location	value
$P.head$	Q
$P.last$	S
$Q.value$	1
$Q.previous$	*null*
$Q.next$	R
$R.value$	2
$R.previous$	Q
$R.next$	S
$S.value$	5
$S.previous$	R
$S.next$	*null*

In a double-linked list, we can define constant-time variants for both *insert* and *delete*. For example:

```
fun delete(x : List[A], e : Elem[A]) =
    if e.previous == null
        x.head := e.next
    else
        e.previous.next := e.next
    if e.next == null
        x.last := e.previous
    else
        e.next.previous := e.previous
```

Growable Arrays

Growable arrays are a compromise between arrays and linked lists. Initially, they behave like an array with a fixed length l. However, we insert an element such that the length become $> l$, we create a second array of length l (elsewhere in memory) and connect the two.

Retrieval and update technically are linear now. To access the element in position n, we have to make n/l retrievals to jump to the needed array. Because is l constant, that yields $O(n)$ retrievals. However, l is usually large so that element access is only a little slower than for an array and much faster than for a linked list.

5.3 Design: Algorithms for Sorting

We assume a fixed set A and a fixed comparison function $\leq: A \times A \to \mathbb{B}$. For $x \in A^*$, we write $Sorted(x)$ if x is $\leq$-sorted.

Auxiliary Functions Many in-place sorting algorithms have to swap two elements in a mutable list at some point. Therefore, we define an auxiliary function

```
fun swap(x : MutableList[A], i : ℕ, j : ℕ) =
    h := x_i
    x_i := x_j
    x_j := h
```

It is easy to see that this function indeed has the side effect of swapping two elements in x. If x is an array, the complexity of *swap* is $O(1)$.

5.3.1 Bubble Sort

Bubble sort is a stable in-place sorting algorithm that closely follows the natural way how a human would sort. The idea is to find two elements that are not in order and swap them. If no such elements exist, the list is sorted.

```
fun bubblesort(x : Array[A]) =
    sorted := false
    while !sorted
        sorted := true
        for i from 0 to length(x) − 2
            if !x[i] ≤ x[i + 1]
                sorted := false
                swap(x, i, i + 1)
```

Correctness The for-loop compares all $length(x) - 1$ pairs of neighboring elements. It sets *sorted* to *false* if the list is not sorted. Thus, we obtain the loop invariant $F(x, sorted) = sorted == Sorted(x)$, which immediately yields partial correctness.
Total correctness follows from the termination ordering

$$T(x, sorted) = \text{number of pairs } i, j \text{ such that } !x_i \leq x_j + \begin{cases} 1 & \text{if } sorted == false \\ 0 & \text{if } sorted == true \end{cases}$$

Indeed, this number decreases in every iteration of the loop in which x is not sorted. The second summand is necessary to make $T(x, sorted)$ also decreases if x is already sorted (which happens exactly one in the last iteration).

Complexity If n is the length of x, each iteration of the while-loop has complexity $\Theta(n)$. Moreover, the while-loop iterates at most n times. That happens in the worst-case: when x is reversely sorted initially. Thus, the complexity is $\Theta(n^2)$.
In the best-case, when x is already sorted initially, the complexity is $\Theta(n)$. That is already optimal because it requires $n - 1$ comparisons to determine that a list is sorted.

5.3.2 Insertion Sort

Insertion is also a stable in-place algorithm.
The idea is to sort increasingly large prefixes of a list x. If $[x_0, \ldots, x_{i-1}]$ is sorted already, the element x_i is inserted among them.

```
fun insertionsort(x : Array[A]) =
    for i from 0 to length(x) − 1
        current := x[i]
        pos := i
        while pos > 0 && !current ≤ x.(pos − 1)          shift elements to the right to make space for current
            x[pos] := x[pos − 1]
```

```
        pos := pos − 1
    x[pos] := current
```

Correctness We use a loop-invariant for the for-loop: $F(x, i) = Sorted([x.0, \ldots, x.(i-1)])$. The preservation of the loop-invariant is non-obvious but easy to verify. It holds initially because the empty list is trivially sorted. That yields partial correctness.
Termination is easy to show using the termination ordering $T(x, i, current, pos) = pos$.

Complexity If n is the length of x, the for-loop runs n times with $i = 0, \ldots, n-1$ Inside, the while-loop runs i times in the worst-case: if x is reversely sorted, all i elements before *current* must be shifted to the right. That sums up to $0 + 1 + \ldots + n - 1 \in \Theta(n^2)$.
Everything else is $O(n)$. Thus, the worst-case complexity is $\Theta(n^2)$.
In the best-case, if x is already sorted, the while-loop never runs, and the complexity is $\Theta(n)$.

5.3.3 Merge Sort

Merge sort is based on the observation that

- sorting smaller lists is much easier than sorting larger lists (because the number of pairs that have to be compared in $\Theta(n^2)$),
- merging two sorted lists is easy (liner time).

Thus, we can divide a list into two halves, sort them recursively, then merge the results. This is similar to the idea of square-and-multiply (Sect. 4.1.3) and an example of the family of divide-and-conquer algorithms.
Because it needs auxiliary memory to do the merging of two half lists into one, it is easiest to implement as non-in-place algorithm. Then the input data structure does not matter and can be assumed to be immutable. The following is a straightforward realization:

```
fun mergesort(x : List[A]) : List[A] =
    n := length(x)
    if n < 2
        x
    else
        k := n div 2
        x1 := mergesort([x.0, ..., x.(k − 1)])
        x2 := mergesort([x.k, ..., x.(n − 1)])
        return merge(x1, x2)

fun merge(x : List[A], y : List[A]) : List[A] =
    xLeft := x
    yLeft := y
    res = []
    while nonempty(xLeft) || nonempty(yLeft)
        takefromX := empty(yLeft) || (nonempty(xLeft) && xLeft.head ≤ yLeft.head)
        if takefromX
            res := cons(xLeft.head, res)
            xLeft := xLeft.tail
        else
            res := cons(yLeft.head, res)
            yLeft := yLeft.tail
    return reverse(res)
```

Correctness Because the function *merge* is not part of the specification, we have to first specify which property we want to prove about it. The needed property for $z := merge(x, y)$ is:

- precondition: $Sorted(x)$ and $Sorted(y)$

- postcondition: $Sorted(z)$ and z is a permutation of $x + y$

Now we can prove each function correct.

