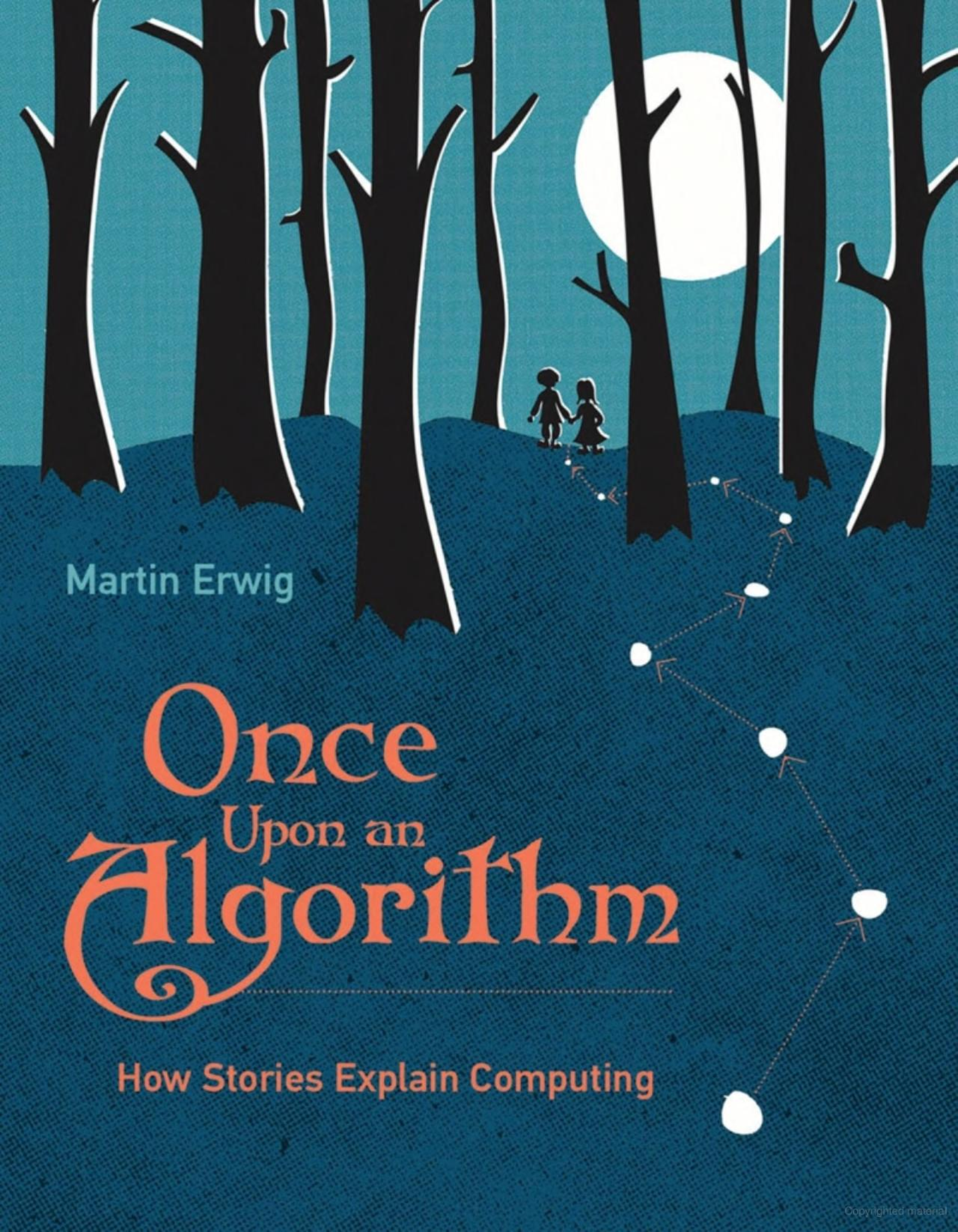


Martin Erwig

Once Upon an Algorithm

How Stories Explain Computing

Once Upon an Algorithm

How Stories Explain Computing

Martin Erwig

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Garamond by the author using the LaTeX document preparation system. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Erwig, Martin, author.
Title: Once upon an algorithm : how stories explain computing / Martin Erwig.
Description: Cambridge, MA : MIT Press, [2017] | Includes bibliographical references and index.
Identifiers: LCCN 2016053945 | ISBN 9780262036634 (hardcover : alk. paper)
Subjects: LCSH: Computer algorithms–Popular works.
Classification: LCC QA76.9.A43 E78 2017 | DDC 005.1–dc23 LC record available at https://lccn.loc.gov/2016053945

10 9 8 7 6 5 4 3 2 1

Contents

Preface

When people ask about my work, the conversation quickly turns to the question of what computer science is. To say computer science is the science of computers is misleading (though strictly speaking not incorrect) because most people will take *computer* to mean PC or laptop and conclude that computer scientists spend their time constructing hardware. On the other hand, defining computer science as the study of computation is simply kicking the can down the road, since it immediately raises the question of what computation is.

Over the years, I have come to realize that teaching by just introducing concept after concept doesn't work very well; it is simply too abstract. Nowadays, I typically start by describing computer science as the study of systematic problem solving. Everybody knows what a problem is, and everybody has seen solutions as well. After explaining this view through an example, I often get the chance to introduce the concept of an algorithm, which allows me to point out important differences between computer science and mathematics. Most of the time, I don't need to talk about programming languages, computers, and related technical matters, but even if it comes to that, the concrete problem makes it easy to illustrate these concepts. *Once Upon an Algorithm* is an elaboration of this approach.

Computer science is a relatively new member of the science club, and sometimes it seems it has not yet earned the respect granted to serious scientific disciplines like physics, chemistry, and biology. Think of a movie scene involving a physicist. You will probably see someone discussing complicated formulas written on a blackboard or supervising an experiment wearing a lab coat. The physicist is shown as a reputable scientist whose knowledge is treasured. Now imagine a similar scene involving a computer scientist. Here you will likely see some nerdy guy sitting in a dark, messy room and staring at a computer screen. He is frantically typing on a keyboard, probably trying to break some code or password. In both scenes, an important problem is being solved, but while the physicist might provide some plausible explanation of how it can be solved, the solution of the computer problem remains mysterious, often magical, and much too complex to be explained to a nonspecialist. If computer science is unexplainable to laypeople, why would anyone ever try to know more about it or understand it?

The subject of computer science is *computation*, a phenomenon that affects everybody. I am not talking only about cell phones, laptops, or the internet. Consider folding a paper airplane, driving to work, cooking a meal, or even DNA transcription, a process that occurs in your cells millions of times while you are reading this sentence. These are all examples of computation—a systematic way of problem solving—even though most people would not perceive them as such.

Science provides us with a basic understanding of how the natural world works, and it gives us the scientific method for reliably establishing that knowledge. What applies to science in general holds for computer science, too, especially since we encounter computation in so many different forms and so many different situations. A basic understanding of computation thus provides benefits similar to a basic knowledge of physics, chemistry, and biology in making sense of the world and tackling many real-world problems more effectively. This aspect of computation is often referred to as *computational thinking*.

A major goal of this book is to emphasize the general nature of computation and thus the wide applicability of computer science. My hope is that this will spark a broader interest in computer science and a desire to learn more about it.

