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. 3.14.

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 at useful scales yet.

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 programs consisting of types and functions.
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.
   A good specification should be:
   - adequate: actually describe the problem that needs solving
   - simple: easy to understand
   - unambiguous: impossible to misunderstand
   - consistent: possible to realize
   - (optionally) complete: no freedom in what it means (An incomplete specification is not necessarily a flaw. For example, one might specify a function on integers without saying what should happen for negative input.)

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 gcdRec(m : ℕ, n : ℕ) : ℕ =
    if n == 0
        m
    else
        gcd(n, m mod 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 gcdRec(m:nat, n: nat): nat = if n = zero then m else gcdRec(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 some 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 as before. For the new variable y, 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 to their specification and make 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 no 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 of x under the hood.

In many programming languages (in particular object-oriented ones), mutable data structures are called *classes*. Some functions involving a mutable data structure will make use of mutability, some will not. This must be part of the specification of each 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, threads, 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 results 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**.

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

- 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 mutable 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
  - 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
- the set of currently active threads
- 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, computers 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 environment-dependency and side-effects as much as possible.

2.4 Parametric Polymorphism

Many important data structures and algorithms are polymorphic in the following sense:

Definition 2.17 (Polymorphism)**.** A **polymorphic data structure** D is an operator that maps data structures $D_1, \ldots, D_n$ to a data structure $D[D_1, \ldots, D_n]$.
A **polymorphic algorithm** F is an operator that maps data structures $D_1, \ldots, D_n$ to an algorithm $F[D_1, \ldots, D_n]$.
The D_i are called the **type parameters** or **type arguments** of the data structure/algorithm.

This is best understood by example:

Example 2.18 (Lists)*.* Lists are a polymorphic data structure. A^* is the set of lists whose elements have type A. Any data structure for A^* should take A as a type parameter.
For example, $List[A]$ may be a data structure such that $List[int]$ is the type of lists of integers.
Most algorithms about lists are polymorphic as well. For example, reversing a list can be realized using an algorithm

```
fun reverse[A](x : List[A]) : List[A] =
    ...
```

Terminology 2.19. There are many different concepts of polymorphism that are (correctly, confusingly, or even wrongly) called *polymorphism*. The special kind described here is usually called *parametric polymorphism*.
Both terminology and notations vary across programming languages, communities, and textbooks.

A more difficult example arises if we want to sort a list: To sort a list over A, we need a comparison function $\leq (x : A, y : A) : bool$. Moreover, $\leq$ has to be a total order. We can handle that using abstract classes:

Example 2.20. Consider the following polymorphic abstract class for total orders:

abstract class $TotOrd[A]()$
  fun $lessOrEqual(x : A, y : A) : bool =$

It requires a function *lessOrEqual* that provides the comparison $\leq$. The axioms for being a total order can usually not be programmed—they can only be added as part of the documentation.
Then a polymorphic sorting algorithm could look like

fun $sort[A](ord : TotOrd[A], x : List[A]) : List[A] =$
  ...

Notation 2.21 (Omitting Type Parameters). Most of the time it is possible to omit the type parameters when calling a polymorphic function without ambiguity. For example, if $l : List[int]$, we can simply say $revert(l)$ instead of $revert[int](l)$—both human readers and compiler can infer the type argument.
Most programming languages that allow polymorphism also allow omitting parameters if they can be inferred uniquely. It is also allowed to do so in examples and pseudo-code.

In Programming Languages

Even though polymorphism is relatively simple mathematically, not all programming languages do a good job of implementing it. Therefore, we will often gloss over issues of polymorphism when giving algorithms.
But we give a few examples of polymorphism in a few typed programming languages.

Scala Scala's syntax is very similar to te pseudo-code used in these notes:

```
abstract class TotOrd[A] {
  def lessOrEqual(x:A, y:A): Boolean
}

object IntSmaller extends TotOrd[Int] {
  def lessOrEqual(x:Int, y:Int): Boolean = x <= y
}

object Sort {
  def sort[A](ord: TotOrd[A], x: List[A]): List[A] = {
    ...
  }
}

object Test {
  def main(args: Array[String]) {
    sort[Int](IntSmaller, List(4,3,5))
  }
}
```

Java In Java, polymorphic data structures are called generics. It uses angular instead of square brackets and puts the parameter types of a polymorphic algorithm before the return type instead of after the name:

```
interface TotOrd<A> {
  public Boolean lessOrEqual(A x, A y);
}

class Sort {
```

```
  static <A> List<A> sort(TotOrd<A> ord, List<A> x) {
    ...
  }
}

class IntSmaller implements TotOrd<Integer> {
  public Boolean lessOrEqual(Integer x, Integer y) {
    return x <= y;
  }
  public static IntSmaller it = new IntSmaller();
}

class Test {
  public static void main (String[] args) {
    Sort.sort(IntSmaller.it, Arrays.asList(3,5,4));
  }
}
```

C++ In C++, we can use templates to implement polymorphism. C++ also uses angular brackets, and the parameter types of classes and functions must be declared using the template keyword.

```
using namespace std;
#include <list>

template <class A>
class TotOrd {
  bool lessOrEqual(A x, A y);
};

class IntSmaller: public TotOrd<int> {
  bool lessOrEqual(int x, int y) {return x <= y;}
};
IntSmaller* is = new IntSmaller();

template <class A>
list<A> sort(TotOrd<A> ord, list<A> x) {
  ...
};

int test() {
    sort<int>(*is, {3,5,4});
}
```

SML In SML, we do not have abstract classes, but we can use a datatype instead. The type parameters of polymorphic types and functions are not declared explicitly. Instead, they are implicit given as variables like 'a.

```
datatype 'a TotOrder = TotOrder of 'a * 'a -> bool
fun lessOrEqual(ord: 'a TotOrder): 'a * 'a -> bool = case ord of TotOrder(f) => f

val IntSmaller: int TotOrder = TotOrder(fn (x,y) => x <= y)

fun sort(ord: 'a TotOrder, x: 'a list) = x

fun test() = sort(IntSmaller, [3,5,4])
```

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 $F(x)$.

Data Structures

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

More severely, we cannot in general give a correct data structure for $\{F(x) : x \in A\}$ at all. Even if F is computable, we cannot give an algorithm that determines whether a given object is in that set.

Neither can we give a correct data structure for A/r for $r \in A \times A \to \mathbb{B}$. Even if r is computable, we cannot give an algorithm for equality of elements of A/r.

Algorithms

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.

To make verification more systematic, we usually split the specification into two parts: precondition and postcondition. Independently, we split the verification arguments into two independent steps: termination and partial correctness. The definitions are as follows:

Definition 3.6. Consider an algorithm A for a function $f(x_1 \in I_1, \ldots, x_n \in I_n) \in O$.

We define:

- A **precondition** for A is a formula $Pre(x_1, \ldots, x_n)$ about the inputs.
- A **postcondition** for A is a formula $Post(x_1, \ldots, x_n, r)$ about the inputs and the output.
- A **terminates for** $v_1, \ldots, v_n$ if running A with these inputs finishes in finitely many steps.
- A **terminates** if it terminates whenever $Pre(v_1, \ldots, v_n)$.
- A is **partially correct** if for all $v_1, \ldots, v_n$
  - if $Pre(v_1, \ldots, v_n)$ and
  - A terminates for $v_1, \ldots, v_n$ with return value r, then
  - $Post(v_1, \ldots, v_n, r)$
- A is **totally correct** it is partially correct and terminates.

Finally we can recover Def. 3.1 by saying that A is a correct algorithm for a function f it is totally correct with

- precondition: nothing (always true)
- postcondition: $r == f(x_1, \ldots, x_n)$

The reason for splitting correctness up is that partial correctness and termination are often proved separately in very different ways. So it is good to have separate definitions for them. Sect. 3.1.2 and 3.1.3 describe the most important techniques.

The reason for splitting the specification into pre- and postcondition is to make fine-granular statements about what input an algorithm expects and what output it provides. They can be seen as a trade between the programmer W who writes function F and the programmer C who calls F. The precondition is the price that C has to pay (by making sure the precondition holds before calling F). And the postcondition in the service that W provides in exchange (by returning a value that satisfies the postcondition).

In particular, W may assume that the precondition holds—she does not have to check it. Instead, C has to check it. Vice versa, C may assume that the postcondition holds afterwards.

Example 3.7 (Pre/Postcondition). Consider a variant $gcd32(x : int, y : int) : int$ of the Euclidean algorithm that uses 32-bit integers. This can never be correct because it cannot handle arbitrarily large natural numbers. Moreover, the input and output type now allow negative values, which we want to exclude.

So we could use the following:

- precondition: $Pre(x, y) = 0 \leq x \leq MaxInt \wedge 0 \leq y \leq MaxInt$
- postcondition: $Post(x, y, r) = 0 \leq r \leq MaxInt \wedge r == \gcd(x, y)$

where $MaxInt$ is the maximal value of the type int.

Note that this specification makes the strong requirement that there will be no overflows. That works out for the Euclidean algorithm because all its intermediate results are smaller than the input.

For other algorithms, like a 32-bit algorithm $fib32(n : int) : int$ for Fibonacci numbers, the input has to be much smaller than $MaxInt$ to make sure the output fits into a 32-bit integer. So we might use:

- precondition: $Pre(n) = 0 \leq x \leq 46$
- postcondition: $Post(n, r) = r == fib(n)$

3.1.2 Partial Correctness

Loop Invariants for while-Loops

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 partial 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** for this loop if F is preserved by the loop: if F holds before executing *code*, it also holds 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 it as follows:

Theorem 3.9. *Consider a loop* **while** $C(\vec{x})$ $\{code\}$ *with 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 $\neg C(\vec{x}) \wedge F(\vec{x})$ *holds if and when the while-loop has been executed.*[2]

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 every iteration of *code*, it also holds after executing the loop. □

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

Example 3.10 (Euclidean Algorithm). We prove partial correctness of 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:

- We show that F holds before the loop.
  Before reaching the loop, we have $x == m$ and $y == n$. Thus, immediately $gcd(m, n) == gcd(x, y)$.
- We show that F is preserved by the loop.
  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 $code(m, n, x, y) = (m', n', x', y')$.

[2] We assume here that the evaluation of $C(\vec{x})$ has no side-effects and thus may not change the values of the x_i. In most programming languages, that would be allowed, but is a very bad practice precisely because it makes loop-invariant arguments more complicated.

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 $\gcd(x, y) = \gcd(x, y - x)$. 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 (i).

Now we can continue. The next statement is **return** x. Using Thm. 3.9, we obtain that $\neg C(m, n, x, y) \wedge F(m, n, x, y)$ holds, i.e., $\neg x \neq y \wedge \gcd(m, n) == \gcd(x, y)$. That yields $x == y$ and therefore $\gcd(m, n) == \gcd(x, x) == x$. Thus, the returned value is indeed $\gcd(m, n)$.
To prove total correctness, we still have to show that the while-loop terminates, which we do in Ex. 3.14

Induction for Recursive Functions

Proving partial correctness of recursive functions is very easy because we can simply use the postcondition about the recursive call. Formally, this means we do an induction proof on the number of recursive calls.

Example 3.11 (Recursive Euclidean Algorithm). We prove partial correctness for the algorithm $gcdRec(m : \mathbb{N}, n : \mathbb{N})$ from Ex. 2.15.
We have to prove the postcondition $gcdRec(m, n) == \gcd(m, n)$ where r is the return value. We proceed by induction, i.e., we assume that the property holds for all recursive calls. Then we have to handle two cases for the two branches of the if-statement:
- $n == 0$: Then $gcdRec(m, n) = m$, and the postcondition follows from $\gcd(m, 0) == m$.
- $n \neq 0$: Then, by using the induction hypothesis, $gcdRec(n, m \bmod n) == \gcd(n, m \bmod n)$. Then the postcondition follows from $\gcd(m, n) == \gcd(n, m \bmod n)$.

To prove total correctness, we still have to show that the recursion terminates which we in Ex. 3.18.