First we consider *mergesort*. Partial correctness means to prove $Sorted(mergesort(x))$. That is very easy:

- If $n < 2$, x is trivially sorted.
- Otherwise:
  - $Sorted(x1)$ and $Sorted(x2)$ follow from the recursive call.
  - Then the property of *merge* yields $Sorted(merge(x1, x2))$.

Relative termination is immediate (assuming that *merge* always terminates, which we prove below). A termination ordering is given by $T(x) = length(x)$. Indeed, *mergesort* recurses only into strictly shorted lists.

Second we consider *merge*. We use a loop invariant $F(x, y, xLeft, yLeft, res)$ that states that

- $Sorted(reverse(res))$ and $Sorted(xLeft)$ and $Sorted(yLeft)$
- All elements in res are in $\leq$-relation to all elements in $xLeft + yLeft$.
- $res + xLeft + yLeft$ is a permutation of $x + y$

It is non-obvious but it is straightforward to see that this indeed a loop invariant:

- $reverse(res)$ remains sorted because we always take the smallest element in $yLeft + xRight$ and prepend it to res. In particular, because $xLeft$ and $yLeft$ are sorted, the smallest element must be $xLeft.head$ or $yLeft.head$.
- For the same reason, all elements of res remain smaller than the ones of $xLeft$ and $yLeft$.
- Because we only remove elements from $xLeft$ and $yLeft$, they remain sorted.
- Because every element that is removed from $xLeft$ or $yLeft$ is immediately added to res, they remain a permutation.

To show partial correctness, we see that

- The loop invariant holds initially, which is obvious.
- After completing the loop, $xLeft$ and $yLeft$ are empty.
- Then, using the loop invariant, it is easy to show that $reverse(res)$ is sorted and a permutation of $x + y$.

To show termination, we use $T(x, y, xLeft, yLeft, res) = length(xLeft) + length(yLeft)$. It is easy to see that T is a the termination ordering for the while-loop.

Complexity We have to analyze the complexity of both functions.

First we consider *merge*. Let $n = length(x) + length(y)$.

- The three assignments in the beginning are $O(1)$.
- The while-loop is repeated once for every element of x and y, which requires $\Theta(n)$ steps. The body of the loop takes $O(1)$. So $\Theta(n)$ in total.
- The last step requires reverting res, which has n elements at this point. Reverting a list requires building a new list by traversing the old one. That is $\Theta(n)$ as well.

Thus, the total complexity of *merge* is $\Theta(n) = \Theta(length(x) + length(y))$.

Second we consider *mergesort*. Let $n = length(x)$ and let $C(n)$ be the needed complexity. We compute $C(n)$:

- The assignments and the if-statement are in $O(1)$.
- The recursive calls to *mergesort* take $C(n/2)$ each.
- The call to *merge* takes $\Theta(length(x1) + length(x2)) = \Theta(n)$.

That yields

$$C(n) = 2 \cdot C(n/2) + \Theta(n) = \ldots = 2^k \cdot C(n/2^k) + k \cdot \Theta(n)$$

Using $k = \log_2 n$ and $C(1) = C(0) \in O(1)$, we obtain

$$C(n) = n \cdot O(1) + \log_2 n \cdot \Theta(n) = \Theta(n \log_2 n)$$

Thus, merge sort is quasilinear and thus strictly more efficient than bubble sort and insertion sort.

Contrary to bubble sort and insertion sort, merge sort takes the same amount of time no matter how sorted the input already is. The recursion and the merging happen in essentially the same way independent of the input list. Thus, its best-case complexity is also $\Theta(n \log_2 n)$.

Remark 5.6 (Building the list reversely in *merge*). *merge* could be simplified by always adding the element $xLeft.head$ or $yLeft.head$ to the *end* of *res* instead of the beginning. However, as discussed in Sect. 5.2, adding an element to the beginning of an immutable list takes constant time whereas adding to the end takes linear time. Therefore, if we added elements to the end of *res* would become quadratic instead of linear. Then merge sort as a whole would also be quadratic.

5.3.4 Quick Sort

Quick sort is similar to merge sort in that two sublists are sorted recursively. The main differences are:

- It does not divide the list x in half. Instead if picks some element a from the list (called the *pivot*). Then it divides x into sublists $x1$ and $x2$ containing the elements smaller and greater than x respectively.
  No merging is necessary because all elements in $x1$ are smaller than all elements in $x2$. Thus the sorted list is $quicksort(x1) + x + quicksort(x2)$.
- To divide the list, quick sort has to traverse and reorder the list anyway. Therefore, it can easily be implemented in-place avoiding the use of auxiliary memory.

