# 1 Introduction

*Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.*

—David Berlinski, *The Advent of the Algorithm,* 2000

Why do you need to study algorithms? If you are going to be a computer professional, there are both practical and theoretical reasons to study algorithms. From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency. From the theoretical standpoint, the study of algorithms, sometimes called ***algorithmics***, has come to be recognized as the cornerstone of computer science. David Harel, in his delightful book pointedly titled *Algorithmics: the Spirit of Computing*, put it as follows:

> Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology. [Har92, p. 6]

But even if you are not a student in a computing-related program, there are compelling reasons to study algorithms. To put it bluntly, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems—not just answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved. Of course, the precision inherently imposed by algorithmic thinking limits the kinds of problems that can be solved with an algorithm. You will not find, for example, an algorithm for living a happy life or becoming rich and famous. On the other hand, this required precision has an important educational advantage. Donald Knuth, one of the most prominent computer scientists in the history of algorithmics, put it as follows:

> A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way. [Knu96, p. 9]

We take up the notion of algorithm in Section 1.1. As examples, we use three algorithms for the same problem: computing the greatest common divisor. There are several reasons for this choice. First, it deals with a problem familiar to everybody from their middle-school days. Second, it makes the important point that the same problem can often be solved by several algorithms. Quite typically, these algorithms differ in their idea, level of sophistication, and efficiency. Third, one of these algorithms deserves to be introduced first, both because of its age—it appeared in Euclid's famous treatise more than two thousand years ago—and its enduring power and importance. Finally, investigation of these three algorithms leads to some general observations about several important properties of algorithms in general.

Section 1.2 deals with algorithmic problem solving. There we discuss several important issues related to the design and analysis of algorithms. The different aspects of algorithmic problem solving range from analysis of the problem and the means of expressing an algorithm to establishing its correctness and analyzing its efficiency. The section does not contain a magic recipe for designing an algorithm for an arbitrary problem. It is a well-established fact that such a recipe does not exist. Still, the material of Section 1.2 should be useful for organizing your work on designing and analyzing algorithms.

Section 1.3 is devoted to a few problem types that have proven to be particularly important to the study of algorithms and their application. In fact, there are textbooks (e.g., [Sed11]) organized around such problem types. I hold the view—shared by many others—that an organization based on algorithm design techniques is superior. In any case, it is very important to be aware of the principal problem types. Not only are they the most commonly encountered problem types in real-life applications, they are used throughout the book to demonstrate particular algorithm design techniques.

Section 1.4 contains a review of fundamental data structures. It is meant to serve as a reference rather than a deliberate discussion of this topic. If you need a more detailed exposition, there is a wealth of good books on the subject, most of them tailored to a particular programming language.

## 1.1 What Is an Algorithm?

Although there is no universally agreed-on wording to describe this notion, there is general agreement about what the concept means:

> An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure 1.1).

The reference to “instructions” in the definition implies that there is something or someone capable of understanding and following the instructions given. We call this a “computer,” keeping in mind that before the electronic computer was invented, the word “computer” meant a human being involved in performing numeric calculations. Nowadays, of course, “computers” are those ubiquitous electronic devices that have become indispensable in almost everything we do. Note, however, that although the majority of algorithms are indeed intended for eventual computer implementation, the notion of algorithm does not depend on such an assumption.

As examples illustrating the notion of the algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

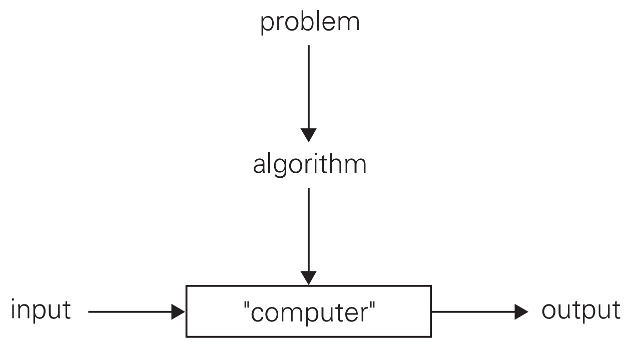

**FIGURE 1.1** The notion of the algorithm.

- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Recall that the greatest common divisor of two nonnegative, not-both-zero integers $m$ and $n$, denoted $\gcd(m, n)$, is defined as the largest integer that divides both $m$ and $n$ evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem in one of the volumes of his *Elements* most famous for its systematic exposition of geometry. In modern terms, ***Euclid's algorithm*** is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n),$$