3.1.3 Termination

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. One of the most important results of theoretical computer science is that the halting function is not computable, i.e., there is no algorithm for it.
Thus, even if do not care what our algorithm actually does and only want to know if it terminates at all, all we can do is prove it manually for each input.
Termination is trivial for assignment, for-loop[3], 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.

Termination Orderings for While-Loops

Definition 3.12 (Termination Ordering)**.** Consider a while-loop of the form **while** $C(\vec{x})$ $\{code\}$.
A **termination ordering** for it is a function $T(\vec{x}) \in \mathbb{N}$ such that for all $\vec{v}$ we have that

$$C(\vec{v}) \quad \text{implies} \quad T(\vec{v}) > T(code(\vec{v})).$$

The intuition behind a termination ordering is that $T(\vec{x})$ strictly decreases in every iteration of the loop. Because it cannot decrease indefinitely, there can only be finitely many iterations, i.e., the loop must terminate. The following theorem makes that precise:

[3] In some programming languages, it is possible to write non-terminating for-loops by explicitly assigning to the counter variable in the body of the loop. That is a very bad practice precisely because it endangers termination.

Theorem 3.13 (Termination Ordering)**.** *Consider a the loop* **while** $C(\vec{x})$ $\{code\}$ *and a termination ordering* $T(\vec{x})$ *for it.*
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 i iterations of executing *code*:

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

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

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 3.14 (Euclidean 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 either x or y smaller. Therefore, it must make 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 $y \neq 0$.

Termination Orderings for Recursion

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

Definition 3.15 (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 with arguments $\vec{v}$ and recursively calls itself with arguments $\vec{v'}$, then $T(\vec{v}) > T(\vec{v'})$.

Then we can prove the corresponding theorem:

Definition 3.16 (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 terminate.

Theorem 3.17 (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 3.18 (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 is trivial either way: Under the assumption that the recursive call returns, the function consists only of if-statements and therefore terminates.
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.

3.1.4 Implementing Loop Invariants and Termination Orderings

Loop invariants and termination orderings an be tricky to understand for beginners. Therefore, the following gives a more concrete explanation.
Consider an arbitrary algorithm that uses a while-loop, e.g.,

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

To exemplify the role of a termination ordering, we modify it as follows:

```
fun T(n : ℕ, product : ℕ, factor : ℕ) : ℕ =
    ???

fun fact(n : ℕ) : ℕ =
    product := 1
    factor := 1
    print T(n, product, factor)
```

```
    while factor ≤ n
        product := product · factor
        factor := factor + 1
        print T(n, product, factor)
    return product
```

Our goal is to implement T such that running the algorithm prints strictly decreasing natural numbers. Any such implementation of T is a termination ordering and proves that the while loop terminates.

To exemplify the role of a loop invariant, we modify the algorithm in a very similar way:

```
fun F(n : ℕ, product : ℕ, factor : ℕ) : bool =
    ???

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

Our goal is to implement F such that running the algorithm prints only *true*. In that case,

- F is true before the loop
- F is a loop invariant, i.e., if it is true before, it is also true after executing the body of the loop.

Thus, if the while-loop should terminate, afterwards F must be true and the condition of the loop must be false.

There are many possible ways to implement F—already **return** *true* trivially satisfies the requirements. A practically useful implementation of F should tell us something that helps establish the postcondition (which in this case is $fact(n) == n!$).

3.2 Efficiency

An algorithm is efficient if it can be run with low cost. *Complexity* measures that cost.[4] Thus, an efficient algorithm has low complexity and vice versa.

There are two kinds of complexity: *time* and *space* complexity. Time complexity measures how long it takes for an algorithm to terminate. Space complexity measures how much temporary memory is needed along the way. Without qualification, the word *complexity* usually but not always means *time complexity*

In this section, we focus on time complexity. While termination describes whether an algorithm A terminates at all, its time complexity describes how long it takes to terminate. The time 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.

3.2.1 Exact Complexity

Exact complexity is tricky because the number of steps and the sizes of inputs depend on the programming language and the physical machine that is used. For example, we might try to use the following definitions for a simple programming language:

Example 3.19 (Counting Steps Exactly). For a typical programming language implemented on a digital machine, the following definition is roughly right:

[4] At Jacobs University, complexity is discussed in detail in a special course in the 2*nd* year.

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
- $\text{Steps}(\textbf{while } C \{B\}) = (n+1) \cdot \text{Steps}(C) + n \cdot \text{Steps}(B)$ 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: $\text{Steps}(x) = 1$
- Applying built-in operators O such as + or &&: $\text{Steps}(O(E_1, \ldots, E_n) = \text{Steps}(E_1) + \ldots + \text{Steps}(E_n) + 1$
  - $\text{Steps}(E_i)$ steps to evaluate the arguments
  - 1 step to apply the operator
- Calling a function: $\text{Steps}(f(E_1, \ldots, E_n)) = \text{Steps}(E_1) + \ldots + \text{Steps}(E_n) + 1 + n$
  - $\text{Steps}(E_i)$ 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.

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 $\text{Steps}(x := E) = \text{Steps}(E) + 1$ or $\text{Steps}(x := E) = \text{Steps}(E) + 2$. It just means that every program takes a little longer. It would matter more if $\text{Steps}(x := E) = 2 \cdot \text{Steps}(E) + 1$, 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 Bachmann-Landau notations.[5] 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 3.20 (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

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

If $f \lessapprox g$, we say that f is **asymptotically smaller** than g.
We write $f \eqslantless g$ if $f \lessapprox g$ and $g \lessapprox f$.
Moreover, for a function $g : \mathbb{N} \to \mathbb{R}^+$, we define the following sets of functions

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

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

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

Intuitively, $f \lessapprox g$ 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 3.21 (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: $f \lessapprox g$ iff $\exists a > 0. \exists b > 0. \forall n. f(n) \leq a \cdot g(n) + b$.
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.

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

Example 3.22 (Complexity Classes). 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 when we go to infinity. 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 table introduces a few more classes and arranges them by increasing size:

$O(1)$	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 example $\log_c n$ is sub-linear, and n^2 is super-linear.

The following theorem collects the basic properties of asymptotic notation:

Theorem 3.23 (Asymptotic 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 3.24. 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 3.25. Using Not. 3.24, 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 3.26 (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 confusion.

3.2.3 Asymptotic Complexity

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 3.27. Consider the algorithm from 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.[6]
   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. 3.2.1 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 additional 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 to compute the average.

– 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 3.28 (Euclidean 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.[7] 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, $C(n) \in \Theta(\log n)$.

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, $f \lneq g$ only means that f is smaller than g for *sufficiently large* input. Thus, $f \lneq g$ 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 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 can 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.

[7] Again we assume that all arithmetic operations take constant time.

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 (on any physical machine we might be able to build). 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. P is the class of all problems that have polynomial complexity, and NP is a related class that contains P. It is generally assumed that NP is strictly larger than P. But to prove that, one has to show that there is no polynomial algorithm for some problem in NP.

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 Sect. 3.2.1 have to be considered because they do not get rounded away. These subtleties can usually not make the implementation less complex than the algorithm, but they may make 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 underlying CPU, e.g., 64-bit numbers. If we implement the data structures for numbers correctly (i.e., for arbitrarily large numbers), the complexity of the arithmetic operations will be greater.

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 only 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 are super-linear. That is more than $O(1)$ but still small enough to often be inessential.

With an efficient implementation of $\mathbb{N}$ and its arithmetic operations, the implementation of gcd, which uses $\Theta(\log_2 n)$ steps and applies 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 $O(1)$. So the whole loop takes $\Theta(n)$ 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 $\Theta(n)$ 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 usually be much slower than building the list in memory, no matter whether 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 input.

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.

Maintainability An often-underestimated goal being able to maintain a program. Software usually lives for years, often decades, and programmers will come and go during its life time. One of the biggest sources of problems can be unclear or undocumented code—even if it is well-designed, correct, and efficient.

Simple data structures and elegant algorithms that are derived systematically from the specification help here. It leads to implementations that are easier to understand, which allows new programmers to take over seamlessly.

Minor optimizations should generally be avoided because they make the implementation less maintainable. Even major optimizations (e.g., linear instead of quadratic) must be weighed against the danger of introducing bugs in the long run.

Chapter 4

Arithmetic Examples

4.1 Exponentiation

4.1.1 Specification

The function $power(x \in \mathbb{Z}, n \in \mathbb{N}) \in \mathbb{N}$ (also written as x^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 n == 0
        1
    else
        x · power(x, n − 1)
```

Correctness The correctness of this algorithm is immediate because it follows the specification literally. For example, $T(x, n) = n$ is already a termination ordering.

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 n ≤ 1
        n
    else
        fib(n − 1) + fib(n − 2)
```

Correctness The correctness of this algorithm is immediate because it follows the specification literally. For example, $T(n) = n$ is a termination ordering.

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 n ≤ 1
        n
    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 can use

$$F(n, prev, current, i) = prev == fib(i-1) \wedge current == fib(i)$$

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 can use $T(n, prev, current, i) = n - i$. 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. While termination is trivial, partial correctness does not hold. We know that the formula $fib(n) = round(\varphi^n/\sqrt{5})$ is true, but that has no immediate use for 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

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

We can now pick any algorithm for computing the n-power of a matrix, e.g., by using square-and-multiply from Sect. 4.1.3 for matrices.

Correctness Correctness follows from the correctness of square-and-multiply.

Complexity Square-and-multiply has complexity $O(\log n)$. 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: $x.columns == y.rows$
  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 Multiplication 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 to 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:

$$M_1 = a(f - h)$$

$$M_2 = (a+b)h$$
$$M_3 = (c+d)e$$
$$M_4 = d(g-e)$$
$$M_5 = (a+d)(e+h)$$
$$M_6 = (b-d)(g+h)$$
$$M_7 = (a-c)(e+f)$$
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} M_5 + M_4 + M_2 + M_6 & M_1 + M_2 \\ M_3 + M_4 & M_1 + M_5 - M_3 - M_7 \end{pmatrix}$$

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 $n^{2.81\cdots}$, 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 $\Theta(n^{2.37\cdots})$. 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.
Because A^* is a set for an arbitrary set A, data structures for lists must be polymorphic with a type parameter A.

Immutable Lists The following table specifies the most important functions involving lists:

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

Most of them are polymorphic. *map* and $fold$ even take a second type parameter for the return type of the function.
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.

Mutable Lists The following table specifies the most important functions on mutable lists that differ from immutable 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 $n < l$			
$delete[A](x \in A^*, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a_{n+1}, \ldots, a_{l-1}]$	
$insert[A](x \in A^*, a \in A, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a, a_n, a_{n+1}, \ldots, a_{l-1}]$	
$update[A](x \in A^*, a \in A, n \in \mathbb{N})$	nothing	$x := [a_0, \ldots, a_{n-1}, a, a_{n+1}, \ldots, a_{l-1}]$	$x_n := a$ or $x[n] := a$

The other functions such as *length* and *get* are not affected.

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. 5.2 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. We will write $List[A]$ whenever we mean an arbitrary data structure for lists.

5.2.1 Immutable Lists

For immutable lists, functions like *delete*, *insert*, and *update* (see Sect. A.5.4) always return new lists. That requires copying (parts of) the old list, which takes more time and memory.

Without further qualification, this is usually what $List[A]$ refers to.

Functional Style: Lists as an Inductive Type

Functional languages usually implement lists an in inductive data type:

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

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:

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

Object-Oriented Style: Linked Lists

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

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

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

A basic realization looks as follows:

abstract class $IndList[A]()$

class $nil[A]()$ **extends** $IndList[A]()$

class $cons[A](head : A, tail : List[A])$ **extends** $IndList[A]()$

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

Moreover, we have to override equality so that, e.g., two instances of *cons* are equal iff they used equal constructor arguments.

Complexity

Complexity of lists is measured in the lenght n of the list.
Most operations on lists are linear because the algorithm must traverse the whole list. For example, the straightforward implementation of *length* takes $\Theta(n)$.
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 $\Theta(n)$.
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 $\Theta(n)$ time for the traversal *and* $\Theta(n)$ memory for the result list.
In the case of $map(x, f)$ and $fold(x, a, f)$, the complexity depends on the passed function f. However, in the typical case where the run time of f does not depend on the length of the list, we can assume it takes constant time c. Thus, the overall run time is $\Theta(cn) = \Theta(n)$.
However, there is one important exception: *prepend* takes $\Theta(1)$. This is because we can implement $prepend(a, x)$ simply by calling $cons(a, x)$. Correspondingly, removing the first element takes $\Theta(1)$.

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, thus allowing for many operations with $\Theta(1)$ time or memory complexity.
Because we can update the list in place, it becomes critical for efficiency how exactly the list is stored in memory. Three cases are of great importance, all with advantages and disadvantages:

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

Arrays

The data structure $Array[A]$ stores all elements in a row in memory. Arrays must be a primitive feature of the programming language and are so in most languages.
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 $\Theta(1)$. $get(x, n)$ 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 $\Theta(n)$. For example, we can implement deleting by:

```
fun delete(x : Array[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 in advance 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. This may waste memory if n is chosen too large. But arrays are unbeatable in the common situation where we know that we will never call *insert* anyway.