When implemented as an in-place sorting algorithm, the recursive call takes two additional arguments: two numbers $first$ and $last$ that describe the sublist that should be sorted. Carrying along auxiliary information is very typical for recursive algorithms. Therefore, we often find pairs of function:

- A recursive function that takes additional arguments.
  That is *quicksortSublist* below, which takes the entire list and the information about which sublist to sort.
- A non-recursive function that does nothing but the other function with the initial arguments.
  That is *quicksort* below, which calls *quicksortSublist* on the entire list (e.g., on the sublist from 0 to the end of x).

```
fun quicksort(x : Array[A]) =
    quicksortSublist(x, 0, length(x) − 1)

fun quicksortSublist(x : Array[A], first : ℕ, last : ℕ) =
    if first ≥ last
        return
    else
        pivot := A[last]
        pivotPos := first
              loop invariant: x[k] ≤ pivot for k = first, ..., pivotPos − 1 and pivot ≤ x[k] for k = pivotPos, ..., j − 1
        for j from first to last − 1
            if x[j] ≤ pivot
                swap(x, pivotPos, j)
                pivotPos := pivotPos + 1
        swap(x, pivotPos, last)

        quicksortSubList(x, first, pivotPos − 1)
        quicksortSubList(x, pivotPos + 1, last)
```

Correctness Before proving correctness we have to specify the behavior of the auxiliary function *quicksortSublist*:

- precondition: none
- postcondition: $Sorted([x_{first}, \ldots, x_{last}])$

Then the correctness of *quicksort* follows immediately from that of *quicksortSublist*.

Now we prove the partial correctness of *quicksortSublist*. First, the base case is trivially correct: It does nothing for lists of length 0 or 1. For the recursive case, we prove that the following two properties holds just before the two recursive calls:

- The sublist $[x_{first}, \ldots, x_{last}]$ is a permutation of its original value, and no other elements of x have changed. That is easy to because we only change x by calling *swap* on positions between $first$ and $last$.
- All values x_k are

- smaller than *pivot* for $k = first, \ldots, pivotPos - 1$,
- equal to *pivot* for $k = pivotPos$,
- greater than *pivot* for $k = pivotPos + 1, \ldots, last$.

We prove that by using the indicated loop invariant for the for-loop. It is trivially true before the for-loop because $first = pivotPos$ and $pivotPos = j$. It is straightforward to check that it is preserved by the for-loop. Therefore, it holds after the for-loop for the value $j = last - 1$. The last call to *swap* moves the pivot element into $x_{pivotPos}$ so that the loop invariant is now also true for $j = last$. Then the needed properties can be seen easily.

To prove the termination of *quicksortSublist*, we use the termination ordering $T(x, first, last) = last - first + 1$ (which is the length of the sublist). That value always decreases because the pivot element is never part of the recursive call.

Complexity Let $n = last - first - 1$ be the length of the sublist. It is easy to see that, apart from the recursion, *quicksortSublist* takes $\Theta(n)$ steps because the for-loop traverses the sublist. Thus, the complexity of quick sort depends entirely on the lengths of the sublists in the recursive calls. However, the pivot position and therefore those lengths are hard to predict.

The best-case complexity arises if the pivot always happens to be in the middle. Then the same reasoning as for merge sort, yields best-case complexity $\Theta(n \log_2 n)$. The worst-case arises if the list is already sorted: then the pivot position will always be the last one, and the two sublists have sizes $n - 1$ and 0. That results in n recursive calls on sublists of length n, $n - 1$, ..., 1 as well as n calls on empty sublists. Consequently, the worst-case complexity is $\Theta(n^2)$.

However, the worst-case complexity does not do quick sort justice because it is much higher than its average-case complexity. Because there are only finitely many permutations for a list of fixed length, the average-case complexity can be worked out systematically. The result is $\Theta(n \log_2 n)$.

It may seem that quick sort is less attractive than merge sort because of its higher worst-case complexity. However, that is a minor effect because the algorithms have the same best-case and average-case complexity. Instead, the constant factors, which are rounded away by using Θ-classes, become important to compare two algorithms with such similar complexity.

Here quick sort is superior to merge sort. Moreover, quick sort can be optimized in many ways. In particular, the choice of the pivot can be tuned in order to increase the likelihood that the two sublists end up having the same size. For example, we can randomly pick 3 elements of the sublist and use the middle-size one as the pivot. With such optimizations, quick sort can become substantially faster than merge sort.

Part II

Important Data Structures

Chapter 6

Finite Data Structures

6.1 Specification

Void The set *void* contains no elements.
The set *void* is rarely used. However, it is usually nice to have when dealing with operations that never return. For example, we can say that throwing an exception or terminating the program returns an element of *void*.

Unit The set *unit* contains exactly one element, which we call ().
The set *unit* is rarely used. However, it is very nice to have when dealing with operations that do not have a particular return value: if an operation returns but returns no value, we say that it returns an element of *unit*.
For example, we assume that assignments and print statements return *unit*. Many methods of mutable data structures also return *unit*. For example, using *unit*, we can specify *insert* for a mutable list from Sect. 5.1.1 as $insert(x \in A^*, a \in A, n \in \mathbb{N}) \in unit$.

Booleans The set *bool* contains exactly two elements, which we call *true* and *false*.

Integers Modulo For $m > 0$, the set $\mathbb{Z}_m$ consists of the elements $\{0, \ldots, m-1\}$.

Enumerations For fresh names $l_1, \ldots, l_n$, the set $enum\{l_1, \ldots, l_n\}$ has n elements, called $l_1, \ldots, l_n$.
The names l_i must be fresh. That means they may not have been defined previously. This is similar to how the name of a new function or class must be fresh.
Defining enumeration set defines new values, namely the l_i.

6.2 Implementation

Void Most programs do not need the type void.

Unit Some programming languages have a built-in type *unit*. If not, we can easily define it as an enumeration.

Booleans Most programming languages have a built-in type *bool*. If not, we can easily define it as an enumeration.

Integers Most programming languages do not offer $\mathbb{Z}_m$ for every m.
However, the base type *int* may be the special case for $m = 2^{32}$ or $m = 2^{64}$.
If we need $\mathbb{Z}_m$ for a specific m, we usually work with *int* and use the mod operation to ensure we remain inside $\mathbb{Z}_m$.