I first identify computation in everyday activities and then explain corresponding computer science concepts through popular stories. The everyday situations are taken from a typical workday: getting up in the morning, having breakfast, commuting to work, episodes at the workplace, a doctor's appointment, a hobby activity in the afternoon, having dinner, and reflecting on the day's events in the evening. Each of these fifteen vignettes introduces a book chapter. The chapters then explain the computation concepts using seven popular stories. Each story spans two or three chapters and deals with a specific topic of computer science.

The book has two parts: *algorithms* and *languages*. These are the two main pillars on which the concept of computation rests. Table 1 summarizes the stories and the computer science concepts they illustrate.

We all appreciate a good story. Stories console us, give us hope, and inspire us. They tell us about the world, make us aware of the problems we face, and sometimes suggest solutions. Stories can also provide guidance for our lives. When you think about what stories have to teach us, you probably think about love, conflict, the human condition. But I also think about computation. When Shakespeare's Juliet asks, "What's in a name?" she is on to an important question about representation. Albert Camus's

Table 1

Story	Chapters	Topics
Part I		
Hansel and Gretel	1, 2	Computation and Algorithms
Sherlock Holmes	3, 4	Representation and Data Structures
Indiana Jones	5, 6, 7	Problem Solving and Its Limitations
Part II		
Over the Rainbow	8, 9	Language and Meaning
Groundhog Day	10, 11	Control Structures and Loops
Back to the Future	12, 13	Recursion
Harry Potter	14, 15	Types and Abstraction

The Myth of Sisyphus raises the question of how to face the absurdity of life and also how to detect a never-ending computation.

Stories have multiple layers of meaning. They often include a computational layer. *Once Upon an Algorithm* is an effort to unveil this layer and offer readers a new perspective on stories and computation. I hope that stories will be appreciated for their computational content and that this novel point of view will spark interest in computer science.

Introduction

Computation plays a prominent role in society now. But unless you want to become a computer scientist, why should this prompt you to learn more about it? You could simply enjoy the technology powered by computation and take advantage of its benefits. You likely also don't study avionics to make use of air travel or seek a medical degree to benefit from the achievements of modern health care.

However, the world we live in consists of more than man-made technology. We still have to interact with nonautonomous objects ruled by physical laws. Therefore, it is beneficial to understand the basics of mechanics simply to predict the behavior of objects and to safely navigate one's environment. A similar case can be made for the benefits of studying computation and its related concepts. Computation is not only revealed in computers and electronic gadgets but also occurs outside of machines. In the following I briefly discuss some of the major computer science principles and explain why they matter.

Computation and Algorithms

I invite you to perform the following simple exercise. For this you need a ruler, a pencil, and a piece of (quadrille-ruled) paper. First, draw a horizontal line that is 1 inch long. Then draw a vertical line of the same length, perpendicular to the first, starting at one of its ends. Finally, connect the two open ends of the lines you drew with a diagonal to form a triangle. Now measure the length of the diagonal just drawn. Congratulations; you have just computed the square root of 2. (See figure 1.)

What does this exercise in geometry have to do with computation? As I explain in chapters 1 and 2, it is the execution of an algorithm by a computer that brings about a computation. In this example, you acted as a computer that executed an algorithm to draw and measure lines, which led to the computation of $\sqrt{2}$. Having an algorithm is crucial because only then can different computers perform a computation repeatedly and at different times. An important aspect of computation is that it requires resources

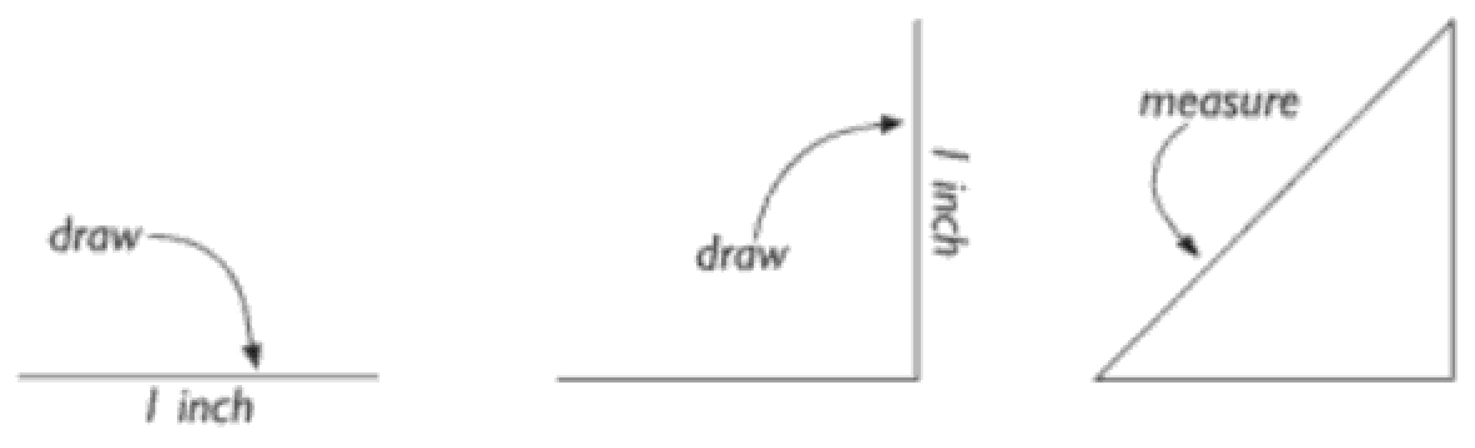


Figure 1 Computing the square root of 2 using a pencil and a ruler.

(such as pencil, paper, and a ruler) and takes time to carry out. Here again, having an algorithmic description is important because it helps in analyzing the resource requirements for computations.

Chapters 1 and 2 explain

- what algorithms are,
- that algorithms are used for systematic problem solving,
- that an algorithm needs to be executed by a computer (human, machine, etc.) to produce a computation,
- that the execution of algorithms consumes resources.

Why does it matter?

Recipes are examples of algorithms. Whenever you make a sandwich, bake a chocolate cake, or cook your favorite dish by following the instructions of a recipe, you are effectively executing an algorithm to transform the raw ingredients into the final product. The required resources include the ingredients, tools, energy, and preparation time.

Knowledge about algorithms makes us sensitive to questions about the correctness of a method and what its resource requirements are. It helps us identify opportunities for improving processes in all areas of life through the (re)organization of steps and materials. For instance, in the geometric square root computation, we could have omitted drawing the diagonal line and just measured the distance between the two unconnected end points.

In cooking the improvement can be as simple and obvious as saving trips to the fridge by planning ahead or gathering ingredients ahead of time. You could plan how to use your oven or stove more efficiently and save time by parallelizing steps, such as preheating the oven or washing the salad while the potatoes are cooking. These techniques also apply to many other areas, from simple assembly instructions for furniture to organizational processes for running an office or managing a factory floor.

In the realm of technology, algorithms control basically all the computations in computers. One prominent example is data compression, without which the transmission of music and movies over the internet would be nearly impossible. A data compression algorithm identifies frequent patterns and replaces them with small codes. Data compression directly addresses the resource problem of computation by reducing the space required to represent songs and movies and thus also the time it takes to load them over the internet. Another example is Google's page-rank algorithm, which determines the order in which search results are presented to a user. It works by assessing the importance of a web page through counting how many links point to it and weighing how important those links are.