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. We can implement that using classes (or similar primitives like structs in C):

```
class LinkedList[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 trick where the built-in value *null* is used:

- If *head* is null, we have the empty list.
- If *next* is null, we have the last element of the list.

those cases.
Now the list $[1, 2, 5]$ is built as $x :=$ **new** $LinkedList($**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 : LinkedList[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 $\Theta(n)$ time for most operations. However, they still perform better because changes can be done in-place. Moreover, many operations can be done in $\Theta(1)$ memory whereas immutable lists often require $\Theta(n)$ memory.

An interesting exception is the following variant of *insert*: Instead of taking the position n at which to insert (which takes linear time to find), it take the element after which to insert:

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

A similar trick for deleting does not work so well: We can implement $delete(x : LinkedList[A], after : Elem[A])$ in $\Theta(1)$ if we know after which element to delete. But a function $delete(x : LinkedList[A], e : Elem[A])$ where e is to be deleted would be useful to delete—that still requires $\Theta(n)$ to find e in the linked 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.

```
class DoubleLinkedList[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
$S.next$	$null$

Operations on doubly-linked lists are usually in the same complexity class as the corresponding ones for singly-linked lists.

A doubly-linked list has more memory overhead and thus copying and update operations have more time overhead. But doubly-linked lists can be traversed efficiently in *both* directions. For example, processing the elements of a singly-linked list in reverse order requires two traversals: one to find the find element, one to process. The same operation on a doubly-linked list requires only one traversal. Both are $\Theta(n)$ though, but the latter may be twice as fast.

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

```
fun delete(x : DoubleLinkedList[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
```

The following table summarizes the complexity of some operations on arrays, linked lists and doubly-linked lists in terms of the length l:

	$length[A]$	$get[A]$	$update[A]$	$insert[A]$	$delete[A]$	$prepend[A]$	$append[A]$	$reverse[A]$
		at position n						
Array	$\Theta(1)$	$\Theta(1)$		$\Theta(l-n)$		$\Theta(l)$	$\Theta(1)$	$\Theta(l)$
Linked list	$\Theta(l)$	$\Theta(n)$		$\Theta(n)$		$\Theta(1)$	$\Theta(l)$	$\Theta(l)$
Doubly-linked List	$\Theta(l)$	$\Theta(n)$		$\Theta(n)$		$\Theta(1)$	$\Theta(1)$	$\Theta(l)$

Growable Arrays

Growable arrays are a compromise between arrays and linked lists. Initially, they behave like an array with a fixed length l. However, when inserting an element that increases the length beyond l, we create a second array of length l (elsewhere in memory) and connect the two. Thus, a growable array is a linked list of fixed-length arrays. The choice of l is up to the data structure designer, who may allow the programmer to tweak it.

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 l is constant, that yields $\Theta(n)$ 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
```

Here $MutableList$ is any of the mutable data structures from above.
It is easy to see that this function indeed has the effect of swapping two elements in x. For arrays, the time complexity of *swap* is $\Theta(1)$. For linked lists, it is $\Theta(n)$.

5.3.1 Bubblesort

Bubblesort 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 Insertionsort

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 straightforward 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 $T(x, i, current, pos) = pos$ for the while-loop.

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 Mergesort

Mergesort 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 (linear 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
        l := mergesort([x_0, ..., x_{k-1}])
        r := mergesort([x_k, ..., x_{n-1}])
        return merge(l, r)

fun merge(x : List[A], y : List[A]) : List[A] =
    xRest := x
    yRest := y
    res = []
    while nonempty(xRest) || nonempty(yRest)
        takefromX := empty(yRest) || (nonempty(xRest) && xRest.head ≤ yRest.head)
        if takefromX
            res := cons(xRest.head, res)
            xRest := xRest.tail
        else
            res := cons(yRest.head, res)
            yRest := yRest.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:
  - $Sorted(a)$ and $Sorted(b)$ follow from the postcondition of the recursive call.
  - Then the postcondition of *merge* yields $Sorted(merge(a, b))$.

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 $F(x, y, xRest, yRest, res)$ that states that

- $Sorted(reverse(res))$ and $Sorted(xRest)$ and $Sorted(yRest)$
- All elements in res are in $\leq$-relation to all elements in $xRest + yRest$.
- $res + xRest + yRest$ 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 $yRest + xRight$ and prepend it to res. In particular, because $xRest$ and $yRest$ are sorted, the smallest element must be $xRest.head$ or $yRest.head$.
- For the same reason, all elements of res remain smaller than the ones of $xRest$ and $yRest$.
- Because we only remove elements from $xRest$ and $yRest$, they remain sorted.
- Because every element that is removed from $xRest$ or $yRest$ 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, $xRest$ and $yRest$ 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 $T(x, y, xRest, yRest, res) = length(xRest) + length(yRest)$. 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 $n = length(x)$. We compute the time complexity $C(n)$:

- 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 $\Theta(length(a) + length(b)) = \Theta(n)$.

That yields

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

By choosing $k = \log_2 n$ 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, mergesort is quasilinear and thus strictly more efficient than bubblesort and insertionsort.

Contrary to bubblesort and insertionsort, mergesort 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 mergesort as a whole would also be quadratic.

5.3.4 Quicksort

Quicksort is similar to mergesort 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 a and b containing the elements smaller and greater than x respectively.
  No merging is necessary because all elements in a are smaller than all elements in b. Thus the sorted list is $quicksort(a) + x + quicksort(b)$.
- To divide the list, quicksort 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.

Remark 5.7 (Additional Arguments in a Recursion). 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 call 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 quicksort 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 mergesort, 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 quicksort 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 quicksort is less attractive than mergesort 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 quicksort is superior to mergesort. Moreover, quicksort 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, quicksort can become substantially faster than mergesort.

5.3.5 Other Algorithms

There is a number of other sorting algorithms that we will not go into here. Examples include counting sort, radix sort, and bucket sort.

One particularly important sorting algorithm is heap sort, which we discuss in Sect. 9.6.4 (after introducing heaps).

5.3.6 In Programming Languages

Most programming languages come with a standard library that includes efficient sorting algorithms. Moreover, other libraries for other algorithms may be around. In some cases, languages only specify the interface and leave the implementation (and thus the choice of algorithm)to individual implementations of compilers/interpreters.

The following gives some examples.

Python uses Timsort (named after the programmer), which is a hybrid of mergesort and insertionsort with various optimizations. It is written directly in C.

Java used to use just quicksort. Java 7 uses either Timsort (ported to Java) or a variant of quicksort that uses two pivot elements.

Scala defers to Java's implementation.

C++'s std library specification does not prescribe a sorting algorithm but requires $O(n \log_2 n)$ worst-case complexity (average-case in earlier versions). Implementations vary in their choice of algorithm, e.g., using hybrid algorithms that perform some iterations of quicksort before switching to insertionsort for the resulting small lists.

For Javascript, the choice is up to the browser (because every browser is a separate implementation of Javascript).

Part II

Important Data Structures

Chapter 6

Finite Data Structures

6.1 Void

The set *void* contains no elements.

Not surprisingly, it is rarely used. However, it is nice to have when dealing with operations that do not return. For example, we say that throwing an exception or terminating the program returns an element of *void*.

Most programs do not need the type void. And most programming language either do not have it or only have it under the hood.

6.2 Unit

The set *unit* contains exactly one element, which we write ().

It is *unit* is rarely used because if we know that $x \in unit$, we already know the value of x. Thus, having a value of type *unit* gives us no information.

However, *unit* is nice to have when dealing with operations that do return, but do not return a value. In that case, we say that the operation returns type *unit*.

For example, assignments, loops, 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 $insert(x \in A^*, a \in A, n \in \mathbb{N}) \in unit$.

Functional programming languages usually have a built-in type *unit*. That way, in a functional programming langauge, every operation has a return type.

6.3 Booleans

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

Most programming languages have a built-in type *bool*, which is the result type of the equality operator.

6.4 Integers Modulo

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

Most programming languages do not offer $\mathbb{Z}_m$ for every m. Usually, they offer at most $\mathbb{Z}_{2^k}$ for $k = 8$ (usally called *byte*), $k = 16$ (*word*), $k = 32$ (*integer*), and/or $k = 64$ (*long*).

Note that, depending on the programming language, the built-in type *int* may refer to one of those (usually for $k = 32$) or to $\mathbb{Z}$.

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 $\mathbb{Z}_m$. [1]

[1] Note that some programming languages implement div and mod in unexpected ways for negative arguments.

6.5 Enumerations

For fresh names $l_1, \ldots, l_n$, the set $enum\{l_1, \ldots, l_n\}$ has exactly n elements, which are 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. This is because defining an enumeration set introduces new values, namely the l_i.

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

7.1 Countable Sets

The sets $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ are well-known from mathematics.

Working with $\mathbb{Z}$ (as opposed to $\mathbb{Z}_m$ for some m) is called *arbitrary precision arithmetic*. $\mathbb{Z}$ may or may not be the built-in type *int*—that depends on the programming language. If not, *int* is $\mathbb{Z}_m$ for some m—in those languages, there is usually a library that defines $\mathbb{Z}$.

A data structure for $\mathbb{Q}$ can be defined by using pairs of integers.

We usually do not use a special data structure for $\mathbb{N}$ and instead just use the positive values of $\mathbb{Z}$. Alternatively, we can give a (very inefficient) definition of $\mathbb{N}$ as an inductive type as in Ex. 2.15.

7.2 Uncountable Sets

We cannot implement data structures for $\mathbb{R}$ and $\mathbb{C}$ because they are uncountable.

There are some approximate solutions to work with $\mathbb{R}$. For example, we can simply represent a real number r as a function $\mathbb{N} \to \{0, \ldots, 9\}$ that provides the infinite decimal expansion of r. Because we can only represent countably many functions as effective objects, not all real numbers can be represented like. However, all practically useful ones can. A major drawback of this representation is that we cannot give an algorithm for equality (because we would have to check that two functions are equal for infinitely many arguments), thus crippling the data structure.

For the $\mathbb{C}$, it is often sufficient to work with the countable set $\mathbb{Q} + \mathbb{Q}i$, which is the set of complex numbers whose real and imaginary parts are rational.

Chapter 8

Option-Like Data Structures

8.1 Specification

$A^?$ is a set containing all the elements of A and one additional element $\perp$.
$A^?$ is used to represent an optional value of A. The element $\perp$ is used to represent an undefined/absent value.
Options are usually immutable. The main operations on $A^?$ are

function	returns	effect
$getOrElse(x \in A^?, default : A) \in A$	get the optional value or a default value	none
$get(x \in A^?) \in A$	get the optional value	error if absent
$map(x \in A^?, f \in A \to B) \in B^?$	apply f to the optional value	none

8.2 Data Structures

8.2.1 Using Inductive Types

In functional programming languages, a data structure for optional values can be realized as an inductive type:

data $Option[A] = Some(value : A) \mid None$

Such a definition (except for possibly using different names) is usually part of the standard library of the language.

8.2.2 Using Pointers

In languages that use pointers, we can represent $Option[A]$ as the type $Pointer[A]$ (written $A*$ in C) of pointers to elements of A. In that case, the *null* pointer represents $\perp$.
Object-oriented languages do not necessarily expose pointers to the programmers (e.g., C++ does, but Java does not). However, even then they use pointers internally, and any class-type provides the value *null*. In this situation it is impossible to represent the set A correctly as a class—any class for A is automatically a data structure for $A^?$. This often causes ambiguous specifications and subtle errors. Therefore, it is good practice to never use *null* even when possible.

Chapter 9

List-Like Data Structures

The specification and several data structures for mutable and immutable lists are already discussed in Sect. 5.1. Here we only discuss some additional data structures for the set A^*.

9.1 Stacks

$Stack[A]$ is a data structure for the set A^*.

$Stack[A]$ is very similar to $List[A]$. The difference is that $Stack[A]$ provides *less* functionality. While $List[A]$ is a general-purpose list, $Stack[A]$ is custom-fitted to one specific, very common use case. By requiring fewer operations, they allow more optimized implementations.

Stacks can be mutable or (less commonly) immutable. Here we will use the mutable variant. The functions for mutable stacks are:

function	returns	effect
$push(x \in A^*, a \in A) \in unit$	nothing	prepend a to x
$pop(x \in A^*) \in A^?$	the first element of x (if any)	remove the first element of x
$top(x \in A^*) \in A^?$	the first element of x (if any)	none

The intuition behind stacks is that they provide a LIFO store of data. LIFO means last-in-first-out because every *pop* returns the most recently pushed value. This is exactly the behavior of a literal stack of items: We can put an item on top of a stack (*push*), remove an item from the stack (*pop*), or check what item is on top (*top*). We cannot easily see or remove the other items.

Very often, the LIFO behavior is exactly what is needed. For example, when we solve a maze, we can push every decision we make. When we hit a dead end, we trace back our steps—for that, we have to pop the most recent decision, and so on.

9.2 Queues

Queues are very similar to stacks. Everything about stacks also applies to queues except for the following.

The functions for mutable queues are:

function	returns	effect
$enqueue(x \in A^*, a \in A) \in unit$	nothing	append x to A
$dequeue(x \in A^*) \in A^?$	the first element of x	remove the first element of x
$empty(x \in A^*) \in bool$	true if x is empty	none

The intuition behind queues is that they provide a FIFO store of data. FIFO means first-in-first-out because every *dequeue* returns the least recently enqueued value. This is exactly the behavior of a literal queue of people: Every newcomer has to queue up at the end of the queue (*enqueue*), and every time a server is ready the first in line gets served (*dequeue*). Newcomers cannot cut in line, and the server cannot easily see who else is waiting.

Very often, the FIFO behavior is exactly what is needed. For example, when we have a list of tasks that need to be done. Every time we create a new task, we enqueue it, and whenever we have time we dequeue the next task.

Queues are often used when components exchange messages or commands. In that case, some components—called the producers—only call enqueue, and other components—called the consumers—only call dequeue. For example, the producers can be different programs, A is the type of print jobs, and the consumers are different printers.

More complex queue data structures may also for dequeueing based on priority (see also Sect. 9.6.3).

9.3 Buffers

Buffers are conceptually very similar to queues. But $Buffer[A]$ is usually optimized for enqueueing and dequeueing many elements of A at once. Therefore, while stacks and queues can be implemented well using linked lists, buffers usually use arrays to be faster.

A typical $Buffer[A]$ consists of three components:

- an $Array[A]$ b
- two integers *begin* and *end* indicating the first and last valid entry in the array.

Enqueueing writes to $b[end + 1]$ and increments *end*. Dequeueing reads from $b[begin]$ and increments *begin*.

A buffer overflow occurs when incrementing *begin*

For example, when a browser receives a web page, its network component loads the page into a $Buffer[char]$. In parallel, its HTML parser component starts processing the partially received page. That way the HTML page can be displayed partially already before it is fully loaded.

Buffers are almost always used automatically when a program is writing to a file. In that case, a $Buffer[int]$ or $Buffer[char]$ is used that holds the data written to the file. The write command does not actually write data to the file directly—it only enqueues it in the buffer. That is advantageous because enqueueing to a buffer in memory is much faster than writing to the hard drive. While the program is already moving on, the programming language libraries or the operating system work in the background to periodically dequeue and write all characters to the file.

When working with files, an important operation is *flushing* the buffer. This forces the immediate processing of all data in the buffer. Flushing happens automatically at the latest when the program terminates. However, occasionally manual flushing is necessary:

- When a program terminates with an error, buffers have to be flushed to avoid losing data.
- When a program writes log data to a file that the programmer wants to read immediately, it is important to flush regularly to make sure the programmer reads updated information.

9.4 Iterators

9.4.1 Specification

$Iterator[A]$ is a data structure for the set A^*.

Iterators are usually mutable. Their functionality is even more restricted than the one of stacks and queues:

function	returns	effect
$getNext(x \in A^*) \in A$	the first element of x	remove the first element of x
$hasNext(x \in A^*) \in bool$	*true* if x is not empty	none

The typical way to use an iterator $i \in Iterator[A]$ is the following:

```
while hasNext(i)
    a := getNext(i)
    do something with a here
```

This is called **traversing** the iterator. Afterwards the iterator is traversed and cannot be used again.

$Iterator[A]$ may look somewhat boring. In order to understand the value of iterator, we have to make one definition: A data structure $D[A]$ is called **iterable** if there is a function

$$iterator(x \in D[A]) \in Iterator[A]$$

Now the imoprtance of iterators follows from two facts:

- Many data structures D are iterable (see Sect. 9.4.4).
- Many important operations for D can be realized using only the functionality of iterators (see Sect. 9.4.3).

Thus, iterators provide a sweet-spot in the trade-off between simplicity and expressivity—they are very simple, but we can do a lot with them.

> *Remark* 9.1 (Simplicity vs. Expressivity). The trade-offs between simplicity and expressivity comes up again and again in computer science. The best data structures combine both properties, but usually they are mutually exclusive.
> All the important data structures presented in Part II have become important because they do well in this way.

An important function on iterators is *map*:

function	returns
$foreach(x \in Iterator[A], f \in A \to B) \in Iterator[B]$	an iterator for $[f(a_1), \ldots, f(a_n)]$ where $X = [a_1, \ldots, a_n]$

The trick behind *map* is that x is not traversed right away. Instead, we create a new iterator that, when traversed, applies f. That way we ensure that f is applied only as often as necessary.

9.4.2 Data Structure

We can give a data structure for iterators as an abstract class:

```
abstract class Iterator[A]()
  fun hasNext() : bool =
                                          precondition for getNext is hasNext == true
  fun getNext() : A =
```

Then we can give an algorithm for *map* as follows:

```
class Map[A, B](x : Iterator[A], f : A → B) extends Iterator[B]
  fun hasNext() : bool = {x.hasNext}
  fun getNext() : B = {f(x.getNext)}

fun map(x : Iterator[A], f : A → B) = {new Map[A, B](x, f)}
```

9.4.3 Working with Iterable Data Structures

Let us assume an iterable data structure $D[A]$. Our goal is to define functions on $x \in D[A]$ that use only $iterator(x)$. There are indeed many of those. Some important ones are:

function			returns
		below, let $X = iterator(x)$	
$length$	$(x \in D[A])$	$\in \mathbb{N}$	numbers of elements in X
$contains$	$(x \in D[A],\ a \in A)$	$\in bool$	*true* if a occurs in X
$index$	$(x \in D[A],\ a \in A)$	$\in \mathbb{N}^?$	the position of the first occurrence of a in X (if any)
$find$	$(x \in D[A],\ p \in A \to bool)$	$\in A^?$	the first element a in X (if any) such that $p(a)$ is *true*
$count$	$(x \in D[A],\ p \in A \to bool)$	$\in \mathbb{N}$	the number of elements a in X for which $p(a)$ is *true*
$forall$	$(x \in D[A],\ p \in A \to bool)$	$\in bool$	*true* if $p(a)$ is *true* for every element a in X
$exists$	$(x \in D[A],\ p \in A \to bool)$	$\in bool$	*true* if $p(a)$ is *true* for some element a in X
$results$	$(x \in D[A],\ f \in A \to B)$	$\in List[B]$	the list of results from applying f to all a in X
$fold$	$(x \in D[A],\ b \in B, f \in A \times B \to B)$	$\in B$	$f(a_1, f(a_2, \ldots, f(a_n, b))\ldots)$ with $X = [a_1, \ldots, a_n]$

All of the above functions should not have a side-effect. However, some of them take other functions as arguments. It is usually a bad to do so, but it is technically possible that these functions have side-effects. There is only one exception where we explicitly allow f to have a side-effect:

function	returns	effect
$foreach(x \in D[A], f \in A \to unit) \in unit$	nothing	apply f to all a in X

9.4.4 Making Data Structures Iterable

Many important data structures are naturally iterable.
That includes in particular all data structures for lists:

```
class ListIterator[A](l : List[A]) extends Iterator[A]
    index := 0
    fun hasNext() : bool = {index < length(l)}
    fun getNext() : A =
        a := get(l, index)
        index := index + 1
        a

fun iterator(l : List[A]) : Iterator[A] = {new ListIterator(l)}
```

9.5 Streams

$Stream[A]$ is not a data structure for the set A^*. Instead, it is a data structure for the set $A^{\mathbb{N}}$.
The set $A^{\mathbb{N}}$ contains functions $f :\to A$, which we can think of as inifite lists $[f(0), f(1), \ldots]$. Because they are so similar to lists, they are usually treated together with lists, even they do not realize the same set.
The set $A^{\mathbb{N}}$ is uncountable. Therefore, not all possible streams are effective objects that can be represented in a physical machine. However, for many practical purposes, it is fine to treat $Stream[A]$ as if it were the type of all possible streams.
$Stream[A]$ is usually implemented in the same way as $Iterator[A]$ with the understanding that $hasNext$ is always $true$, i.e., the stream is never over.
Consequently, the functions on $Iterator[A]$ behave slightly differently when used for $Stream[A]$. For exapmle:

- We cannot call $length$, $count$, $results$, $fold$, and $foreach$ on streams.
- We can call $contains$ on a stream. However, the function may run forever if the searched-for element is not in the stream. The same caveat applies to $index$, $find$, $forall$, and $exists$.
- We can call map. But it must be a special variant of map that returns a new iterator without actually applying the map-function.

9.6 Heaps

Heaps are formally defined in Sect. 10.1.3.
$Heap[A, O]$ is not a data structure for the set A^*. Instead, it is a data structure for the subset of A^* containing only lists sorted according to O. Therefore, heaps are very useful for sorting and prioritizing. We discuss applications of heaps to lists in Sect. 9.6.3 and 9.6.4.
First we introduce some basic operations on heaps in Sect. 9.6.1.

9.6.1 Operations on Heaps

Because heaps are mostly used for efficiency, they are usually mutable. The main operations on a heap are similar to those on a stack:

function	returns	effect
$insert(x \in Heap[A,O], a \in A) \in unit$	nothing	add a to x in any position
$extract(x \in Heap[A,O]) \in A^?$	the O-smallest element of x (if any)	remove that element from x
$find(x \in Heap[A,O]) \in A^?$	the O-smallest element of x (if any)	none

insert, *extract*, and $find$ for heaps correspond exactly to *push*, *pop*, and *top* for stacks. The crucial different is that $insert(x, a)$ does not prepend a to x—instead, it is unspecified where and how x is added. *extract* and $find$ do not return the most recently added element—instead, they return the smallest element with respect to O.

It is unspecified what exactly a heap looks like and where and how *insert* actually performs the insertion. That way heaps have a lot of freedom to organize the data in an efficient way. That freedom is exploited to make the operations *extract* and $find$ fast.

Because $Heap[A, O]$ is underspecified, there are many different options how to implement heaps. In practice, there are dozens of competing variants using different efficiency trade-offs. A critical property is that all operations take only $O(\log n)$ where n is the number of elements in the heap.

9.6.2 A Heap Implementation

For a straightforward implementation of $Heap[A, O]$, we use a binary heap H, i.e., a binary tree over A that is also a heap.

Let n be the number of nodes in H and h be the height of h. All operations are such that H remains almost-perfect: for every depth $d < h$ there are maximally many nodes, i.e., 2^d nodes. (At depth h, we have to allow for fewer than 2^h nodes because not every n there is a perfect heap.) That way, we always have $h \leq \log_2 n$, and all branches have length h or $h - 1$, i.e., $O(\log_2 n)$.

$find$ is trivial: We return the root of H. That takes $O(1)$.

$insert(H, x)$ inserts x into one of the branches with minimal length. The insertion occurs at the position that keeps the branch sorted. Because it was sorted already, that requires $O(l)$ operations, where l is the length of the branch, i.e., $O(\log_2 n)$.

extract removes the root of H and returns it. That takes $O(1)$. Additionally, we have to repair the heap property. To do that, we take some leaf l of H and put it at the root. Now have a near-perfect binary tree again, but it is not a heap yet: l is too big to be the root. Therefore, we push l down by iteratively swapping it with its smallest child until we have a heap. Finding a leaf and pushing along some branch takes $O(\log_2 n)$.

9.6.3 Priority Queues

A $PriorityQueue[A]$ behaves like a $Queue[A]$ except that dequeueing returns the element with the highest priority. This is achieved by using a data strucutre for $Heap[A, O]$ where O orders elements by decreasing priority. Then *insert* and *extract* correspond to *enqueue* and *dequeue*.

9.6.4 Heapsort Algorithm

Heapsort is a sorting algorithm that runs in $\Theta(n \log n)$.

If $\leq$ is the total order for sorting, a simple heapsort is given by