Enumerations Most programming languages allow defining enumeration types in some way.
For example, in SML:

```
datatype answer = yes | no | maybe
```

Or in C:

```
enum {yes, no, maybe} answer;
```

Chapter 7

Number-Based Data-Structures

Chapter 8

List-Like Data Structures

Chapter 9

Tree-Like Data Structures

Chapter 10

Set-Like Data Structures

Chapter 11

Function-Like Data Structures

Chapter 12

Product-Like Data Structures

Chapter 13

Union-Like Data Structures

Chapter 14

Graph-Like Data Structures

Chapter 15

Algebraic Data Structures

Part III

Important Families of Algorithms

Chapter 16

Divide and Conquer

Chapter 17

Dynamic Programming

Chapter 18

Greedy Algorithms

Chapter 19

Recursion

Chapter 20

Backtracking

Chapter 21

Randomization

Chapter 22

Parallelization and Distribution

Chapter 23

Protocols

Part IV

Concrete Languages

Chapter 24

Data Description Languages

24.1 JSON

24.2 XML

24.3 UML

Chapter 25

Programming Languages

Part V

Appendix

Appendix A

Mathematical Preliminaries

A.1 Binary Relations

A binary relation is on A is a subset $\# \subseteq A \times A$. We usually write $(x, y) \in \#$ as $x\#y$.

A.1.1 Classification

Definition A.1 (Properties of Binary Relations)**.** We say that $\#$ is ... if the following holds:

- reflexive: for all x, $x\#x$
- irreflexive: for no x, $x\#x$
- transitive: for all x, y, z, if $x\#y$ and $y\#z$, then $x\#z$
- a strict order: irreflexive and transitive
- a preorder: reflexive and transitive
- anti-symmetric: for all x, y, if $x\#y$ and $y\#x$, then $x = y$
- symmetric: for all x, y, if $x\#y$, then $y\#x$
- an order[1]: preorder and anti-symmetric
- an equivalence: preorder and symmetric
- a total order: order and for all x, y, $x\#y$ or $y\#x$

An element $a \in A$ is called ... of $\#$ if the following holds:

- least element: for all x, $a\#x$
- greatest element: for all x, $x\#a$
- least upper bound for x, y: $x\#a$ and $y\#a$ and for all z, if $x\#z$ and $y\#z$, then $a\#z$
- greatest lower bound for x, y: $a\#x$ and $a\#y$ and for all z, if $z\#x$ and $z\#y$, then $z\#a$

Definition A.2 (Dual Relation)**.** For every relation $\#$, the relation $\#^{-1}$ is defined by $x \#^{-1} y$ iff $y \# x$.
$\#^{-1}$ is called the **dual** of $\#$.

Theorem A.3 (Dual Relation)**.** *If a relation is reflexive/irreflexive/transitive/symmetric/antisymmetric/total, then so is its dual.*

A.1.2 Equivalence Relations

Equivalence relations are usually written using infix symbols whose shape is reminiscent of horizontal lines, such as $=$, $\sim$, or $\equiv$. Often vertically symmetric symbols are used to emphasize the symmetry property.

Definition A.4 (Quotient)**.** Consider a relation $\equiv$ on A. Then

- For $x \in A$, the set $\{y \in A \mid x \equiv y\}$ is called the (equivalence) **class** of x. It is often written as $[x]_{\equiv}$.
- $A/\equiv$ is the set of all classes. It is called the **quotient** of A by $\equiv$.

Theorem A.5. *For a relation $\equiv$ on A, the following are equivalent[2]:*

- *$\equiv$ is an equivalence.*
- *There is a set B and a function $f : A \to B$ such that $x \equiv y$ iff $f(x) = f(y)$.*
- *Every element of A is in exactly one class in $A/\equiv$.*

In particular, the elements of $A/\equiv$

- *are pairwise disjoint,*
- *have A as their overall union.*

A.1.3 Orders

Theorem A.6 (Strict Order vs. Order)**.** *For every strict order $<$ on A, the relation "$x < y$ or $x = y$" is an order.*
For every order $\leq$ on A, the relation "$x \leq y$ and $x \neq y$" is a strict order.

Thus, strict orders and orders come in pairs that carry the same information.
Strict orders are usually written using infix symbols whose shape is reminiscent of a semi-circle that is open to the right, such as $<$, $\subset$, or $\prec$. This emphasizes the anti-symmetry ($x < y$ is very different from $y < x$.) and the transitivity ($< \ldots <$ is still $<$.) The corresponding order is written with an additional horizontal bar at the bottom, i.e., $\leq$, $\subseteq$, or $\preceq$. In both case, the mirrored symbol is used for the dual relation, i.e., $>$, $\supset$, or $\succ$, and $\geq$, $\supseteq$, and $\succeq$.

Theorem A.7. *If $\leq$ is an order, then least element, greatest element, least upper bound of x, y, and greatest lower bound of x, y are unique whenever they exist.*

Theorem A.8 (Preorder vs. Order)**.** *For every preorder $\leq$ on A, the relation "$x \leq y$ or $y \leq x$" is an equivalence.*
For equivalence classes X and Y of the resulting quotient, $x \leq y$ holds for either all pairs or no pair $(x, y) \in X \times Y$. If it holds for all pairs, we write $X \leq Y$ as well.
The relation $\leq$ on the quotient is an order.

A.2 Binary Functions

A binary function on A is a function $\circ : A \times A \to A$. We usually write $\circ(x, y)$ as $x \circ y$.

Definition A.9 (Properties of Binary Functions)**.** We say that $\circ$ is ... if the following holds:

- associative: for all x, y, z, $x \circ (y \circ z) = (x \circ y) \circ z$
- commutative: for all x, y, $x \circ y = y \circ x$
- idempotent: for all x, $x \circ x = x$

An element $a \in A$ is called a ... element of $\circ$ if the following holds:

- left-neutral: for all x, $a \circ x = x$
- right-neutral: for all x, and $x \circ a = x$
- neutral: left-neutral and right-neutral
- left-absorbing: for all x, $a \circ x = a$
- right-absorbing: for all x, $x \circ a = a$
- absorbing: left-absorbing and right-absorbing

Theorem A.10. *Neutral and absorbing element of $\circ$ are unique whenever they exist.*

A.3 The Integer Numbers

A.3.1 Divisibility

Definition A.11 (Divisibility)**.** For $x, y \in \mathbb{Z}$, we write $x|y$ iff there is a $k \in \mathbb{Z}$ such that $x * k = y$.
We say that y is divisible by x or that x divides y.