Representation and Data Structures

One might expect that numerical computations are carried out with the help of numbers because we use the Hindu-Arabic numeral system, and machines work with 0s and 1s. So the geometric method for computing $\sqrt{2}$ with the help of lines might be surprising. The example demonstrates, however, that one and the same thing (for example, a quantity) can be represented in different ways (number symbols or lines).

The essence of a computation is the transformation of representation. I explain in chapter 3 what representations are and how they are employed in computations. Since many computations deal with large amounts of information, I explain in chapter 4 how collections of data can be efficiently organized. What complicates this question is the fact that any particular data organization may support some forms of accessing the data efficiently but not others.

Chapters 3 and 4 discuss

- different forms of representation,
- different ways of organizing and accessing data collections,
- the advantages and disadvantages of different data organizations.

Why does it matter?

Measurements of ingredients in recipes can be given by weight or by volume. These are different forms of representation, which require different cooking utensils (scales or measuring cups) for properly executing the recipe/algorithm. As for data organization, the way your fridge or pantry is organized has an impact on how fast you can retrieve all the ingredients required for the recipe. Or consider how the question of representation applies to the recipe itself. It may be given as a textual description, through a series of pictures, or even in the form of a YouTube video. The choice of representation often makes a big difference in the effectiveness of algorithms.

In particular, the question of how to arrange a collection of items or people has many applications, for instance, how to organize your desk or garage to help you find items more quickly or how to organize the book shelves in a library. Or consider the different ways you wait in line: in a grocery store (standing in a queue) or at a doctor's office (sitting in a waiting room having picked a number) or when boarding an airplane (multiple queues).

In the realm of technology, spreadsheets are among the most successful programming tools. The organization of data in tabular form has contributed to much of their success because it supports the quick and easy formulation of sums over rows and columns and represents the data and computation results for them in one place. On the other hand, the internet, one of the most transformative inventions of the late twentieth century, organizes web pages, computers, and the connections between them as a network. This representation supports flexible access to information and efficient transmission of data.

Problem Solving and Its Limitations

An algorithm is a method for solving problems, be it finding the square root of a number or baking a cake. And computer science is a discipline that is concerned with systematic problem solving.

Of the many problems that can be solved through algorithms, two deserve to be discussed in detail. In chapter 5, I explain the problem of searching, one of the most frequently used computations on data. Chapter 6 then explains the problem of sorting, which illustrates a powerful problem-solving method and also the notion of intrinsic problem complexity. In chapter 7, I describe the class of so-called intractable problems. Algorithms exist for these problems, but take too long to execute, so the problems are not solvable in practice.

Chapters 5, 6, and 7 clarify

- why searching can be difficult and time consuming,
- methods to improve searching,
- different algorithms for sorting,

- that some computations can support others, such as sorting can support searching,
- that algorithms with exponential runtimes cannot really be considered solutions to problems.

Why does it matter?

We spend countless hours of our lives searching, be it for our car keys or for information on the internet. It is therefore helpful to understand searching and to know about techniques that can help make it more efficient. In addition, the problem of searching illustrates how the choice of representation affects the efficiency of algorithms, reflecting John Dewey's observation, "A problem well put is half solved."[1]

Knowing when a problem *cannot* be solved efficiently is just as important as knowing algorithms for solvable problems because it helps us avoid searching for efficient solutions where none can exist. It suggests that in some cases we should be content with approximate answers.

In the realm of technology, the most obvious instance of searching is given by internet search engines such as Google. Search results for a query are not presented in arbitrary order but are typically sorted according to their anticipated importance or relevance. The knowledge about the hardness of problems is used to develop algorithms that compute approximate solutions where exact solutions would take too long to compute. One famous example is the traveling salesman problem, which is to find the round-trip that visits a certain number of cities in an order that minimizes the total distance traveled.

Knowledge about the lack of an efficient algorithm to solve a problem can also be exploited in a positive way. One example is public key encryption, which enables private transactions on the internet, including managing bank accounts and shopping online. This encryption only works because currently no efficient algorithm is known for prime factorization (that is, writing a number as the product of prime numbers). If that were to change, public key encryption would not be safe anymore.

Language and Meaning

Any algorithm has to be expressed in some language. Current computers cannot be programmed in English because natural languages contain too many ambiguities, which humans, but not machines, can deal with easily. Therefore, algorithms that are to be

In the realm of technology, control structures are used wherever algorithms are used, and thus they are everywhere. Any information sent over the internet is transmitted in a loop repeatedly until it has been properly received. Traffic lights are controlled by endlessly repeating loops, and many manufacturing processes contain tasks that are repeated until a quality measure is met. Predicting the behavior of algorithms for unknown future inputs has many applications in security. For example, one would like to know if a system is vulnerable to attacks by hackers. It also applies to rescue robots that have to be used in situations different from the ones they are trained in. Accurately predicting robot behavior in unknown situations can mean the difference between life and death.

Recursion

The principle of reduction—the process of explaining or implementing a complex system by simpler parts—plays an important role in much of science and technology. Recursion is a special form of reduction that refers to itself. Many algorithms are recursive. Consider, for example, the instructions for looking up a word in a dictionary that contains one entry per page: "Open the dictionary. If you can see the word, stop. Otherwise, look up the word in the dictionary part before or after the current page." Notice how the look-up instruction in the last sentence is a recursive reference to the whole process that brings you back to the beginning of the instructions. There is no need for adding something like "repeat this until the word is found" to the description.

In chapter 12, I explain recursion, which is a control structure but is also used in the definition of data organization. In chapter 13, I illustrate different approaches for understanding recursion.

Chapters 12 and 13 examine

- the idea of recursion,
- how to distinguish between different forms of recursion,
- two different methods to unravel and make sense of recursive definitions,
- how these methods help in understanding recursion and the relationship between its different forms.

Why does it matter?

The recursive definition of "season to taste" is as follows: "Taste the dish. If it tastes fine, stop. Otherwise, add a pinch of seasoning, and then season to taste." Any repeated action can be described recursively by using the action to be repeated (here "season to taste") in its description and a condition when to stop.

Recursion is an essential principle for obtaining finite descriptions of potentially infinite data and computations. Recursion in the grammar of a language facilitates an infinite number of sentences, and a recursive algorithm allows it to process inputs of arbitrary size.

Since recursion is a general control structure and a mechanism for organizing data, it is part of many software systems. In addition, there are several direct applications of recursion. For example, the Droste effect, in which a picture contains a smaller version of itself, can be obtained as a result of a feedback loop between a signal (a picture) and a receiver (a camera). The feedback loop is a recursive description of the repetitious effect. Fractals are self-similar geometric patterns that can be described through recursive equations. Fractals can be found in nature, for example, in snowflakes and crystals, and are also used in analyzing protein and DNA structures. Moreover, fractals are employed in nanotechnology for designing self-assembling nanocircuits. Self-replicating machines are a recursive concept because once they are operating, they reproduce copies of themselves that reproduce further copies, and so on. Self-replicating machines are investigated for space exploration.

Types and Abstraction

Computation works by transforming representations. But not every transformation is applicable to every representation. While we can multiply numbers, we cannot multiply lines, and similarly, while we can compute the length of a line or the area of a rectangle, it does not make sense to do that for a number.