```
fun heapsort(x : A*) : A* =
    h := new Heap[A, ≥]()

    left := x
    for i from 0 to length(x) − 1
        next := left.head
        insert(h, next)
        left := left.tail

    res := Nil
    for i from 0 to length(x) − 1
```

$next := extract(h)$
$res := prepend(next, res)$

This uses two loops using $length(n)$ iterations each. The first loop throws all elements of x into the heap; the second loop pulls them out again and builds the list res to be returned. Because $extract$ always returns the greatest element, the result automatically sorted.

If n is the length of the list, each $insert$ and $extract$ operation takes at most $\Theta(\log n)$. Thus, heapsort runs in $\Theta(n \log n)$.

There are much more optimized implementations of heapsort than the above example, possibly using optimized implementations of heaps. In particular, there are encodings of the heap structure in an array, which allows using heapsort as an in-place sorting algorithm. With those optimizations, heapsort is among the fastest sorting algorithms (but still takes $\Theta(n \log n)$).

Chapter 10

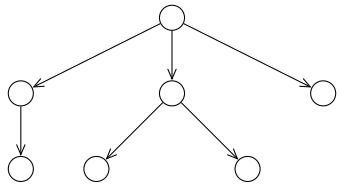
Tree-Like Data Structures

After lists, trees are the next most important data structure in computer science. They can be seen as a generalization of lists where the elements are not arranged in a row, but branching is allowed.

10.1 Specification

10.1.1 General Trees

There are many equivalent definitions. The easiest is by graphical example: A tree is something that looks like



A more formal definition is this:

Definition 10.1 (Tree)**.** A **tree** is a connected directed graph in which

- there is exactly one node (called the **root**) with in-degree 0,
- all other nodes have in-degree 1.

Here we already used the more general concept of graphs, which we define formally in Sect. 12.

Talking about the shape and parts of a tree can be confusing. Therefore, we introduce some vocabulary that helps us:

Definition 10.2 (Parts of a Tree)**.** For every edge from p to c, we call p the **parent** of c and n a **child** of p. Thus, the root has no parent; every non-root node has exactly one parent. A node may have any number of children. A node with 0 children is called a **leaf**. A node that is neither the root nor a child is called an **inner node**.

For every path from a to d, we call a an **ancestor** of d and d a **descendant** of a. Thus, all nodes are descendants of the root Every node is an ancestor/descendant of itself; a **proper** ancestor/descendant of n is an ancestor/descendant that is not n itself.

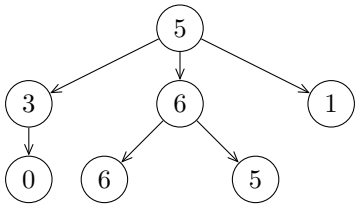
The number of proper ancestors of n is called the **depth** of n. Thus, the root has depth 0.

For a node n, the descendants of n form a tree again, which has root n. It is called the **subtree** at n.

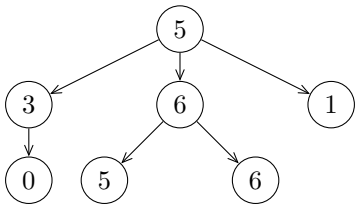
A path from the root to a leaf is called a **branch**. Thus, every leaf l is part of exactly one branch, whose length is the depth of l. The length of the longest branch(es) is called the **height** of the tree.

Remark 10.3. Contrary to all these tree metaphors, computer scientists prefer drawing trees with the root at the top and the leafs at the bottom.

Def. 10.1 only defines the abstract shape of trees. But trees are only useful if we can store some data in each node. For example, the following is a tree of integers:



Once we store data in a tree, we have to be a bit more careful: the order of children matters now. For example, the above tree of integers is different from the tree of integers below even both are based on the same tree.



Keeping track of the order makes the definition more complicated. The following definition is one out of several equivalent formal definitions:

Definition 10.4 (Trees over a Set)**.** The set $Tree[A]$ contains the **trees over the set** A. Such a tree over A consists of

- a set N (whose elements we call the **nodes**),
- a function $label : N \to A$ that maps nodes to elements of A ($label(n)$ is called the **label** of n, it is the data stored in each node),
- a function $children : N \to N^*$ that maps every node to its list of children,

such that N and $children$ form a tree.

10.1.2 Binary Trees

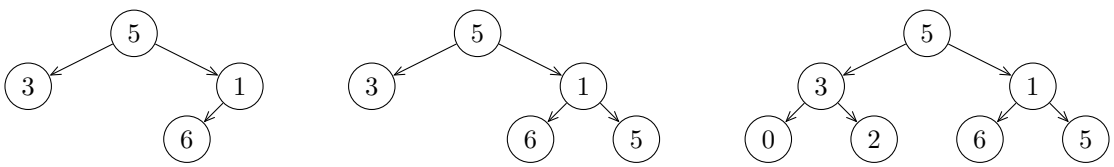
Binary trees are an important special case:

Definition 10.5 (Binary Tree)**.** A **binary tree** is a tree in which all nodes have at most 2 children. If a node has 2 children, the first and second child are called the **left** and **right** child, respectively.

Binary trees over a set are defined accordingly.

A binary tree is called **full** if all non-leaf nodes have exactly two children. A full binary tree is called **perfect** all leafs have the same depth.

For example, the following are, from left to right, a non-full, a full but not perfect, and a perfect binary tree of integers:



It is important to know the number of nodes in a binary tree:

Theorem 10.6. *A binary tree of height h has at most 2^n nodes at depth n. It has at most $2^{h+1}-1$ nodes in total. If it is perfect, it has exactly 2^n nodes at depth n and exactly $2^{h+1}-1$ nodes in total.*

Proof. Exercise. □

In particular, the number of nodes grows exponentially with the depth. Vice versa, we can organize n nodes as a binary tree of height $\log_2 n$. The latter property is often useful to obtain logarithmic implementations: if we organize n elements in a (nearly) perfect binary tree, we can reach any element in $\log_2 n$ steps.

10.1.3 Trees for Ordered Sets

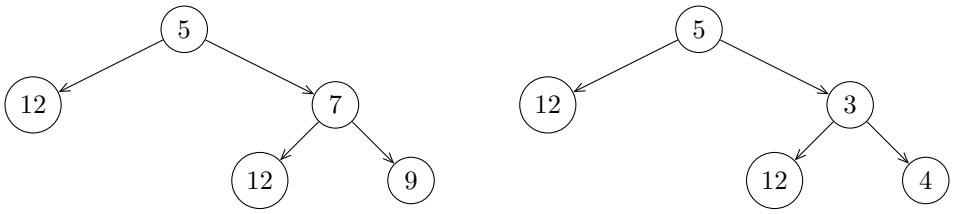
Special cases of trees are sometimes used to store sets of values for which we have a total order O.

Heaps

Assume a fixed total order O on A.

Definition 10.7 (Heap)**.** $Heap[A, O]$ is the subset of $Tree[A]$ containing only trees in which all branches are sorted with respect to O.

The elements of $Heap[\mathbb{Z}, \leq]$ are also called **min-heaps**. The elements of $Heap[\mathbb{Z}, \geq]$ are also called **max-heaps**.
The left tree below is a (binary) min-heap, the right one is neither a min-heap nor a max-heap:



In a heap, the every node is smaller than all its descendants. The root is always the smallest element in the heap. That makes heaps practical for sorting. Applications are presented in Sect. 9.6.

Binary Search Trees

Binary search trees are similar to heaps but the order property is different. In a heap every node is smaller than both its children. In a binary search tree, every node is greater than all its left descendants and smaller than all its right descendants.
They are discussed in Sect. 11.2.4.

10.1.4 Variants

Trees are simple enough to come up everywhere. But they are difficult enough to defy standardization. Contrary to, e.g., lists, the definition of tree can vary subtly across textbooks, programming language libraries, and computer scientists.
The following lists some details to watch out for when interacting with what someone else calls *trees*.

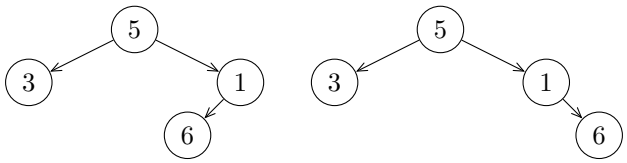
Rooted Trees Some definitions speak of *rooted trees*. That is usually redundant because there are no trees without a root.
But some definitions (unlike ours) allow for trees where the root is undetermined and multiple nodes could be the root. Then rooted trees are trees with a distinguished root node.

Trees vs. Labeled Trees We distinguish between trees, which just define the shape, and trees over a set, where the nodes are labeled with data. Others may or may not make that distinction and may use the word *tree* to refer to either concept.

Order of Children Some definition may make the nodes in a tree a *set* of children instead of (as in our definition) a *list*.

Leaf-Labeled Trees Our $Tree[A]$ data structure contains trees in which *every* node stores data from A. Occasionally, we are also interested in trees where only the *leafs* are labeled. And sometimes we need trees where inner nodes are labeled with elements of A and leafs with elements of B.

Single Children in Binary Trees Some people will speak of binary trees if every node has 0 or 2 nodes (but not 1).
When nodes with 1 child are allowed (like in our definition), definitions may or may not distinguish whether that one child is the left or the right child. Thus, they may consider the following trees to be the same (like in our definition) or different (which would make the definition of binary search tree in Sect. 11.2.4 simpler):

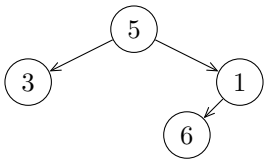


Properties of Binary Trees The properties *complete*, *full*, *balanced*, and *perfect* are all similar. They all relate to the goal of arranging a fixed number of nodes into a tree of small height.
But their definitions vary slightly.

Heaps Some people say *heap* to refer exclusively to heaps of integers.
Some people will assume that heaps are always binary trees.

10.2 Data Structures

Trees can be mutable or immutable. However, trees are mostly used to store data. Many algorithms work with a single mutable tree and insert data into it or delete data from it over time.
We consider two different data structures and use the following as an example tree



10.2.1 Using Lists

The simplest data structure for trees uses lists:

class $Tree[A](data : A,\ children : List[Tree[A]])\{\}$

The example tree is represented as

$$\textbf{new}\ Tree[\mathbb{Z}](5,\ [\textbf{new}\ Tree[\mathbb{Z}](3, Nil),\ \textbf{new}\ Tree[\mathbb{Z}](1, [\textbf{new}\ Tree[\mathbb{Z}](6, Nil)])])$$

10.2.2 Using Sibling Pointers

Some programmers or programming languages prefer a more awkward (but less memory-intensive) data structure that does not use lists.

Here every node has two pointers: one to its first child and one to its next sibling:

class $Node[A](data : A,\ firstChild : Node[A], nextSibling : Node[A])\{\}$

For leafs, the field $firstChild$ is $null$; for the last child of a node, the field $nextSibling$ is $null$. It would be better not to use $null$. But programmers who use this data structure usually do not mind.

The example tree is represented as

$$\textbf{new}\ Node[\mathbb{Z}](5,\ \textbf{new}\ Tree[\mathbb{Z}](3, null,\ \textbf{new}\ Tree[A](1, \textbf{new}\ Tree[\mathbb{Z}](6, null, null)), null), null)$$

10.3 Important Algorithms

10.3.1 Search

Trees are often used to represent a problem.

> *Example* 10.8. Consider a labyrinth in which some treasure is hidden. We represent it as a tree. The entrance is the root. Every fork in the path is a node with multiple children—one child per direction we can go in. Every dead end is a leaf. One node in the tree is special because it has the treasure.
>
> To find the treasure, we have to explore the labyrinth. That means we have to visit every node of the tree until we find the treasure.

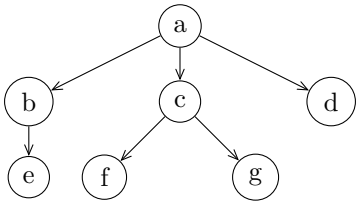
Many problems in real life can be seen as labyrinths in the sense that we have to make a series of decisions, each time choose between multiple options.

Therefore, many problems can naturally be represented as trees. Moreover, if we do not have any special knowledge (e.g., a map leading to the treasure), the only thing we can do is systematically explore all nodes of the tree.

That is straightforward in principle, but we have to decide in which order we explore the nodes. Two strategies are important:

- In Breadth-First Search (BFS), we explore nodes in increasing order of depth: first the root, then the children, then the grandchildren of the root, and so on. We can visualize this as searching top-to-bottom (if the tree is drawn in the usual way with the root at the top). Thus, we search the entire breadth before moving on to deeper nodes.
- In Depth-First Search (DFS), we first explore all descendants of a node n before moving on the siblings of n. We can visualize this as searching left-to-right. Thus, we search as deep as we can before moving on to the siblings.

Consider the tree below. BFS yields abcdefg. DFS yields abecfgd.



BFS has the drawback of back-and-forth movement. For the tree above, we have to go from a to b, back up to a and down to c, back up and down to d, then all the way back to b, so that we can go e, back up all the way to a, down to c again, and so on. DFS is much simpler.

However, it is much more common to have a very high tree (i.e., long branches) than a very wide tree (i.e., lots of branches). This is because we often have many decisions to make, but each decision only has a few options. For

example, many games consist of an unlimited number of turns where at each turn we have to choose from a limited number of moves. In those situations, if DFS picks the wrong cild of the root early on, it may have to explore a huge subtree before coming back to pick the right child.

BFS is more balanced and predictable. If we know the probability of finding a solution becomes smaller at greater depths, BFS makes sure that we explore the most promising nodes first.

Depth-First Search

DFS can be realized quite easily with a recursive function, especially if we use the data structure from Sect. 10.2.1. We use an arbitrary function f as the payload, i.e., a function that is to be called at every node n. For example, f can check if n is the needed solution or do some other work on n.