Remark A.12 (Divisible by 0 and 1). Even though division by 0 is forbidden, the case $x = 0$ is perfectly fine. But it is boring: $0|x$ iff $x = 0$.
Similarly, the case $x = 1$ is trivial: $1|x$ for all x.

Theorem A.13 (Divisibility)**.** *Divisibility has the following properties for all $x, y, z \in Z$*

- *reflexive:* $x|x$
- *transitive: if* $x|y$ *and* $y|z$ *then* $x|z$
- *anti-symmetric for natural numbers* $x, y \in \mathbb{N}$*: if* $x|y$ *and* $y|x$*, then* $x = y$
- 1 *is a least element:* $1|x$
- 0 *is a greatest element:* $x|0$
- $\gcd(x, y)$ *is a greatest lower bound of* x, y
- $\mathrm{lcm}(x, y)$ *is a least upper bound of* x, y

Thus, $|$ *is a preorder on* $\mathbb{Z}$ *and an order on* $\mathbb{N}$.

Divisibility is preserved by arithmetic operations: If $x|m$ *and* $y|m$*, then*

- *preserved by addition:* $x + y|m$
- *preserved by subtraction:* $x - y|m$
- *preserved by multiplication:* $x * y|m$
- *preserved by division if* $x/y \in Z$*:* $x/y|m$
- *preserved by negation of any argument:* $-x|m$ *and* $x| - m$

gcd *has the following properties for all* $x, y \in \mathbb{N}$*:*

- *associative:* $\gcd(\gcd(x, y), z) = \gcd(x, \gcd(y, z))$
- *commutative:* $\gcd(x, y) = \gcd(y, x)$
- *idempotence:* $\gcd(x, x) = x$
- 0 *is a neutral element:* $\gcd(0, x) = x$
- 1 *is an absorbing element:* $\gcd(1, x) = 1$

lcm *has the same properties as* gcd *except that* 1 *is neutral and* 0 *is absorbing.*

Theorem A.14. *For all $x, y \in \mathbb{Z}$, there are numbers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.*
a and b can be computed using the extended Euclidean algorithms.

Definition A.15. If $\gcd(x, y) = 1$, we call x and y **coprime**.
For $x \in \mathbb{N}$, the number of coprime $y \in \{0, \ldots, x - 1\}$ is called $\varphi(x)$. φ is called Euler's **totient function**.

We have $\varphi(0) = 0$, $\varphi(1) = \varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(4) = 1$, and so on. Because $\gcd(x, 0) = x$, we have $\varphi(x) \leq x - 1$. x is prime iff $\varphi(x) = x - 1$.

A.3.2 Equivalence Modulo

Definition A.16 (Equivalence Modulo)**.** For $x, y, m \in \mathbb{Z}$, we write $x \equiv_m y$ iff $m|x - y$.

Theorem A.17 (Relationship between Divisibility and Modulo)**.** *The following are equivalent:*

- $m|n$
- $\equiv_m \supseteq \equiv_n$ *(i.e., for all x, y we have that $x \equiv_n y$ implies $x \equiv_m y$)*
- $n \equiv_m 0$

Remark A.18 (Modulo 0 and 1). In particular, the cases $m = 0$ and $m = 1$ are trivial again:

- $x \equiv_0 y$ iff $x = y$,
- $x \equiv_1 y$ always

Thus, just like 0 and 1 are greatest and least element for $|$, we have that $\equiv_0$ and $\equiv_1$ are the smallest and the largest equivalence relation on $\mathbb{Z}$.

Theorem A.19 (Modulo)**.** *The relation $\equiv_m$ has the following properties*

- *reflexive: $x \equiv_m x$*
- *transitive: if $x \equiv_m y$ and $y \equiv_m z$ then $x \equiv_m z$*
- *symmetric: if $x|y$ then $y|x$*

Thus, it is an equivalence relation.

It is also preserved by arithmetic operations: If $x \equiv_m x'$ and $y \equiv_m y'$, then

- *preserved by addition: $x + y \equiv_m x' + y'$*
- *preserved by subtraction: $x - y \equiv_m x' - y'$*
- *preserved by multiplication: $x * y \equiv_m x' * y'$*
- *preserved by division if $x/y \in Z$ and $x'/y' \in \mathbb{Z}$: $x/y \equiv_m x'/y'$*
- *preserved by negation of both arguments: $-x \equiv_m -x'$*

A.3.3 Arithmetic Modulo

Definition A.20 (Modulus)**.** We write $x \bmod m$ for the smallest $y \in \mathbb{N}$ such that $x \equiv_m y$.
We also write $modulus_m$ for the function $x \mapsto x \bmod m$. We write $\mathbb{Z}_m$ for the image of $modulus_m$.

Remark A.21 (Modulo 0 and 1). The cases $m = 0$ and $m = 1$ are trivial again:

- $x \bmod 0 = x$ and $\mathbb{Z}_0 = \mathbb{Z}$
- $x \bmod 1 = 0$ and $\mathbb{Z}_1 = \{0\}$

Remark A.22 (Possible Values). For $m \neq 0$, we have $x \bmod m \in \{0, \dots, m-1\}$. In particular, there are m possible values $m \bmod x$.
For example, we have $x \bmod 1 \in \{0\}$. And we have $x \bmod 2 = 0$ if x is even and $x \bmod 2 = 1$ if x is odd.