Representations and transformations can be classified into different groups to facilitate the distinction between transformations that are viable and those that don't make sense. These groups are called types, and the rules that determine which combinations of transformations and representations are allowed are called typing rules. Types and typing rules support the design of algorithms. For example, if you need to compute a number, you should employ an operation that produces numbers, and if you need to process a list of numbers, you have to use an operation that accepts lists of numbers as input.

In chapter 14, I explain what types are and how they can be used to formulate rules for describing regularities of computations. Such rules can be used to find errors in algorithms. The power of types lies in their ability to ignore details about individual objects and therefore to formulate rules on a more general level. The process of ignoring details is called abstraction, which is the subject of chapter 15, where I explain why abstraction is central to computer science and how it applies not only to types but also to algorithms, and even computers and languages.

Chapters 14 and 15 discuss

- what types and typing rules are,
- how they can be used to describe laws about computation that help to detect errors in algorithms and to construct reliable algorithms,
- that types and typing rules are just a special case of the more general idea of abstraction,
- that algorithms are abstractions of computation,
- that types are abstractions of representations,
- that runtime complexity is an abstraction of execution time.

Why does it matter?

If a recipe requires the opening of a can of beans, you'd be surprised if someone approached this task using a spoon because that would violate a typing rule that deems spoons inadequate for that task.

The use of types and other abstractions to describe rules and processes is ubiquitous. Any procedure that has to be repeated suggests an algorithmic abstraction, that is, a description that ignores unnecessary details and replaces variable parts by parameters. Recipes also contain algorithmic abstractions. For example, many cookbooks contain a section describing basic techniques, such as peeling and seeding tomatoes, which can then be referred to in recipes by simply requesting a particular amount of peeled and seeded tomatoes. Moreover, the roles of the different objects that take part in such an abstraction are summarized by types that characterize their requirements.

In the realm of technology, there are many examples of types and abstractions. Examples of physical types are all kinds of differently shaped plugs and outlets, screws and screwdrivers and drill bits, and locks and keys. The shapes' purpose is to prevent improper combinations. Software examples of types can be found in web forms that require entering phone numbers or email addresses in a specific format. There are many examples of costly mistakes that were caused by ignoring types. For instance, in 1998, NASA lost its $655 million Mars Climate Orbiter because of incompatible number representations. This was a type error that could have been prevented by a type system. Finally, the notion of a computer is itself an abstraction of humans, machines, or other actors that are capable of executing algorithms.

How to Read This Book

Figure 2 presents an overview of the concepts discussed in this book and how they are related. Chapters 7, 11, and 13 (dark-shaded boxes in figure 2) contain more technical material. These chapters may be skipped and are not required in order to understand the rest of the book.

While the material in this book is arranged in a specific order, it doesn't have to be read in that order. Many of the chapters can be read independently of one another, even though later chapters of the book sometimes refer to concepts an examples that have been introduced earlier.

The following provides guidance for selecting chapters to read and the order in which to read them. It also provides exits and shortcuts that allow readers to skip ahead while reading chapters. While I discuss the computing concepts via events, people, and objects that occur in stories, I also sometimes introduce new notation and work through examples to demonstrate important aspects in more detail. Hence, some parts of the book are easier to follow than others. As a reader of many popular science books, I am well aware of the fact that one's appetite for such details may vary. That is why I hope this guidance will help the reader to better navigate the book's content.

Figure 2 Concepts of computation and how they are related.

I suggest reading chapters 1 and 2 first, since they introduce fundamental concepts of computing that appear throughout the book, such as algorithm, parameter, computer, and runtime complexity. These two chapters should be easy to follow.

The other six topic areas (light-shaded boxes in figure 2) are largely independent of one another, but within each topic area the chapters should be read in order. Chapter 4 introduces several data structures, so it should be read before chapters 5, 6, 8, 12, and 13 (see diagram).

$$\begin{tikzcd}
& 5 \arrow[r, dotted] & 6 \arrow[dr, dotted] & \\
4 \arrow[ur, dotted] \arrow[rr, dotted] \arrow[ddr, dotted] & & 8 \arrow[r, dotted] & 13 \\
& & 10 \arrow[ur, dotted] & \\
& 12 \arrow[uurr, dotted] & &
\end{tikzcd}$$

Finally, the Glossary at the end of the book provides additional information about how the chapters relate by grouping definitions into topic areas and cross-linking their entries.

Computation and Algorithms

Hansel and Gretel

Getting up

with well-defined answers. A *problem* is thus any question or situation that demands a solution, even if it is clear how to get it. In this sense, having to get up in the morning is a problem that has a well-known method for producing a solution.

Once we know how to solve a problem, we seldom wonder how the corresponding method was conceived. In particular, when the method is obvious and simple to use, there seems to be no point in reflecting on it. But thinking about *how* we can solve problems can help us solve unknown problems in the future. Solutions to problems are not always obvious. In hindsight, most solutions seem evident, but what if you didn't know how to solve the getting-up problem. How would you approach it?

One crucial observation is that nontrivial problems can be decomposed into subproblems and that the solutions to the subproblems can be combined into a solution for the original problem. The getting-up problem consists of the two subproblems: getting out of bed and getting dressed. We have algorithms for solving both of these problems, that is, moving our body out of the bed and putting on clothes, respectively, and we can combine these algorithms into an algorithm for getting up, but we have to be careful to do it in the right order. Since it is rather difficult to get dressed in bed, we should take the step of getting out of bed first. If you don't find this example convincing, think about the order in which to take a shower and to get dressed. In this simple example there is only one ordering of the steps that leads to a viable solution, but this is not always the case.

Problem decomposition is not limited to one level. For example, the problem of getting dressed can be further decomposed into several subproblems, such as putting on pants, a shirt, shoes, and so on. The nice thing about problem decomposition is that it helps us to modularize the process of finding solutions, which means that solutions to different subproblems can be developed independently of one another. Modularity is important because it enables the parallel development of solutions to problems by teams.

Finding an algorithm for solving a problem is not the end of the story. The algorithm has to be put to work to actually solve the problem. Knowing how to do something and actually doing it are two different things. Some of us are made painfully aware of this difference each morning when the alarm clock rings. Thus there is a difference between an algorithm and its use.

In computer science each use of an algorithm is called a *computation*. So does that mean, when we actually do get up, that we are computing ourselves out of bed and into our clothes? That sounds crazy, but what would we say about a robot that did the same thing? The robot needs to be programmed to accomplish the task. In other words, the robot is told the algorithm in a language it understands. When the robot executes its program to get up, wouldn't we say it carries out a computation? This is not to say that humans are robots, but it does show that people compute when they execute algorithms.

The power of algorithms derives from the fact that they can be executed many times. Like the proverbial wheel that should not be reinvented, a good algorithm, once developed, is there to stay and serve us forever. It will be reused by many people in many situations to reliably compute solutions to recurring problems. This is why algorithms play a central role in computer science and why the design of algorithms is one of the most important and exciting tasks for computer scientists.

Computer science could be called the science of problem solving. Even though you won't find this definition in many textbooks, this perspective is a useful reminder of why computer science affects more and more areas of our lives. Moreover, many useful computations happen outside of machines and are performed by people (without a computer science education) to solve problems. Chapter 1 introduces the concept of computation and, through the story of Hansel and Gretel, highlights its problem-solving and human aspects.