where $m \bmod n$ is the remainder of the division of $m$ by $n$, until $m \bmod n$ is equal to 0. Since $\gcd(m, 0) = m$ (why?), the last value of $m$ is also the greatest common divisor of the initial $m$ and $n$.

For example, $\gcd(60, 24)$ can be computed as follows:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

(If you are not impressed by this algorithm, try finding the greatest common divisor of larger numbers, such as those in Problem 6 in this section's exercises.)

Here is a more structured description of this algorithm:

**Euclid's algorithm** for computing $\gcd(m, n)$

**Step 1** If $n = 0$, return the value of $m$ as the answer and stop; otherwise, proceed to Step 2.

**Step 2** Divide $m$ by $n$ and assign the value of the remainder to $r$.

**Step 3** Assign the value of $n$ to $m$ and the value of $r$ to $n$. Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

```
ALGORITHM    Euclid(m, n)
    //Computes gcd(m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
    return m
```

How do we know that Euclid's algorithm eventually comes to a stop? This follows from the observation that the second integer of the pair gets smaller with each iteration and it cannot become negative. Indeed, the new value of $n$ on the next iteration is $m \bmod n$, which is always smaller than $n$ (why?). Hence, the value of the second integer eventually becomes 0, and the algorithm stops.

Just as with many other problems, there are several algorithms for computing the greatest common divisor. Let us look at the other two methods for this problem. The first is simply based on the definition of the greatest common divisor of $m$ and $n$ as the largest integer that divides both numbers evenly. Obviously, such a common divisor cannot be greater than the smaller of these numbers, which we will denote by $t = \min\{m, n\}$. So we can start by checking whether $t$ divides both $m$ and $n$: if it does, $t$ is the answer; if it does not, we simply decrease $t$ by 1 and try again. (How do we know that the process will eventually stop?) For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on, until it reaches 12, where it stops.

**Consecutive integer checking algorithm** for computing $\gcd(m, n)$

**Step 1** Assign the value of $\min\{m, n\}$ to $t$.
**Step 2** Divide $m$ by $t$. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
**Step 3** Divide $n$ by $t$. If the remainder of this division is 0, return the value of $t$ as the answer and stop; otherwise, proceed to Step 4.
**Step 4** Decrease the value of $t$ by 1. Go to Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the set of an algorithm's inputs explicitly and carefully.

The third procedure for finding the greatest common divisor should be familiar to you from middle school.

**Middle-school procedure** for computing $\gcd(m, n)$

**Step 1** Find the prime factors of $m$.
**Step 2** Find the prime factors of $n$.
**Step 3** Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If $p$ is a common factor occurring $p_m$ and $p_n$ times in $m$ and $n$, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)
**Step 4** Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$\begin{aligned} 60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\ 24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ \gcd(60, 24) &= 2 \cdot 2 \cdot 3 = 12. \end{aligned}$$

Nostalgia for the days when we learned this method should not prevent us from noting that the last procedure is much more complex and slower than Euclid's algorithm. (We will discuss methods for finding and comparing running times of algorithms in the next chapter.) In addition to inferior efficiency, the middle-school procedure does not qualify, in the form presented, as a legitimate algorithm. Why? Because the prime factorization steps are not defined unambiguously: they require a list of prime numbers, and I strongly suspect that your middle-school math teacher did not explain how to obtain such a list. This is not a matter of unnecessary nitpicking. Unless this issue is resolved, we cannot, say, write a program implementing this procedure. Incidentally, Step 3 is also not defined clearly enough. Its ambiguity is much easier to rectify than that of the factorization steps, however. How would you find common elements in two sorted lists?