```
fun DFS[A](n : Tree[A], f : Tree[A] → unit) =
    f(n)
    foreach(n.children, x ↦ DFSAux[A](x, f))
```

In this variant of DFS, f acts on every node n before it recurses into the children. It is also possible to switch those two, i.e., first recurse into the children, then call $f(n)$.

Breadth-First Search

BFS is a bit more complicated. One way to do it is to use a queue that stores all nodes that we have already seen but not acted on yet. That way we can avoid the back-and-forth movement.

```
fun BFS[A](n : Tree[A], f : Tree[A] → unit) =
    needToVisit := new Queue[Tree[A]]()
    enqueue(needToVisit, n)
    while !empty(needToVisit)
        n := dequeue(needToVisit)
        f(n)
        foreach(n.children, x ↦ enqueue(needToVisit, x))
```

Here in every iteration of the loop, we process the next node n (dequeue) and then put its children at the end of the queue. That way all children of n are guaranteed to be processed before any grandchildren of n.

The above BFS-algorithm is interesting because we can easily turn it into a DFS-algorithm: all we have to do is use a stack instead of a queue. That way all descendants of n are processed before anything else.

10.3.2 Min-Max Algorithm

Many games can be represented as trees. Consider a 2-player game in which the players alternate taking turns. At every turn, a player has to choose among multple moves. We assume there is no luck (e.g., no dice-rolling) and no hidden information (e.g., no bluffing).

We can represent all possible courses of the games in a single tree as follows:

- Every node represents a turn.
  - root: initial state
  - nodes of even depth (including root): turn of player 1
  - nodes of odd depth: turn of player 2
  - leafs: terminal states (when the game is over)
- For every node n, the children of n are the possible moves in that turn.
- Every branch represents a possible course of the game.

For leafs l, let $score(l) \in \mathbb{Z}^\infty$ represent the outcome:

- ∞: player 1 wins
- positive values: player 1 is ahead
- 0: draw

- negative values: player 2 is ahead
- $-\infty$: player 2 wins

Thus, player 1 wants to maximize the result, player 2 wants to minimize it.

The min-max algorithm builds the entire tree by exploring all possible courses of the game. Let $State$ be the type of game states. We assume some basic functions $isTerminal : State \to bool$ and (for terminal states) $result : State \to \mathbb{Z}^\infty$ that represent the rules of the game.

Let us assume we have built the tree $game : Tree[State]$. Then we can call the minmax algorithm with $minxmax(game, 0)$ to aggregate the results of the terminal states:

```
fun minmax(current : Tree[State], depth : ℕ) : ℤ^∞ =
    state := current.data
    if isTerminal(state)
        result(state)
    else
        childResults := map(current.children, n ↦ minmax(n, depth + 1))
        if even(depth)
            max(childResults)
        else
            min(childResults)
```

If $minxmax(game, 0) = \infty$, then player 1 has a perfect strategy to win every game. Correspondingly for player 2. If $minxmax(game, 0) = 0$, then both players have a perfect strategies to hold a draw.

In practice, the tree is usually far too big to build. Therefore, instead of obtaining the result at terminal states, we must estimate the result at cut-off. For example, at depth 6, we estimate the current score using heuristic function $State \to \mathbb{Z}^\infty$.

This is a basic design used in artificially intelligent computer players for many games. Many optimizations are needed to obtain strong players.

Chapter 11

Set-Like Data Structures

11.1 Specification

The set $Set[A]$ contains the finite subsets of A. It is countable if A.
Sets can be mutable or immutable. The main operations for immutable sets are:

function	returns	effect
$contains(x \in Set[A], a \in A) \in bool$	*true* iff $a \in x$	none
$insert(x \in Set[A], a \in A) \in Set[A]$	$x \cup \{a\}$	none
$delete(x \in Set[A], a \in A) \in Set[A]$	$x \setminus \{a\}$	none

The main operations for mutable sets are:

function	returns	effect
$contains(x \in Set[A], a \in A) \in bool$	*true* iff $a \in x$	none
$insert(x \in Set[A], a \in A) \in unit$	nothing	$x := x \cup \{a\}$
$delete(x \in Set[A], a \in A) \in unit$	nothing	$x := x \setminus \{a\}$

In both cases, we often need operations for combining and comparing sets:

function	returns	effect
$equal(x \in Set[A], y \in Set[A]) \in bool$	*true* iff $x = y$	none
$union(x \in Set[A], y \in Set[A]) \in Set[A]$	$x \cup y$	none
$inter(x \in Set[A], y \in Set[A]) \in Set[A]$	$x \cap y$	none
$diff(x \in Set[A], y \in Set[A]) \in Set[A]$	$x \setminus y$	none

Equality is listed explicitly here because it can be very complex. For most data structures such as the ones for lists and trees, equality is straightforward. This may or may not be the case for data structures for sets.

11.2 Data Structures

The complexity of data structures for sets is usually measured in terms of the size n of the set.

11.2.1 Bit Vectors

Design

If A is finite with $|A| = m$, an easy data structure for $Set[A]$ are bit vectors of length m such as $Array[bool](m)$. Given such a vector a, we put $a[i] = true$ to represent that i is in the set.

Complexity

We can implement *insert* and *delete* easily in $\Theta(1)$. We can also implement *equal*, *union*, *inter*, and *diff* easily in $\Theta(m)$.
A major drawback is the memory requirement: We need $\Theta(m)$ for each $x : Set[A]$, which is only feasible for small m.

11.2.2 List Sets

Design

If A is much larger than the sets to be represented, a better data structure for $Set[A]$ is $ListSet[A]$. It represents the set $\{a_1, \ldots, a_n\}$ as the list $[a_1, \ldots, a_n]$. Thus, it represent sets as lists without repetition.
The operations on $ListSet[A]$ are defined in the same way as for $List[A]$ with one exception: the $insert(x, a)$ operation does nothing if x already contains a.

Complexity

If n is the size of the $ListSet$, the operations *contains*, *insert*, and *delete* takes $\Theta(n)$. However, higher-level operations like building a set with n elements step-by-step by calling *insert* n times requires n insertions and thus costs $\Theta(n^2)$.
Moreover, these operations require calls to the equality on A. For example, to implement $contains(x, a)$, we have to compare a to every element of x. That may be easy, e.g., if $A = int$. But it can be arbitrarily costly if A is more complex data structure itself.
For equality, union, intersection, and difference of x and y, we may have to compare every element of x with every element of y. So it may take $O(|x| \cdot |y|)$.
These operations quickly become too costly for large subsets of A.

11.2.3 Hash Sets

Design

Hash sets try to combine the advantages of bit vector and list sets. The key parameter is a function $hash : A \to \mathbb{Z}_m$. This is called the hash function.
hash has two purposes:

- The set A is supported by a finite, managably small set $\mathbb{Z}_m$. That makes if feasible to use arrays of length m.
- The equality operation on A is supported by the $O(1)$ equality on $\mathbb{Z}_m$. To check $a = a'$, we first check $hash(a) = hash(a')$. If false, we know $a \neq a'$; otherwise, we call the usual equality on A. That minimizes the number of equality on A is called.

Of course, the function *hash* will usually not be injective. A **collision** is a pair $x, y \in A$ such that $hash(x) = hash(y)$.
A good hash function should be fast and rarely have collisions. An (unrealistically) ideal hash function runs in $O(1)$ and the probability of $hash(x) = hash(y)$ is $1/m$. Those two properties work against each other: For example, it is easy to be fast by always returning 0, but that has maximally many collisions. Vice versa, it is easy to minimize collisions by choosing *hash* carefully, but then *hash* may be very expensive to compute. Thus, hash functions must make a trade-off.
For a fixed hash function $hash : A \to \mathbb{Z}_m$, the data structure $HashSet[A]$ is given by

type $HashSet[A] = Array[ListSet[A]](m)$
fun $insert(h : HashSet[A], a : A) = \{insert(h[hash(a)], a)\}$
fun $delete(h : HashSet[A], a : A) = \{delete(h[hash(a)], a)\}$

Complexity

If n is the size of the subset of A, the sets $h[0], \ldots, h[m-1]$ have average size n/m. Thus, *contains*, *insert* and *delete* take n/m on average. *equal*, *union*, *inter*, and $diff$ are similarly sped up.

Asymptotically, hash sets do not beat list sets because they only spped up by a constant factor. But that constant factor is a critical improvement.

The speed up is bigger if m is bigger. However, the memory requirement increases linearly with m: Even the empty subset requires $\Theta(m)$ space and $\Theta(m)$ time to initialize that space.

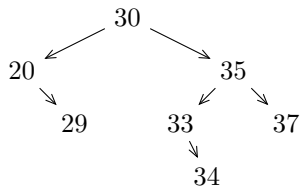
Optimized data structures for hash sets can dynamically choose m in order to find a good trade-off. Often users of the *HashSet* data structure can choose the value of m. That can help if they know in advance how big the subset is going to get and what kind of operations will be called.

11.2.4 Binary Search Trees

Design

If we have a total order $\leq$ on A, we can use binary trees to realize $Set[A]$. The idea is that the nodes of the tree hold the elements of the set, and every node n splits a range of values into two subranges: all left descendants of n have smaller and all right descendants have greater values than n.

Example 11.1. We represent the set $\{20, 29, 30, 33, 34, 37\} \subseteq \mathbb{N}$ as the following binary tree over $\mathbb{N}$:



Now we can locate an element efficiently by traversing just one branch of the tree.

Note how a binary search tree has to distinguish between the left and the right subtree even if there is only one subtree (e.g., for the nodes for 20 and 33). In those cases, we have to use a dummy node as the omitted left child. These are usually labeled with *null* and omitted when drawing the tree.

The formal definition is as follows:

Definition 11.2 (Binary Search Trees)**.** $BST[A, O]$ is the subset of $Tree[A^?]$ containing only full binary trees satisfying the following properties:

- All leafs are labeled with the value $\perp$; all other nodes are labeled with elements of A.
- For every non-leaf node n:
  - All values in the left subtree of n are strictly smaller than the one at n.
  - All values in the right subtree of n are strictly greater than the one at n.

The leafs hold dummy values as placeholders for elements that we may want to insert later. When drawing binary search trees, we always omit them.

Complexity

Ideally, the binary tree is perfect or nearly perfect. That happens when every node is labeled with the median of all the values among its descendants.

In that case, the height of the tree is logarithmic in the size n of the set. Then *contains*, *insert*, and *delete* can be implemented in $O(\log n)$.

But a series of random insertions and deletions may make the tree arbitrarily imperfect. In the worst case, the tree degenerates to a list. Therefore, a binary search tree must be rearranged from time to time.

This can be done from scratch in one big operation. For example, we can do a depth-first traversal of the tree to obtain a sorted list of all elements. Then we can create a perfect binary tree of height $\log n$ and put the elements into it. Overall that takes $\Theta(n)$.

Alternatively, we can rearrange the tree incrementally. Here we try to make minor changes to the tree after every insertion or deletion. To keep the tree near perfect. One way to do that is to use red-black trees (see Sect. 11.2.5).

11.2.5 Red-Black Trees

Red-black trees are a sophisticated variant of binary search trees. They guarantee $\Theta(\log n)$ cost for for insertion and deletion.

Chapter 12

Graph-Like Data Structures

After lists, and trees, graphs are the most important data structure in computer science. In fact, just like lists are a special of trees, trees are a special of graphs.

Data structures for lists and trees are of course used to represent lists and trees. But they are also used to represent other data. For example, we can represent a set as a list (Sect. 11.2.2) or as a tree (Sect. 11.2.5) or a list as a tree (Sect. 9.6). That is because choosing the more complex data structure (i.e., a tree instead of a list) can allow for more efficient algorithms.

Data structures for graphs on the other hand are almost exclusively used to represent graphs. That is because they are rather difficult to work with. But they are needed because graph-like data occurs very frequently.

12.1 Specification

Definition 12.1 (Graph)**.** A **graph** consists is a pair $G = (N, E)$ such that E is a binary relation on N.
If E is symmetric, G is called **undirected**, otherwise **directed**.

The set N is usually but not necessarily finite.

Like for trees, there are many definitions to talk about the parts of a graph:

Definition 12.2 (Parts of a Graph)**.** Consider a graph $G = (N, E)$.

An element $n \in N$ is called a **node** or a **vertex**. An element $(m, n) \in E$ is called an **edge** from m to n. It is also called an **incoming edge** of n and an **outgoing edge** of m.

For every node n, the number of incoming edges is called the **in-degree** of n, and the number of outgoing edges is called the **out-degree** of n. If G is undirected graph, incoming and outgoing edges are the same, and we simply speak of the **degree** of n.

A **path** from a_0 to a_n is a list $[a_0, \ldots, a_n] \in N^*$ such that there is an edge from a_{i-1} to a_i for $i = 1\,ldots, n$.

n is called the **length** of the path. If $n = 0$ (and thus $a_0 = a_n$), the path is called **empty**. If there is a path from a_0 to a_n, then a_n is called **reachable** from a_0.

A **cycle** is a non-empty path from a to itself. If G has (no) cycles, it is called **(a)cyclic**.

Let us write $\overline{G}$ for the undirected graph $(N, E \cup E^{-1})$ in which all edges go both ways. Then G is called **connected** if all nodes in $\overline{G}$ are reachable from each other.

A **clique** is a subset C of N such that there is an edge from every $a \in C$ to every other $b \in C$. G is called **complete** if N is a clique.

Visualization A good intuition to think of graph is to imagine the nodes as places and the edges as streets between them. In a directed graph, all edges are one-way streets.

All concepts about graphs also have very intuitive visual aspects:

	Visual Intuition	
Concept	undirected	directed
node	point	
edge from a to b	line from a to b	arrow from a to b
incoming edge of a		arrow pointing at a
outgoing edge of a		arrow pointing away from a
b reachable from a	we can walk from a to b along edges	... in arrow direction
path from a to b of length n	a walk from a to b in n steps	... in arrow direction
weight[1] of an edge	cost intuition: length of the line	
	capacity intuition: width of the line	
complete	we can walk everywhere in 1 step	... in arrow direction
cycle	we can walk in a circle	... in arrow direction
connected	graph can be drawn in one stroke	

Reachability Relation Many graph properties are just rephrasings of or closely related to relation properties. Most importantly:

Theorem 12.3 (Reachability)**.** *For every graph G, the relation "b is reachable from a" is*
- *reflexive and transitive*
- *symmetric iff G is undirected*
- *anti-symmetric iff G is acyclic*

Proof. Exercise. □

Labeled Graphs Like for trees, graphs are only useful for compuation, if we can store data in them. Contrary to trees, we often need to store data in the nodes *and* the edges.

Definition 12.4 (Labeled Graph)**.** A A-B-**labeled** graph is a triple of
- a graph $G = (N, E)$
- a function $nodeLabel : N \to A$
- a function $edgeLabel : E \to B$

$Graph[A, B]$ is the set of A-B-labeled graphs.

The most important special case arises when the nodes are not labeled (i.e., we put $A = unit$) and the edges are labeled with numbers, i.e., $B = \mathbb{Z}$:

Definition 12.5. A **weighted** graph is a $unit$-$\mathbb{N}$-labeled graph. The label of an edge from is called its **weight**.

There are two important applications of weigthed graphs that use different interpreations of the weights:

- Cost intuition: The weight of an edge is the cost of moving along the edge. For example, if the nodes represent cities and the edges flight routes, the weight can be the distance.
- Capacity intuition: The weight of an edge is the capacity for moving objects along the edge. For example, if the nodes represent cities and the edges flight routes, the weight can be the number of flights per day.

Correspondingly, we define:

Definition 12.6. Consider a weighted graph. We write $weight(i, j)$ for the weight of the edge from i to j.
We make $weight : N \times N \to \mathbb{N}^\infty$ a total function by using a default value whenever there is no edge from i to j:
- A **cost-weighted** graph uses the default $weight(i, j) = \infty$.
- A **capacity-weighted** graph uses the default $weight(i, j) = 0$.

In a cost-weighted graph, the **cost of a path** is the sum of the weights of all edges.
In a capacity-weighted graph, the **capacity of a path** is the minimal weight of any edge in it.

[1] See below for weighted graphs.

12.2 Data Structures

Graphs $G = (N, E)$ are among the trickiest data structures to design. It is straightforward to represent N as a set, e.g., the set $\mathbb{Z}_m$ if there are m nodes.
But there are many options to represent E, all with different advantages: For example,

- a function $N \times N \to Bool$
  This makes it easy to check whether an edge exists but very hard to enumerate all edges.
- a set e with functions $from : e \to N$ and $out : e \to N$
  This makes it easy to enumerate all edges but hard to navigate in the graph.
- a function $outgoing : N \to Set[N]$ or $incoming : N \to Set[N]$
  The former makes it easy to navigate forwards (in arrow direction) but hard to navigate backwards. The latter has the dual problem.

- two functions $outgoing : N \to Set[N]$ and $incoming : N \to Set[N]$
  This makes navigation easy in both directions. But the representation is redundant: Every time we add/remove an edge, we have to update both *outgoing* and *incoming*.

12.2.1 Adjacency Matrix

An often useful representation is via a matrix, called the **adjacency matrix** of G.

Definition 12.7 (Adjacency Matrix)**.** Given a graph $G = (N, E)$ with $|N| = m$. The adjacency matrix of G is the matrix $A \in bool^{mm}$ where $A_{ij} == true$ iff there is an edge from i to j in G.

Adjacency matrices have the nice property that we can multiply them. Here matrix multiplication is computed using conjunction and disjunction instead of multiplication and addition:

Definition 12.8. $A, B \in bool^{mm}$, we define $(A \cdot B)_{ik} := \bigvee_{j=0,\dots,m-1} A_{ij} \wedge B_{jk}$.

This is useful because:

Theorem 12.9. *If A is the adjacency matrix of G, then $(A^n)ij$ iff there is a path of length n from i to j in G.*

This is advantageous because it lets us compute all paths of a given length efficiently A^n using square-and-multiply (Sect. 4.1.3).
Moreover, in acyclic graph, there are only finitely many paths. Thus, we eventually have $A^n = A^{n+1} = \dots$, at which point we have computed all paths.
A drawback of the adjacency matrix is that its size m^2. In particular, for a undirected graph, half the space is wasted because we always have $A_{ij} = A_{ji}$).

12.2.2 Adjacency Lists

For graphs with many nodes and few edges, it is better to store adjacency lists:

Definition 12.10 (Adjacency List)**.** Given a graph $G = (N, E)$ with $|N| = m$. The adjacency list of a node i is the sorted list A_i of all j such that there is an edge from i to j in G.
The adjacency list–representation of G consists of an list $[(0, A_0), \dots, (m-1, A_{m-1})]$ pairing every node with its adjacency list.

The size of the adjacency list–representation is $|N| + |E|$, which is usually much smaller than $|N|^2$.

12.3 Important Algorithms

12.3.1 Search

12.3.2 Minimal Spanning Tree

12.3.3 Shortest Path

12.3.4 Maximal Flow

Chapter 13

Function-Like Data Structures

Chapter 14

Union-Like Data Structures

Chapter 15

Product-Like Data Structures

Chapter 16

Algebraic Data Structures

16.1 Specification

Algebraic data types are a not clearly delineated family of record types. Typically one field of the record is a type, and the other fields are operations on that type.

Many of them represent mathematical theories, in which case their elements represent mathematical structures. Therefore, they tend to come up a lot. But their high level of abstraction leads to them often being neglected or not understood.

Three classes are of major importance: these are the data types based on one type U and

- one binary relation $U \times U \to \mathit{bool}$ such as in graphs, preorders, orders, and equivalence relations,
- one binary function $U \times U \to U$ such as in monoids, groups, and semi-lattices,
- two binary functions $U \times U \to U$ such as in rings, fields, and lattices

and several axioms such as transitivity, associativity, or distributivity.

The most important special cases are specified in Sect. A.1 for a binary relation and in Sect. A.2 for one binary function. We omit the case for two binary functions.

16.2 Data Structures

Apart from the axioms, we can implement algebraic data types very well as polymorphic abstract classes that take U as the type parameter. However, the axioms can usually not be implemented elegantly (except in very advanced programming languages) and must be realized as comments.

16.2.1 One Binary Relation

We already implemented some of them when sorting lists.

For example, the type of orders on U can be realized as