1

A Path to Understanding Computation

What is computation? This question lies at the core of computer science. This chapter provides an answer—at least a tentative one—and connects the notion of computation to some closely related concepts. In particular, I explain the relationship between computation and the concepts of problem solving and algorithms. To this end, I describe two complementary aspects of computation: what it does, and what it is.

The first view, *computation solves problems*, emphasizes that a problem can be solved through computation once it is suitably represented and broken down into subproblems. It not only reflects the tremendous impact computer science has had in so many different areas of society but also explains why computation is an essential part of all kinds of human activities, independent of the use of computing machines.

However, the problem-solving perspective leaves out some important aspects of computation. A closer look at the differences between computation and problem solving leads to a second view, *computation is algorithm execution*. An algorithm is a precise description of computation and makes it possible to automate and analyze computation. This view portrays computation as a process consisting of several steps, which helps explain how and why it is so effective in solving problems.

The key to harnessing computation lies in grouping similar problems into one class and designing an algorithm that solves each problem in that class. This makes an algorithm similar to a skill. A skill such as baking a cake or repairing a car can be invoked at different times and thus can be employed repeatedly to solve different instances of a particular problem class. Skills can also be taught to and shared with others, which

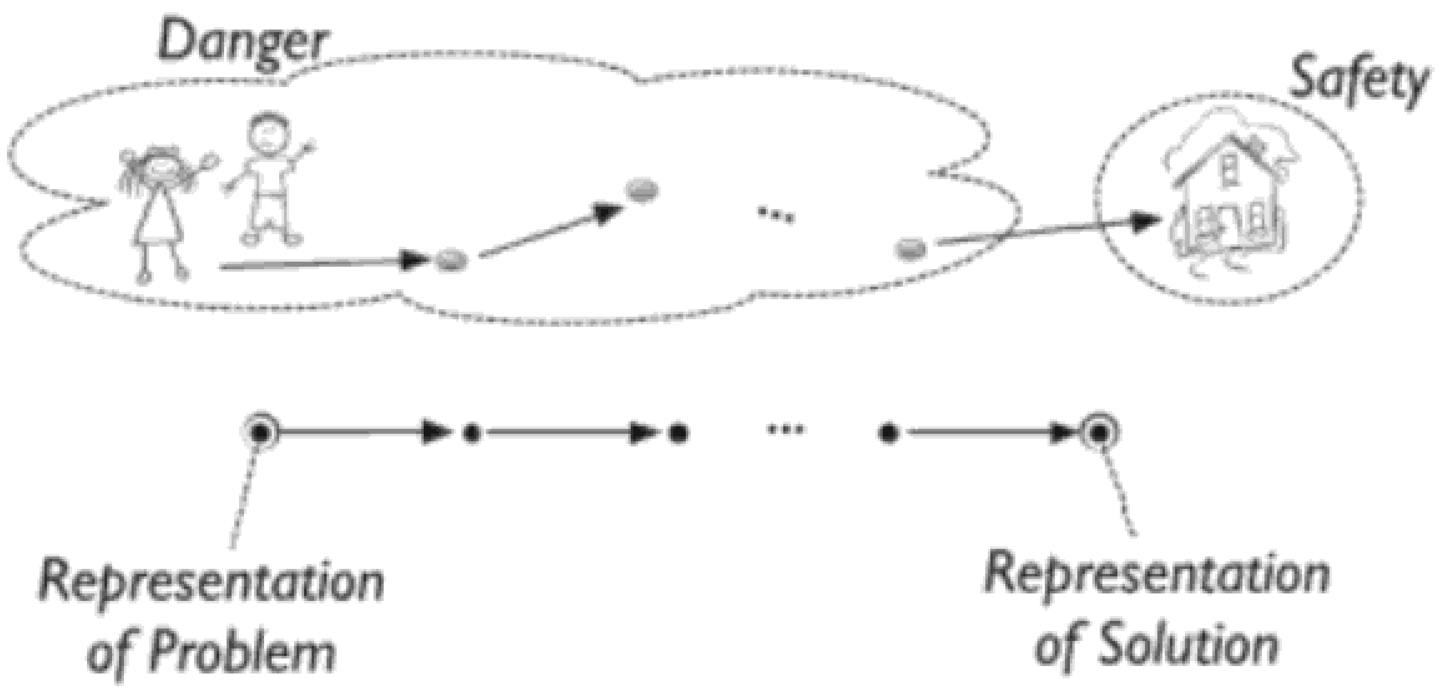


Figure 1.1 Computation is a process for solving a particular problem. Usually, a computation consists of several steps. Starting with a representation of the problem, each step transforms the representation until the solution is obtained. Hansel and Gretel solve the problem of surviving through a process of changing their position step-by-step and pebble-by-pebble from within the forest to their home.

computer, that is, Hansel and Gretel, to actually carry out the steps of the computation. It is common to have several layers of representation. In this case we have one that defines the problem (locations) and one that makes it possible to compute a solution (pebbles). In addition, all the pebbles taken together constitute another level of representation, since they represent the path out of the forest back home. These representations are summarized in table 1.1.

Figure 1.1 summarizes the problem-solving picture of computation; it shows Hansel and Gretel's way-finding as an instance of the view that computation manipulates representation in a sequence of steps. In the getting-up problem we also find representations, for example, as location (in bed, out of bed) and as an alarm clock representing time. Representations can take many different forms. They are discussed in more detail in chapter 3.

Table 1.1

Computation Representation		*Problem Representation*	
Object	**Represents**	**Concept**	**Represents**
One pebble	Location in forest	Location in forest	Danger
	Home	Home	Safety
All pebbles	Path out of forest	Path out of forest	Problem Solution

Beyond Problem Solving

Regarding computation as a problem-solving process captures the purpose of computation but doesn't explain what computation really *is*. Moreover, the problem-solving view has some limitations, since not every act of problem solving is a computation.

As illustrated in figure 1.2, there are computations, and there is problem solving. Although these often overlap, some computations do not solve problems, and some problems are not solved through computation. The emphasis in this book is on the intersection of computations and problem solving, but to make this focus clear, I consider some examples for the other two cases.

For the first case, imagine a computation consisting of following pebbles from one place to another within the forest. The steps of this process are in principle the same as in the original story, but the corresponding change of location would not solve Hansel and Gretel's problem of survival. As an even more drastic example, imagine the situation when the pebbles are arranged in a loop, which means the corresponding computation doesn't seem to achieve anything, since the initial and final positions are identical. In other words, the computation has no cumulative effect. The difference between these two cases and the one in the story is the meaning that is attached to the process.

Processes without such an apparent meaning still qualify as computation, but they may not be perceived as problem solving. This case is not hugely important, since we can always assign some arbitrary meaning to the representation operated on by a particular computation. Therefore, any computation could arguably be considered problem solving; it always depends on what meaning is associated with the representation. For example, following a loop inside a forest may not be helpful for Hansel and Gretel, but it could solve the problem of getting exercise for a runner. Thus, whether a computation solves a problem is in the eye of the beholder, that is, in the utility of a computation. In any case, whether or not one grants a particular computation the status of problem solving does not affect the essence of computation.