Definition A.23 (Arithmetic Modulo m)**.** For $x, y \in \mathbb{Z}$, we define arithmetic operations modulo m by

$$x \circ_m y = (x \circ y) \bmod m \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, if there is a unique $q \in \mathbb{Z}_m$ such that $q \cdot x \equiv_m y$, we define $x/_m y = q$.

Note that the condition $y|x$ is neither necessary nor sufficient for $x/_m y$ to de defined. For example, $2/_4 2$ is undefined because $1 \cdot 2 \equiv_4 3 \cdot 2 \equiv_4 2$. Conversely, $2/_4 3$ is defined, namely 2.

Theorem A.24 (Arithmetic Modulo m)**.** *For $x, y \in \mathbb{Z}$,* mod *commutes with arithmetic operations in the sense that*

$$(x \circ y) \bmod m = (x \bmod m) \circ_m (y \bmod m) \qquad \text{for} \qquad \circ \in \{+, -, \cdot\}$$

Moreover, $x/_m y$ is defined iff $\gcd(y, m) = 1$ *and*

$$(x/y) \bmod m = (x \bmod m)/_m (y \bmod m) \qquad \text{if} \qquad y|x$$

$$x/_m y = x \cdot_m a \qquad \text{if } ay + bm = 1 \text{ as in see Thm. A.14}$$

Theorem A.25 (Fermat's Little Theorem)**.** *For all prime numbers p and $x \in \mathbb{Z}$, we have that $x^p \equiv_p x$.*
If x and p are coprime, that is equivalent to $x^{p-1} \equiv 1$.

A.3.4 Digit-Base Representations

Fix $m \in \mathbb{N} \setminus \{0\}$, which we call the base.

Theorem A.26 (Div-Mod Representation)**.** *Every $x \in \mathbb{Z}$ can be uniquely represented as $a \cdot m + b$ for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_m$.*
Moreover, $b = x \bmod m$. We write $b \operatorname{div} m$ for a.

Definition A.27 (Base-m-Notation)**.** For $d_i \in \mathbb{Z}_m$, we define $(d_k \ \ldots \ d_0)_m = d_k \cdot m^k + \ldots + d_1 \cdot k + d_0$.
The d_i are called digits.

Theorem A.28 (Base-m Representation)**.** *Every $x \in \mathbb{N}$ can be uniquely represented as $(0)_m$ or $(d_k \ \ldots \ d_0)_m$ such that $d_k \neq 0$.*
Moreover, we have $k = \lfloor \log_m x \rfloor$ and $d_0 = x \bmod m$, $d_1 = (x \operatorname{div} m) \bmod m$, $d_2 = ((x \operatorname{div} m) \operatorname{div} m) \bmod m$ and so on.

Example A.29 (Important Bases). We call $(d_k \ \ldots \ d_0)_m$ the binary/octal/decimal/hexadecimal representation if $m = 2, 8, 10, 16$, respectively.
In case $m = 16$, we write the elements of $\mathbb{Z}_m$ as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

A.3.5 Finite Fields

In this section, let $m = p$ be prime.

Construction Then $x/_p y$ is defined for all $x, y \in \mathbb{Z}_p$ with $y \neq 0$. Consequently, $\mathbb{Z}_p$ is a field.
Up to isomorphism, all finite fields are obtained as an n-dimensional vector space $\mathbb{Z}_p^n$ for $n \geq 1$. This field is usually called F_{p^n} because it has p^n elements. From now on, let $q = p^n$.
All elements of F_q are vectors $(a_0, \ldots, a_{n-1})$ for $a_i \in \mathbb{Z}_p$. Addition and subtraction are component-wise, the 0-element is $(0, \ldots, 0)$, the 1-elements is $(1, 0, \ldots, 0)$.
However, multiplication in F_q is tricky. To multiply two elements, we think of the vectors $(a_0, \ldots, a_{n-1})$ as polynomials $a_{n-1}X^{n-1} + \ldots + a_1 X + a_0$, and multiply the polynomials. This can introduce powers X^n and higher, which we eliminate using $X^n = k_{n-1}X^{n-1} + \ldots + k_1 X + k_0$. The resulting polynomial has degree at most $n - 1$, and its coefficient (modulo p) yield the result.
The values k_i always exists but are non-trivial to find. They must be such that the polynomial $X^n - k_{n-1}X^{n-1} - \ldots - k_1 X - k_0$ has no roots in $\mathbb{Z}_p$. There may be multiple polynomials, which may lead to different multiplication operations. However, all of them yield isomorphic fields.

Binary Fields The operations become particularly easy if $p = 2$. The elements of F_{2^n} are just the bit strings of length n. Addition and subtraction are the same operation and can be computed by component-wise XOR. Multiplication is a bit more complex but can be obtained as a sequence of bit-shifts and XORs.

Exponentiation and Logarithm Because F_q has multiplication, we can define natural powers in the usual way:

Definition A.30. For $x \in F^q$ and $l \in \mathbb{N}$, we define $x^l \in F_q$ by $x^0 = 1$ and $x^{l+1} = x \cdot x^l$.
If l is the smallest number such that $x^l = y$, we write $l = \log_x y$ and call n the **discrete q-logarithm** of y with base x.

The powers $1, x, x^2, \ldots \in F_q$ of x can take only $q-1$ different values because F_q has only q elements and x^l can never be 0 (unless $x = 0$). Therefore, they must be periodic:

Theorem A.31. *For every $x \in F_q$, we have $x^q = x$ or equivalently $x^{q-1} = 1$ for $x \neq 0$.*

For some x, the period is indeed $q-1$, i.e., we have $\{1, x, x^2, \ldots, x^{q-1} = F_q \setminus \{0\}$. Those x are called primitive elements of F_q. But the period may be smaller. For example, the powers of 1 are $1, \ldots, 1$, i.e., 1 has period 1. For a non-trivial example consider $p = 5$, $n = 1$, (i.e., $q = 5$): The powers of 4 are $4^0 = 1$, $4^1 = 4$, $4^2 = 16 \bmod 5 = 1$, and $4^3 = 4$.
If the period is smaller, x^l does not take all possible values in F_q. Therefore, $\log_x y$ is not defined for all $y \in F_q$.