So, let us introduce a simple algorithm for generating consecutive primes not exceeding any given integer $n > 1$. It was probably invented in ancient Greece and is known as the ***sieve of Eratosthenes*** (ca. 200 B.C.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to $n$. Then, on its first iteration, the algorithm eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| **2** | 3 | | 5 | | 7 | | 9 | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 | | 23 | | 25 |
| 2 | **3** | | 5 | | 7 | | | | 11 | | 13 | | | | 17 | | 19 | | | | 23 | | 25 |
| 2 | 3 | | **5** | | 7 | | | | 11 | | 13 | | | | 17 | | 19 | | | | 23 | | |

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

What is the largest number $p$ whose multiples can still remain on the list to make further iterations of the algorithm necessary? Before we answer this question, let us first note that if $p$ is a number whose multiples are being eliminated on the current pass, then the first multiple we should consider is $p \cdot p$ because all its smaller multiples $2p, \ldots, (p-1)p$ have been eliminated on earlier passes through the list. This observation helps to avoid eliminating the same number more than once. Obviously, $p \cdot p$ should not be greater than $n$, and therefore $p$ cannot exceed $\sqrt{n}$ rounded down (denoted $\lfloor \sqrt{n} \rfloor$ using the so-called ***floor function***). We assume in the following pseudocode that there is a function available for computing $\lfloor \sqrt{n} \rfloor$; alternatively, we could check the inequality $p \cdot p \le n$ as the loop continuation condition there.

```
ALGORITHM  Sieve(n)
    //Implements the sieve of Eratosthenes
    //Input: A positive integer $n > 1$
    //Output: Array $L$ of all prime numbers less than or equal to $n$
```

```
for $p \leftarrow 2$ to $n$ do $A[p] \leftarrow p$
for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ do    //see note before pseudocode
    if $A[p] \neq 0$             //$p$ hasn't been eliminated on previous passes
        $j \leftarrow p * p$
        while $j \leq n$ do
            $A[j] \leftarrow 0$    //mark element as eliminated
            $j \leftarrow j + p$
//copy the remaining elements of $A$ to array $L$ of the primes
$i \leftarrow 0$
for $p \leftarrow 2$ to $n$ do
    if $A[p] \neq 0$
        $L[i] \leftarrow A[p]$
        $i \leftarrow i + 1$
return $L$
```

So now we can incorporate the sieve of Eratosthenes into the middle-school procedure to get a legitimate algorithm for computing the greatest common divisor of two positive integers. Note that special care needs to be exercised if one or both input numbers are equal to 1: because mathematicians do not consider 1 to be a prime number, strictly speaking, the method does not work for such inputs.

Before we leave this section, one more comment is in order. The examples considered in this section notwithstanding, the majority of algorithms in use today—even those that are implemented as computer programs—do not deal with mathematical problems. Look around for algorithms helping us through our daily routines, both professional and personal. May this ubiquity of algorithms in today's world strengthen your resolve to learn more about these fascinating engines of the information age.

## Exercises 1.1

**1.** Do some research on al-Khorezmi (also al-Khwarizmi), the man from whose name the word "algorithm" is derived. In particular, you should learn what the origins of the words "algorithm" and "algebra" have in common.

**2.** Given that the official purpose of the U.S. patent system is the promotion of the "useful arts," do you think algorithms are patentable in this country? Should they be?

**3. a.** Write down driving directions for going from your school to your home with the precision required from an algorithm's description.

**b.** Write down a recipe for cooking your favorite dish with the precision required by an algorithm.

**4.** Design an algorithm for computing $\lfloor \sqrt{n} \rfloor$ for any positive integer $n$. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

5. Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are $m$ and $n$, respectively?

6. **a.** Find gcd(31415, 14142) by applying Euclid's algorithm.
   **b.** Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from $\min\{m, n\}$ down to $\gcd(m, n)$.

7. Prove the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ for every pair of positive integers $m$ and $n$.

8. What does Euclid's algorithm do for a pair of integers in which the first is smaller than the second? What is the maximum number of times this can happen during the algorithm's execution on such an input?