The situation is markedly different for noncomputational problem solving, since it provides us with further criteria for computation. In figure 1.2 two such criteria are mentioned, both of which are, in fact, very closely related. First, if a problem is solved in an ad hoc way that doesn't follow a particular method, it is not a computation. In other words, a computation has to be systematic. We can find several instances of this kind of noncomputational problem solving in the story. One example occurs when Hansel and Gretel are held captive by the witch who tries to fatten up Hansel with the goal of eating him. Since she can't see well, she estimates Hansel's weight by feeling his finger. Hansel misleads the witch about his weight by using a little bone in place of his

Figure 1.2 Distinguishing problem solving from computation. A computation whose effect carries no meaning in the real world does not solve any problems. An ad hoc solution to a problem that is not repeatable is not a computation.

finger. This idea is not the result of a systematic computation, but it solves the problem: it postpones the witch's eating Hansel.

Another example of noncomputational problem solving occurs right after Hansel and Gretel have returned home. The parents plan to lead them into the forest again the next day, but this time the stepmother locks the doors the night before to prevent Hansel from collecting any pebbles. The problem is that Hansel cannot get access to the pebbles that served him so well the first time and that he relied on for finding the way back home. His solution is to find a substitute in the form of breadcrumbs. The important point here is how Hansel arrives at this solution—he has an idea, a creative thought. A solution that involves a eureka moment is generally very difficult, if not impossible, to derive systematically through a computation because it requires reasoning on a subtle level about objects and their properties.

Unfortunately for Hansel and Gretel, the breadcrumbs solution doesn't work as expected:

> *When the moon came, they set out, but they found no crumbs, for the many thousands of birds which fly about in the woods and fields had picked them all up.*[1]

Since the breadcrumbs are gone, Hansel and Gretel can't find their way home, and the rest of the story unfolds.

However, let us assume for a moment that Hansel and Gretel could somehow have found their way back home again and that the parents would have tried a third time to leave them behind in the forest. Hansel and Gretel would have had to think of yet another means to mark their way home. They would have had to find something else to drop along the way, or maybe they would have tried to mark trees or bushes. Whatever the solution, it would have been produced by thinking about the problem and having another creative idea, but not by systematically applying a method. This highlights the other criterion for computation, which is its ability to be repeated and solve many similar problems. The method for solving the way-finding problem by following pebbles is different in this regard because it can be executed repeatedly for many different pebble placements.

To summarize, while the problem-solving perspective shows that computation is a systematic and decomposable process, it is not sufficient to give a comprehensive and precise picture of computation. Viewing computation as problem solving demonstrates how computation can be employed in all kinds of situations and thus illustrates its importance but ignores some important attributes that explain how computation works and why it can be successfully applied in so many different ways.

When Problems Reappear

Hansel and Gretel are faced with the problem of finding their way back home *twice*. Except for the practical problems caused by the lack of pebbles, the second instance could be solved in the same way as the first, namely, by following a series of markers. There is nothing really surprising about this fact, since Hansel and Gretel are simply applying a general method of way-finding. Such a method is called an *algorithm*.

Let us take a look at the algorithm that was used by Hansel and Gretel to find their way home. The exact method is not explained in detail in the original fairy tale. All we are told is the following:

> *And when the full moon had risen, Hansel took his little sister by the hand, and followed the pebbles which shone like newly-coined silver pieces, and showed them the way.*

A simple algorithm that fits this characterization is given, for example, by the following description:

> Find a shining pebble that was not visited before, and go toward it.
> Continue this process until you reach your parents' house.

An important property of an algorithm is that it can be used repeatedly by the same or a different person to solve the same or a closely related problem. An algorithm that generates a computation with a physical effect is useful even if it solves only one specific problem. For example, a recipe for a cake will produce the same cake over and over again. Since the output of the algorithm is transient—the cake gets eaten—reproducing the same result is very useful. The same holds for the problem of getting out of bed and dressed; the effect of the algorithm has to be reproduced every day, although likely with different clothes and at a different time on weekends. This also applies to Hansel and Gretel. Even if they are brought to the same place in the forest as on the first day, getting home has to be recomputed by repeating the algorithm to solve the exact same problem.

The situation is different for algorithms that produce nonphysical, abstract results, such as numbers. In such a case one can simply write down the result and look it up the next time it is needed instead of executing the algorithm again. For an algorithm to be useful in such situations it must be able to solve a whole class of problems, which means that it must be possible to apply the method to several different, but related, problems.[2]

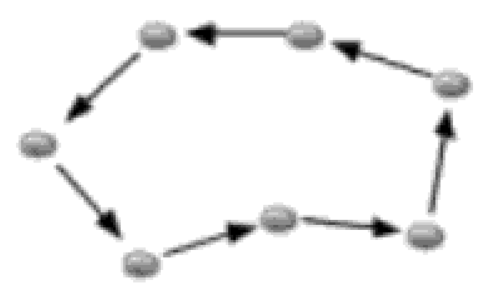
In the story the method is general enough to solve many different way-finding problems, since the exact positions of the pebbles do not matter. No matter where exactly in the forest the parents lead the children to, the algorithm will work in every case[3] and consequently will cause a computation that solves Hansel and Gretel's survival problem. Much of the power and impact of algorithms comes from the fact that *one* algorithm gives rise to *many* computations.

The notion of an algorithm is one of the most important concepts in computer science because it provides the foundation for the systematic study of computation. Accordingly, many different aspects of algorithms are discussed throughout this book.

Do You Speak "Algorithmish"?

An algorithm is a description of how to perform a computation and must therefore be formulated in some language. In the story the algorithm is only marginally mentioned. Hansel certainly has the algorithm in his head and might have told Gretel about it, but the algorithm is not written down as part of the story. However, the fact that an algorithm can be written down is an important property because it enables the reliable sharing of the algorithm and thus allows many people to use it to solve problems. The ability to express an algorithm in some language supports the proliferation of computation because instead of one person producing many computations, it facilitates many people's producing even more computations. If the language in which the algorithm is expressed can be understood by a computing shows that the behavior of an algorithm is not always easy to predict, which makes the design of algorithms a challenging and interesting endeavor.

The termination of algorithms is not an easy-to-recognize property either. If we remove the condition in the algorithm to find only nonvisited pebbles, a computation can easily slip into a nonterminating back-and-forth movement between two pebbles. One might object that Hansel and Gretel would never do such a silly thing and would recognize such a repeating pattern. This might be true, but then they would not be following the exact algorithm and would, in fact, be deliberately avoiding a previously visited pebble.



Whereas the case of a nonterminating back-and-forth between two pebbles would be easy to spot, the problem can be much more difficult generally. Imagine a path taken by the parents into the forest that crosses itself a few times. The resulting pebble placement would include several loops, each of which Hansel and Gretel could be caught in; only by remembering visited pebbles could they be certain to avoid such loops. Chapter 11 considers the problem of termination in more detail.

Questions of correctness and termination do not seem that important for the getting-up algorithm, but people have been known to put on nonmatching socks or to violate the rules for correctly buttoning a shirt. And if you persist in repeatedly hitting the snooze button, the getting-up algorithm will not even terminate.

Starting the Day

Most people's day doesn't really start before they've had breakfast. Cereal, fruit, eggs with bacon, juice, coffee—whatever is on the menu, chances are that breakfast needs some form of preparation. Some of these preparations can be described by algorithms.