Computing x^l is straightforward and can be done efficiently. (If $n > 1$, we first have to find the values k_i needed to do the multiplication, but we can precompute them once and for all.)
Determining whether $\log_x y$ is defined and computing its value is also straightforward: We can enumerate all powers $1, x, x^2, \ldots$ until we find 1 or y. However, no efficient algorithm is known.

A.4 Size of Sets

The size $|S|$ of a set S is a very complex topic of mathematics because there are different degrees of infinity. Specifically, we have that $|\mathcal{P}(S)| > |S|$, i.e., we have infinitely many degrees of infinity.
In computer science, we are only interested in countable sets. We use a very simple definition that writes C for countable and merges all greater sizes into uncountable sets, whose size we write as U.

Definition A.32 (Size of sets)**.** The size $|S| \in \mathbb{N} \cup \{C, U\}$ of a set S is defined by:

- if S is finite: $|S|$ is the number of elements of S
- if S is infinite and bijective to $\mathbb{N}$: $|S| = C$, and we say that S is countable
- if S is infinite and not bijective to $\mathbb{N}$: $|S| = U$, and we say that S is uncountable

We can compute with set sizes as follows:

Definition A.33 (Computing with Sizes)**.** For two sizes $s, t \in \mathbb{N} \cup \{C, U\}$, we define addition, multiplication, and exponentiation by the following tables:

$s+t$	t: $n \in \mathbb{N}$	C	U
$m \in \mathbb{N}$	$m+n$	C	U
C	C	C	U
U	U	U	U

$s*t$	t: $n \in \mathbb{N}$	C	U
$m \in \mathbb{N}$	$m*n$	C	U
C	C	C	U
U	U	U	U

s^t	t: 0	1	$n \in \mathbb{N} \setminus \{0\}$	C	U
0	1	0	0	0	0
1	1	1	1	1	1
$m \in \mathbb{N} \setminus \{0\}$	1	m	m^n	U	U
C	1	C	C	U	U
U	1	U	U	U	U

Because exponentiation s^t is not commutative, the order matters: s is given by the row and t by the column.

The intuition behind these rules is given by the following:

Theorem A.34. *For all sets S, T, we have for the size of the*

- *disjoint union:*
$$|S \uplus T| = |S| + |T|$$
- *Cartesian product:*
$$|S \times T| = |S| * |T|$$
- *set of functions from T to S:*
$$|S^T| = |S|^{|T|}$$

Thus, we can understand the rules for exponentiation as follows. Let us first consider the 4 cases where one of the arguments has size 0 or 1: For every set A

1. there is exactly one function from the empty set (namely the empty function): $|A^{\varnothing}| = 1$,
2. there are as many functions from a singleton set as there are elements of A: $|A^{\{x\}}| = |A|$,
3. there are no functions to the empty set (unless A is empty): $|\varnothing^A| = 0$ if $A \neq \varnothing$,
4. there is exactly one function into a singleton set (namely the constant function): $|\{x\}^A| = 1$,

Now we need only one more rule: The set of functions from a non-empty finite set to a finite/countable/uncountable set is again finite/countable/uncountable. In all other cases, the set of functions is uncountable.

A.5 Important Sets and Functions

The meaning and purpose of a data structure is to describe a set in the sense of mathematics. Similarly, the meaning and purpose of an algorithm is to describe a function between two sets.

Thus, it is helpful to collect some sets and functions as examples. These are typically among the first data structures and algorithms implemented in any programming language and they serve as test cases for evaluating our languages.

A.5.1 Base Sets

When building sets, we have to start somewhere with some sets that are assumed to exist. These are called the *bases sets* or the *primitive sets*.

The following table gives an overview, where we also list the size of each set according to Def. A.32:

set	description/definition	size
typical base sets of mathematics[3]		
$\varnothing$	empty set	0
$\mathbb{N}$	natural numbers	C
$\mathbb{Z}$	integers	C
$\mathbb{Z}_m$ for $m > 0$	integers modulo m, $\{0, \ldots, m-1\}$ [4]	m
$\mathbb{Q}$	rational numbers	C
$\mathbb{R}$	real numbers	U
additional or alternative base sets used in computer science		
unit	unit type, $\{()\}$, equivalent to $\mathbb{Z}_1$	1
$\mathbb{B}$	booleans, $\{false, true\}$, equivalent to $\mathbb{Z}_2$	2
int	primitive integers, $-2^{n-1}, \ldots, 2^{n-1} - 1$ for machine-dependent n, equivalent to $\mathbb{Z}_{2^n}$ [5]	2^n
float	IEEE floating point approximations of real numbers	C
char	characters	finite[6]
string	lists of characters	C

[3] All of mathematics can be built by using $\varnothing$ as the only base set because the others are definable. But it is common to assume at least the number sets as primitives.

[4] $\mathbb{Z}_0$ also exists but is trivial: $\mathbb{Z}_0 = \mathbb{Z}$.

[5] Primitive integers are the 2^n possible values for a sequence of n bits. Old machines used $n = 8$ (and the integers were called "bytes"), later machines used $n = 16$ (called "words"). Modern machines typically use 32-bit or 64-bit integers. Modern programmers usually—but dangerously—assume that 2^n is much bigger than any number that comes up in practice so that essentially $int = \mathbb{Z}$.

[6] The ASCII standard defined 2^7 or 2^8 characters. Nowadays, we use Unicode characters, which is a constantly growing set containing

A.5.2 Functions on the Base Sets

For every base set, we can define some basic operations. These are usually built-in features of programming languages whenever the respective base set is built-in.
We only list a few examples here.