```
abstract class Relation[A]()
    fun rel(x : A, y : A) : bool =
        ...
abstract class Preorder[A]() extends Relation[A]
                                                    rel is reflexive and transitive
abstract class Order[A]() extends Preorder[A]
                                                    rel is anti-symmetric
abstract class TotalOrder[A]() extends Order[A]
                                                    rel is total
```

Individual structures can now be implemented as concrete classes that implement the abstract ones. We have already implemented some concrete orders such as $\leq$ on *int*, $|$ on *int*, or the lexicographic ordering on *string*.

16.2.2 One Binary Function

For example, the type of monoids on U can be realized as

```
abstract class BinOp[A]()
    fun op(x : A, y : A) : A =
        ...
abstract class Monoid[A]() extends BinOp[A]
    fun e() : A =
        ...
                                        op is associative and has neutral element e
```

Individual structures can now be implemented as concrete classes that implement the abstract ones.
The following implements some example monoids:

```
class Addition() extends Monoid[int]
    fun op(x : int, y : int) = {x + y}
    fun e() : A = {0}
```

```
class Multiplication() extends Monoid[int]
    fun op(x : int, y : int) = {x * y}
    fun e() : A = {1}
```

```
class Concatenation() extends Monoid[string]
    fun op(x : string, y : string) = {x + y}
    fun e() : A = {""}
```

```
class Maximum() extends Monoid[ℕ]
    fun op(x : ℕ, y : ℕ) = {if (x ≤ y) {y} else {x}}
    fun e() : A = {0}
```

In each case, we have to check that the axioms (associativity and neutrality) actually hold to show the correctness of the implementations.
For a more complex example, consider the monoid of 2×2 matrices under multiplication:

```
class Matrix22(a : int, b : int, c : int, d : int)

class Matrix22Multiplication() extends Monoid[Matrix22]
    fun op(x : ℕ, y : ℕ) = {...}
    fun e() : A = {new Matrix22(1, 0, 0, 1)}
```

where **new** $Mat(a, b, c, d)$ represents the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

16.2.3 Two Binary Functions

The types of, e.g., rings and fields are implemented accordingly.

16.3 Important Algorithms

The implementation of algebraic data types and algorithms that

- are used in concrete algebraic structures
- compute new algebraic structures from existing ones

are studied in the field of *computer algebra*. This is essential to for mathematical computation.

Additionally, many algorithms can be easily generalized to take an arbitrary structure as their input. For example, sorting can be implemented as a function

$$sort(x : List[int]) : List[int].$$

But it is much more general, equally easy, and essentially as fast to implement a function

$$sort[A](ord : TotOrd[A], x : List[A]) : List[A]$$

that sorts with respect to an arbitrary total order.
We give some examples.

16.3.1 Folding Lists over a Monoid

The function

$$fold[A, B](x : List[A], f : A \times B \to B) : B$$

for lists is important but often confusing.
It becomes much simpler if we consider the special case $A = B$:

$$fold[A](x : List[A], e : A, f : A \times A \to A) : A$$

If we additionally write f as infix, i.e., write $x\, f\, y$ instead of $f(x, y)$, we have

$$fold[A]([a_1, \ldots, a_n], e, f) = a_1\, f\, (a_2\ \ldots\ (a_{n-1}\, f\, (a_n\, f\, e))\ldots)$$

In general, the bracketing matters. But if f is associative, the bracketing becomes irrelevant. In that case, we usually want e to be a neutral element for f.
That yields

$$monoidFold[A](mon : Monoid[A], x : List[A])$$
$$monoidFold[A](mon, [a_1, \ldots, a_n]) = a_1\, mon.op\, a_2\ \ldots\ a_{n-1}\, mon.op\, a_n\, mon.op\, mon.e$$

In particular, we have

$$monoidFold[A](mon, []) = mon.e \qquad \text{and} \qquad monoidFold[A](mon, [a]) = a.$$

For example,

- $monoidFold[int](\textbf{new}\ Addition(), x)$ is the sum of all elements in x,
- $monoidFold[int](\textbf{new}\ Multiplication(), x)$ is the product of all elements in x,
- $monoidFold[\mathbb{N}](\textbf{new}\ Maximum(), x)$ is the greatest element in x,
- $monoidFold[string](\textbf{new}\ Concatenation(), x)$ is the concatenation of all strings in x.

16.3.2 Square-and-Multiply

We can finally give the square-and-multiply algorithm from Sect. 4.1.3 in full generality.
This should be a function

```
fun sqmult[A](mon : Monoid[A], x : A, n : ℕ) : A =
    ...
```

that satisfies the specification

$$sqmult(mon, x, 0) = mon.e$$
$$sqmult(mon, x, n + 1) = mon.op(power(mon, x, n), x)$$

and whose time complexity is $\Theta(\log_2 n)$.

Part III

Important Families of Algorithms

Chapter 17

Recursion

Chapter 18

Backtracking

Chapter 19

Divide and Conquer

Chapter 20

Parallelization and Distribution

Chapter 21

Greedy Algorithms

Chapter 22

Dynamic Programming

Chapter 23

Protocols

Chapter 24

Randomization

Chapter 25

Quantum Algorithms

Part IV

Concrete Languages

Part V

Appendix

Appendix A

Mathematical Preliminaries

A.1 Binary Relations

A binary relation on a set 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 $\equiv$.

[1] Orders are also called *partial order*, *poset* (for partially ordered set), or *ordering*.

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 cases, 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 "$x \leq y$ and $y \leq x$" is an equivalence.*
For equivalence classes X and Y of the resulting quotient, $x \leq y$ holds for either all pairs or no pairs $(x, y) \in X \times Y$.
If it holds for all pairs, we write $X \leq Y$.
The relation $\leq$ on the quotient is an order.

Remark A.9 (Totality). If $\leq$ is a preorder, then for all elements x, y, there are four mutually exclusive options:

	$x \leq y$	$x \geq y$	$x = y$
x strictly smaller than y, i.e., $x > y$	true	false	false
x strictly greater than y, i.e., $x < y$	false	true	false
x and y incomparable	false	false	false
x and y similar	true	true	maybe

Now anti-symmetry excludes the option of similarity (except when $x = y$ in which case trivially $x \leq y$ and $x \geq y$). And totality excludes the option of incomparability.
Combining the two exclusions, a total order only allows for $x > y$, $y < x$, and $x = y$.

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.10 (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
- if e is a neutral element:
  - left-inverse of x: $a \circ x = e$
  - right-inverse of x: $x \circ a = e$
  - inverse of x: left-neutral and right-neutral of x

Moreover, we say that $\circ$ is a ... if it is/has:

- semigroup: associative
- monoid: associative and neutral element
- group: monoid and inverse elements for all x
- semilattice: associative, commutative, and idempotent
- bounded semilattice: semilattice and neutral element

Terminology A.11. The terminology for *absorbing* is not well-standardized. *Attractive* is an alternative word sometimes used instead.

Theorem A.12. *Neutral and absorbing element of $\circ$ are unique whenever they exist.*
If $\circ$ is a monoid, then the inverse of x is unique whenever it exists.

A.3 The Integer Numbers

A.3.1 Divisibility

Definition A.13 (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.14 (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.15 (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:* $\gcd(x, y) = \gcd(y, x)$
- *idempotent:* $\gcd(x, x) = x$
- 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.16. *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 algorithm.

Definition A.17. If $\gcd(x, y) = 1$, we call x and y **coprime**.
For $x \in \mathbb{N}$, the number of coprime $y \in \{0, \dots, x-1\}$ is called $\varphi(x)$. φ is called Euler's **totient function**.

Example A.18. 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.19 (Equivalence Modulo)**.** For $x, y, m \in \mathbb{Z}$, we write $x \equiv_m y$ iff $m | x - y$.

Theorem A.20 (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.21 (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.22 (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 $x \equiv_m y$ then $y \equiv_m x$*

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: $x \cdot y \equiv_m x' \cdot y'$*
- *preserved by division if $x/y \in \mathbb{Z}$ 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.23 (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.24 (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.25 (Possible Values). For $m \neq 0$, we have $x \bmod m \in \{0, \ldots, m-1\}$. In particular, there are m possible values for $x \bmod m$.
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.26 (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.27 (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*

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

$$x/_m y = x \cdot_m a \qquad \text{if } ay + bm = 1 \text{ as in Thm. A.16}$$

Theorem A.28 (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 $x^{p-1} \equiv_p 1$.

A.3.4 Digit-Base Representations

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

Theorem A.29 (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.30 (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.31 (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.32 (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 Because p is prime, $x/_p y$ is defined for all $x, y \in \mathbb{Z}_p$ with $y \neq 0$. Moreover, $\mathbb{Z}_p$ is a field.

Up to isomorphism, all finite fields are obtained as n-dimensional vector spaces $\mathbb{Z}_p^n$ for some prime p and $n \geq 1$. This field is usually called F_{p^n} because it has p^n elements. From now on, let $q = p^n$.

The 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 if $n > 1$. 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 $X^n = k_{n-1}X^{n-1} + \ldots + k_1 X + k_0$ for certain k_i. The resulting polynomial has degree at most $n-1$, and its coefficients (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 such 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 vectors 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.33. 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.34. *For every $x \in F_q$, we have $x^q = x$. If $x \neq 0$, that is equivalently to $x^{q-1} = 1$.*

For some x, the period is indeed $q-1$, i.e., we have $\{1, x, x^2, \ldots, x^{q-1}\} = F_q \setminus \{0\}$. Such an x is called a **primitive element** 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 than $q-1$, x^l does not take all possible values in F_q. In that case $\log_x y$ 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 $x, x^2, \ldots$ until $x^l = 1$ (in which case the logarithm is undefined) or $x^l = y$ (in which case the logarithm is l). However, no efficient algorithm is known.

A.3.6 Infinity

Occasionally, it is useful to compute also with infinity ∞ or $-\infty$. When adding infinity, some but not all arithmetic operations still behave nicely.

Positive Infinity We write $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$.

The order $\leq$ works as usual. ∞ is the greatest element.

Addition works as usual. ∞ is an attractive element.

Subtraction is introduced as usual, i.e., $a - b = x$ whenever x is the unique value such that $a = x + b$. Thus, $\infty - n = \infty$ for $n \in N$. $x - \infty$ is undefined. The law $x - x = 0$ does not hold anymore.
Multiplication becomes partial because $\infty \cdot 0$ is undefined. For $x \neq 0$, we put $\infty \cdot x = \infty$.
Divisibility $|$ is defined as usual. Thus, we have $x|\infty$ for all $x \neq 0$, and $\infty|x$ iff $x = \infty$. There is no greatest element anymore because: 0 and ∞ are both greater than every other element except for each other.

Negative Infinity We write $\mathbb{Z}^\infty = \mathbb{Z} \cup \{\infty, -\infty\}$.
The order $\leq$ works as usual. $-\infty$ is the least and ∞ the greatest element.
Addition becomes partial because $-\infty + \infty$ is undefined. We put $-\infty + z = -\infty$ for $z \neq \infty$.
Subtraction is introduced as usual. Thus, $z - \infty = -\infty - z = -\infty$ for $z \in \mathbb{Z}$. $\infty - \infty$ is undefined.
Multiplication works similarly to $\mathbb{N}^\infty$. $-\infty \cdot 0$ is undefined. And for $x \neq 0$, we define $\infty \cdot x$ and $-\infty \cdot x$ as ∞ or $-\infty$ depending on the signs.

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.35 (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.36 (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
s	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
s	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
s	$m \in \mathbb{N} \setminus \{0\}$	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.37. *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.35:

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		
void	alternative name for $\varnothing$	0
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 $int = \mathbb{Z}$. Some programming languages (e.g., Python) correctly implement $int = \mathbb{Z}$.

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

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 function on int. [7]

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 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.36:

[7] Some care must be taken if x is negative because not all programming languages agree.

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	$\begin{cases} 1 & \text{if } \lvert A\rvert = 0 \\ U & \text{if } \lvert A\rvert = U \\ C & \text{otherwise} \end{cases}$
$A^?$	optional element[8] 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[9]		C
classes[10]		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^*	$[a_0, \ldots, a_{l-1}]$[11] 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

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

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

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

[11] 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.