If you like to vary your breakfast, for example, by adding different toppings to your cereal or brewing different amounts of coffee, the algorithm that describes the preparation must be able to reflect this flexibility. The key to providing controlled variability is to employ one or more placeholders, called *parameters*, that are replaced by concrete values whenever the algorithm is executed. Using different values for a placeholder causes the algorithm to produce different computations. For example, a parameter "fruit" can be substituted by different fruits on different days, allowing the execution of the algorithm to produce blueberry cereal as well as banana cereal. The getting-up algorithm also contains parameters so that we are not forced to wake up at the same time and wear the same shirt every day.

If you are grabbing coffee from a coffee shop on your way to work, or if you are ordering breakfast in a restaurant, algorithms are still employed in producing your breakfast. It is just that other people have to do the work for you. The person or machine that is executing an algorithm is called a *computer* and has a profound effect on the outcome of a computation. It may happen that a computer is unable to execute the algorithm if it doesn't understand the language in which the algorithm is given or if it is unable to perform one of its steps. Imagine you are a guest on a farm, and the algorithm for getting your morning milk involves your milking a cow. This step might prove prohibitive.

But even if a computer can execute all the steps of an algorithm, the time it takes to do so matters. In particular, the execution time can vary substantially between different computers. For example, an experienced cow milker can extract a glass of milk faster than a novice. However, computer science mostly ignores these differences because they are transient and not very meaningful, since the speed of electronic computers increases over time—and novice cow milkers can get more experienced and thus faster. What is of great importance, however, is the difference in execution time between different algorithms for the same problem. For example, if you want to get a glass of milk for everyone in your family, you could fetch each glass separately, or you could fetch a milk can once and then fill all glasses at the breakfast table. In the latter case you have to walk the distance to the stable only twice, whereas in the former case you have to walk it ten times for a family of five. This difference between the two algorithms exists independently of how fast you can milk or walk. It is thus an indicator for the complexity of the two algorithms and can be the basis for choosing between them.

In addition to execution time, algorithms may also differ with regard to other resources that are required for their execution. Say your morning drink is coffee and not milk, and you have the option to brew coffee with a coffee maker or use a French press. Both methods require water and ground coffee, but the first method additionally requires coffee filters. The resource requirements for different milking algorithms are even more pronounced. Getting fresh milk requires a cow, whereas milk bought at a grocery store requires a fridge for storing it. This example also shows that computation results can be saved for later use and that computation can be sometimes traded for storage space. We can save the effort of milking a cow by storing previously drawn milk in a fridge.

The execution of an algorithm has to pay for its effect with the use of resources. Therefore, in order to compare different algorithms for the same problem, it is important to be able to measure the resources they consume. Sometimes we may even want to sacrifice correctness for efficiency. Suppose you are on your way to your office and you have to grab a few items from the grocery store. Since you are in a hurry, you leave the change behind instead of storing the coins returned to you. The correct algorithm would exchange the exact amount of money for the bought items, but the approximation algorithm that rounds up finishes your transaction faster.

Studying the properties of algorithms and their computations, including the resource requirements, is an important task of computer science. It facilitates judging whether a particular algorithm is a viable solution to a specific problem. Continuing with the story about Hansel and Gretel, I explain in chapter 2 how different computations can be produced with one algorithm and how to measure the resources required.

2

Walk the Walk: When Computation Really Happens

In the previous chapter we saw how Hansel and Gretel solved a problem of survival by computing their way back home. This computation transformed their position systematically in individual steps, and it solved the problem by moving from a position in the forest, representing danger, to the final position at home, representing safety. The back-to-home computation was the result of executing an algorithm to follow a path of pebbles. Computation happens when algorithms get to work.

While we now have a good picture of what computation *is*, we have only seen one aspect of what computation actually *does*, namely, the transformation of representation. But there are additional details that deserve attention. Therefore, to expand understanding beyond the *static* representation through algorithms, I discuss the *dynamic* behavior of computation.

The great thing about an algorithm is that it can be used repeatedly to solve different problems. How does this work? And how is it actually possible that one fixed algorithm description can produce different computations? Moreover, we have said that computation results from executing an algorithm, but who or what is executing the algorithm? What are the skills needed to execute an algorithm? Can anybody do it? And finally, while it is great to have an algorithm for solving a problem, we have to ask at what cost. Only when an algorithm can deliver a solution to a problem fast enough and with the resources allocated is it a viable option.

Figure 2.1 The execution of an algorithm generates a computation. The algorithm describes a method that works for a whole class of problems, and an execution operates on the representation of a particular example problem. The execution must be performed by a computer, for example, a person or a machine, that can understand the language in which the algorithm is given.

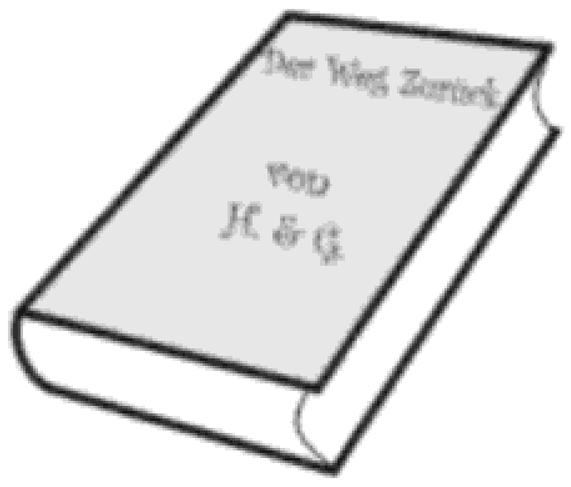
The word for somebody or something that can perform a computation is, of course, *computer*. In fact, the original meaning of the word referred to people who carried out computations.[2] In the following, I use the term in a general sense that refers to any natural or artificial agent that can perform computations.

We can distinguish two principal cases of algorithm execution based on the abilities of the performing computer. On the one hand, there are universal computers, such as people or laptops or smart phones. A universal computer can in principle execute any algorithm as long as it is expressed in a language understandable by the computer. Universal computers establish an execution relationship between algorithms and computation. Whenever the computer executes an algorithm for a particular problem, it performs steps that change some representation (see figure 2.1).

On the other hand, there are computers that execute just one algorithm (or perhaps a set of predefined algorithms). For example, a pocket calculator contains hard-wired electronic circuits for executing algorithms to perform arithmetic computations, as does an alarm clock for making sounds at particular times. Another intriguing example can be found in cell biology.

Think about what happens in your cells millions of times while you are reading this sentence. Ribosomes are producing proteins to support your cells' functions. Ribosomes are little machines that assemble proteins as described by RNA molecules. These are sequences of amino acids that tell the ribosome to produce specific proteins. You are alive thanks to the computation that results from the ribosomal computers in your cells reliably executing an algorithm to translate RNA molecules into proteins. But even though the algorithm used by a ribosome can produce a huge variety of proteins, this is the only algorithm that ribosomes can ever execute. While very useful, ribosomes are limited; they cannot get you dressed or find a way out of the forest.

In contrast to computers that consist of hard-wired algorithms, an important requirement for universal computers is that they understand the language in which their algorithms are given. If the computer is a machine, the algorithm is also called a *program*, and the language in which it is given is called a *programming language*.



If Hansel and Gretel wrote their memoirs and included a description of the algorithm that saved their lives, other children with access to that book could execute the algorithm only if they understood the language in which the book was written. This requirement does not apply to nonuniversal computers that carry out only fixed, hard-wired algorithms.