Numbers

For all number sets, we can define addition, subtraction, multiplication, and division in the usual way.
Some care must be taken when subtracting or dividing because the result may be in a different set. For example, the difference of two natural numbers is not in general a natural number but only an integer (e.g., $3 - 5 \notin \mathbb{N}$). Moreover, division by 0 is always forbidden.

Quotients of the Integers

The function $modulus_m$ (see Sect. A.3.3) for $m \in \mathbb{N}$ maps $x \in \mathbb{Z}$ to $x \bmod m \in \mathbb{Z}_m$.
In programming languages, the set $\mathbb{Z}_m$ is usually not provided. Instead, $x \bmod y$ is built-in as a functions on *int*.

Booleans

On booleans, we can define the usual boolean operations conjunction (usually written & or &&), disjunction (usually written | or ||), and negation (usually written !).
Moreover, we have the equality and inequality functions, which take two objects x, y and return a boolean. These are usually written $x == y$ and $x! = y$ in text files languages and $x = y$ and $x \neq y$ on paper.

A.5.3 Set Constructors

From the base sets, we build all other sets by applying set constructors. Those are operations that take sets and return new sets.
The following table gives an overview, where we also list the size of each set according to Def. A.33:

the characters of virtually any writing system, many scientific symbols, emojis, etc. Many programming languages assume that there is one character for every primitive integers, e.g., typically 2^{32} characters.

set	description/definition	size
	typical constructors in mathematics	
$A \uplus B$	disjoint union	$\lvert A\rvert + \lvert B\rvert$
$A \times B$	(Cartesian) product	$\lvert A\rvert * \lvert B\rvert$
A^n for $n \in \mathbb{N}$	n-dimensional vectors over A	$\lvert A\rvert^n$
B^A or $A \to B$	functions from A to B	$\lvert B\rvert^{\lvert A\rvert}$
$\mathcal{P}(A)$	power set, equivalent to $\mathbb{B}^A$	$2^{\lvert A\rvert} = \begin{cases} 2^n & \text{if } \lvert A\rvert = n \\ U & \text{otherwise} \end{cases}$
$\{x \in A \mid P(x)\}$	subset of A given by property P	$\leq \lvert A\rvert$
$\{f(x) : x \in A\}$	image of operation f when applied to elements of A	$\leq \lvert A\rvert$
A/r	quotient set for an equivalence relation r on A	$\leq \lvert A\rvert$
	selected additional constructors often used in computer science	
A^*	lists over A	$\begin{cases} 1 & \text{if } A = \varnothing \\ U & \text{if } \lvert A\rvert = U \\ C & \text{otherwise} \end{cases}$
$A^?$	optional element[7] of A	$1 + \lvert A\rvert$
	for new names $l_1, \ldots, l_n$	
$enum\{l_1, \ldots, l_n\}$	enumeration: like $\mathbb{Z}_n$ but also introduces named elements l_i of the enumeration	n
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	labeled union: like $A_1 \uplus \ldots \uplus A_n$ but also introduces named injections l_i from A_i into the union	$\lvert A_1\rvert + \ldots + \lvert A_n\rvert$
$\{l_1 : A_1, \ldots, l_n : A_n\}$	record: like $A_1 \times \ldots \times A_n$ but also introduces named projections l_i from the record into A_i	$\lvert A_1\rvert * \ldots * \lvert A_n\rvert$
inductive data types[8]		C
classes[9]		U

A.5.4 Characteristic Functions of the Set Constructors

Every set constructor comes systematically with characteristic functions into and out of the constructed sets C. These functions allow building elements of C or using elements of C for other computations.

For some sets, these functions do not have standard notations in mathematics. In those cases, different programming languages may use slightly different notations.

The following table gives an overview:

set C	build an element of C	use an element x of C
$A_1 \uplus A_2$	$inj_1(a_1)$ or $inj_2(a_2)$ for $a_i \in A_i$	pattern-matching
$A_1 \times A_2$	(a_1, a_2) for $a_i \in A_i$	$x.i \in A_i$ for $i = 1, 2$
A^n	$(a_1, \ldots, a_n)$ for $a_i \in A$	$x.i \in A$ for $i = 1, \ldots, n$
B^A	$(a \in A) \mapsto b(a)$	$x(a)$ for $a \in A$
A^*	$[a_0, \ldots, a_{l-1}]$[10] for $a_i \in A$	pattern-matching
$A^?$	$None$ or $Some(a)$ for $a \in A$	pattern-matching
$enum\{l_1, \ldots, l_n\}$	l_1 or $\ldots$ or l_n	switch statement or pattern-matching
$l_1(A_1) \mid \ldots \mid l_n(A_n)$	$l_1(a_1)$ or $\ldots$ or $l_n(a_n)$ for $a_i \in A_i$	pattern-matching
$\{l_1 : A_1, \ldots, l_n : A_n\}$	$\{l_1 = a_1, \ldots, l_n = a_n\}$ for $a_i \in A_i$	$x.l_i \in A_i$
inductive data type A	$l(u_1, \ldots, u_n)$ for a constructor l of A	pattern-matching
class A	**new** A	$x.l(u_1, \ldots, u_n)$ for a field l of A

[7] An optional element of A is either absent or an element of A.

[8] These are too complex to define at this point. They are a key feature of functional programming languages like SML.

[9] These are too complex to define at this point. They are a key feature of object-oriented programming languages like Java.

[10] Mathematicians start counting at 1 and would usually write a list of length n as $[a_1, \ldots, a_n]$. However, computer scientists always start counting at 0 and therefore write it as $[a_0, \ldots, a_{n-1}]$. We use the computer science numbering here.

Bibliography

[CLR10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 2010.

[EucBC] Euclid. *Elements*. around 300 BC. English translation by T. Heath (1956) available online.

[Hil00] D. Hilbert. Mathematische Probleme. *Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, pages 253–297, 1900.

[Hil26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–90, 1926.

[Knu73] D. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.