A requirement that applies to every kind of computer is the ability to access the representation that is used by the algorithm. In particular, a computer must be able to affect the required changes to the representation. If Hansel and Gretel were chained to a tree, the algorithm would be of no help to them, since they wouldn't be able to change their position, which is what an execution of the way-finding algorithm requires.

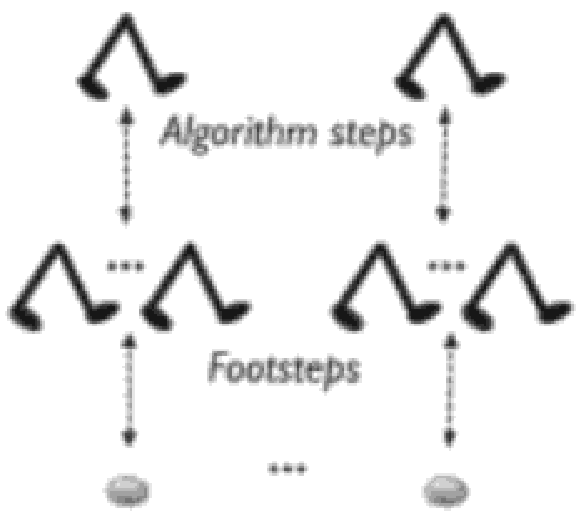
To summarize, any computer has to be able to read and manipulate the representations that the algorithms operate on. In addition, a universal computer has to understand the language in which algorithms are expressed. From now on, I use the term computer to mean a universal computer.

The Cost of Living

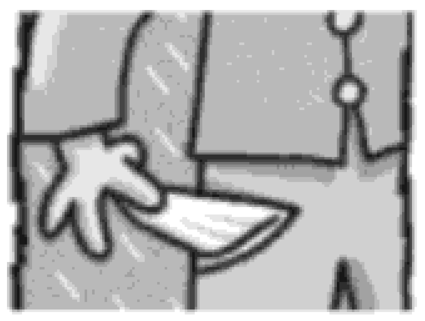
A computer has some real work to do, a fact one is reminded of whenever the laptop gets hot while rendering high-end graphics in a video game or when the smart phone battery drains too quickly with too many apps running in the background. And the reason you have to set your alarm clock considerably earlier than the time of your first appointment is that the execution of the getting-up algorithm takes some time.

Having figured out an algorithm to solve a problem is one thing, but ensuring that an actual computation generated by the algorithm will produce a solution quickly enough is an entirely different matter. Related is the question of whether the computer in charge has enough resources available to perform the computation in the first place.

For example, when Hansel and Gretel follow the pebble trace back home, the entire computation takes as many steps as the number of pebbles that Hansel dropped.[3] Note that step here means "step of the algorithm" and not "step as a part of walking." In particular, one step of the algorithm generally corresponds to several steps taken by Hansel and Gretel through the forest. Thus the number of pebbles is a measure for the execution time of the algorithm, since one step of the algorithm is required for each pebble. Thinking about the number of steps an algorithm needs to perform its task is judging its *runtime complexity*.



Moreover, the algorithm works only if Hansel and Gretel have enough pebbles to cover the path from their home to the place in the forest where they are left by their parents. This is an example of a resource constraint. A shortfall of pebbles could be due either to a limited availability of pebbles, which would be a limit of external resources, or to the limited space offered by Hansel's pockets, which would be a limit of the computer. To judge an algorithm's *space complexity* means to ask how much space a computer needs to execute the algorithm. In the example, this amounts to asking how many pebbles are needed to find a path of a particular length and whether Hansel's pockets are big enough to carry them all.



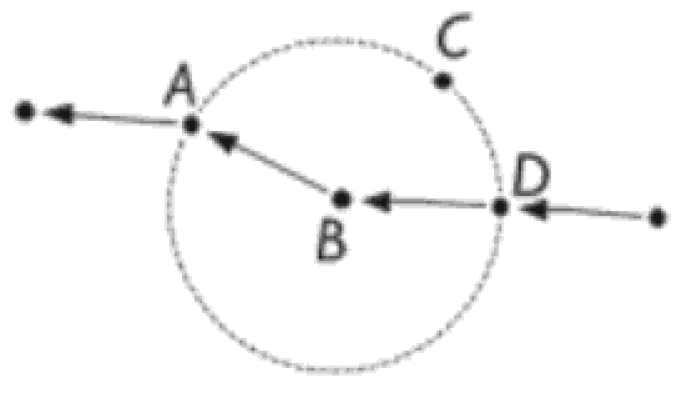
Therefore, while the algorithm may work in theory for any place in the forest, it is not clear ahead of time that a computation will succeed in practice because it may take too much time or require an amount of resources exceeding what is available. Before examining computation resources more closely, I explain two important assumptions about measuring computing costs that make this kind of analysis practical at all. In the following, I focus on the runtime aspect, but the discussion also applies to the question of space resources.

The Big Picture of Costs

An algorithm can be viewed as a generalization of many computations. As explained earlier, the differences in the computations are captured in the algorithm description by parameters, and any particular computation can be obtained through an execution of the algorithm with particular input values substituted for the parameters. In the same way, we would like to obtain a generalized description of the resource requirements of an algorithm—a description that does not just apply to a particular computation but captures all computations. In other words, we are looking for a generalization of cost descriptions. This generalization can be achieved by using parameters that can make the number of steps required for executing an algorithm dependent on the size of its input. Thus runtime complexity is a function that yields the number of steps in a computation for an input of a given size.

For example, the number of computation steps, and thus the time, required to execute the pebble-tracing algorithm is roughly equal to the number of dropped pebbles. Since paths to different places in the forest generally contain different numbers of pebbles, computations for these paths also require different numbers of steps. This fact is reflected in expressing runtime complexity as a function of the size of input. For the pebble-tracing algorithm it is easy to derive a precise measure for each computation, since the number of computation steps seems to be in one-to-one correspondence with the number of pebbles. For example, for a path with 87 pebbles, the computation requires 87 steps.

However, this is *not* always the case. Take another look at the path depicted in figure 1.3. That example was used to illustrate how the algorithm can get stuck, but we can also use it to demonstrate how the algorithm can produce computations that take fewer steps than there are pebbles. Since B and C are visible from D, we can pick B, and since both A and C are visible from B, we can next pick A, that is, the path D, B, A is a valid path, and since it bypasses C, the computation contains at least one less step than the number of pebbles in the path.



Note that in this case the number of steps in the computation is actually *lower* than predicted by the measure for the algorithm, which means that the cost of the computation is overestimated. Thus, runtime complexity reports the complexity a computation can have in the *worst case*. This helps us decide whether or not to execute the algorithm for a particular input. If the estimated runtime is acceptable, then the algorithm can be executed. If the computation actually performs faster and takes fewer steps, then all the better, but the worst-case complexity provides a guarantee that the algorithm will not take more time. In the case of getting up, your worst-case estimate for the time it takes to take a shower in the morning might be 5 minutes. Since this includes the time it takes for the water to get warm, your actual shower time might be shorter if someone has taken a shower before you.

Since runtime analysis is applied on the level of algorithms, only algorithms (not individual computations) are amenable to an analysis. This also means that it is possible