9. **a.** What is the minimum number of divisions made by Euclid's algorithm among all inputs $1 \le m, n \le 10$?
   **b.** What is the maximum number of divisions made by Euclid's algorithm among all inputs $1 \le m, n \le 10$?

10. **a.** Euclid's algorithm, as presented in Euclid's treatise, uses subtractions rather than integer divisions. Write pseudocode for this version of Euclid's algorithm.
    **b.** *Euclid's game* (see [Bog]) starts with two unequal positive integers on the board. Two players move in turn. On each move, a player has to write on the board a positive number equal to the difference of two numbers already on the board; this number must be new, i.e., different from all the numbers already on the board. The player who cannot move loses the game. Should you choose to move first or second in this game?

11. The ***extended Euclid's algorithm*** determines not only the greatest common divisor $d$ of two positive integers $m$ and $n$ but also integers (not necessarily positive) $x$ and $y$, such that $mx + ny = d$.
    **a.** Look up a description of the extended Euclid's algorithm (see, e.g., [KnuI, p. 13]) and implement it in the language of your choice.
    **b.** Modify your program to find integer solutions to the Diophantine equation $ax + by = c$ with any set of integer coefficients $a$, $b$, and $c$.

12. *Locker doors* There are $n$ lockers in a hallway, numbered sequentially from 1 to $n$. Initially, all the locker doors are closed. You make $n$ passes by the lockers, each time starting with locker #1. On the $i$th pass, $i = 1, 2, \ldots, n$, you toggle the door of every $i$th locker: if the door is closed, you open it; if it is open, you close it. After the last pass, which locker doors are open and which are closed? How many of them are open?

## 1.2 Fundamentals of Algorithmic Problem Solving

Let us start by reiterating an important point made in the introduction to this chapter:

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

### Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms. But often you will not find a readily available algorithm and will have to design your own. The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. (As an example, recall the variations in the set of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Do not skimp on this first step of the algorithmic problem-solving process; otherwise, you will run the risk of unnecessary rework.

### Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of

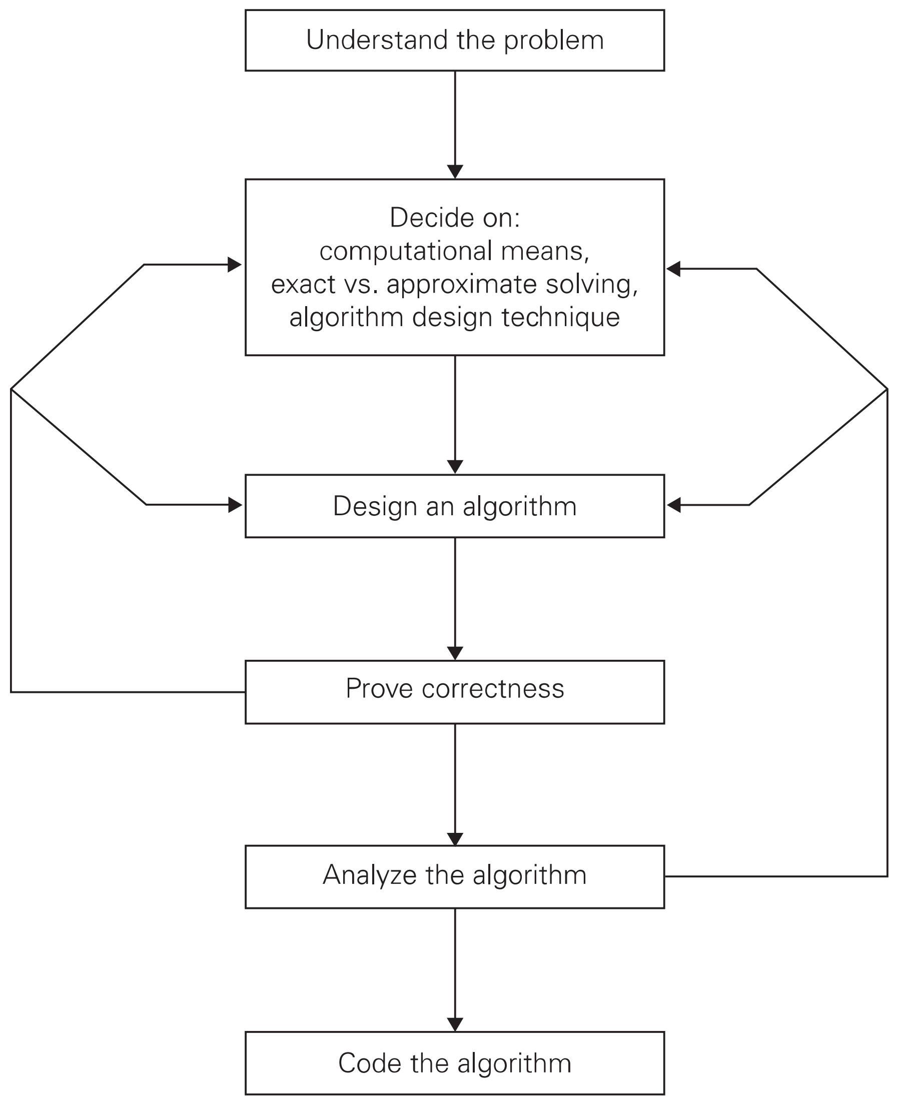

**FIGURE 1.2** Algorithm design and analysis process.

algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even the "slow" computers of today are almost unimaginably fast. Consequently, in many situations you need not worry about a computer being too slow for the task. There are important problems, however, that are very complex by their nature, or have to process huge volumes of data, or deal with applications where the time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

## Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices; you will see examples of such difficult problems in Chapters 3, 11, and 12. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

## Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques.

What is an algorithm design technique?

> An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Check this book's table of contents and you will see that a majority of its chapters are devoted to individual design techniques. They distill a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons.

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

## Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer. With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes introduced in Section 1.1 would run longer if we used a linked list instead of an array in its implementation (why?). Also note that some of the algorithm design techniques discussed in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

## Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, Euclid's algorithm is described in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

***Pseudocode*** is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors with a need to design their own “dialects.” Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

This book’s dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as **for**, **if**, and **while**. As you saw in the previous section, we use an arrow “$\leftarrow$” for the assignment operation and two slashes “//” for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm’s description—be it in a natural language or pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm’s implementation.

## Proving an Algorithm’s Correctness

Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid’s algorithm for computing the greatest common divisor stems from the correctness of the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm’s iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm’s performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm’s correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit. You can find examples of such investigations in Chapter 12.

## Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2.

Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing gcd$(m, n)$, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm. Still, simplicity is an important algorithm characteristic to strive for. Why? Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs. There is also the undeniable aesthetic appeal of simplicity. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

Yet another desirable characteristic of an algorithm is ***generality***. There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms. Consider, for example, the problem of determining whether two integers are relatively prime, i.e., whether their only common divisor is equal to 1. It is easier to design an algorithm for a more general problem of computing the greatest common divisor of two integers and, to solve the former problem, check whether the gcd is 1 or not. There are situations, however, where designing a more general algorithm is unnecessary or difficult or even impossible. For example, it is unnecessary to sort a list of $n$ numbers to find its median, which is its $\lceil n/2 \rceil$th smallest element. To give another example, the standard formula for roots of a quadratic equation cannot be generalized to handle polynomials of arbitrary degrees.

As to the set of inputs, your main concern should be designing an algorithm that can handle a set of inputs that is natural for the problem at hand. For example, excluding integers equal to 1 as possible inputs for a greatest common divisor algorithm would be quite unnatural. On the other hand, although the standard formula for the roots of a quadratic equation holds for complex coefficients, we would normally not implement it on this level of generality unless this capability is explicitly required.

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions. Recall the three different algorithms in the previous section for computing the greatest common divisor: generally, you should not expect to get the best algorithm on the first try. At the very least, you should try to fine-tune the algorithm you already have. For example, we made several improvements in our implementation of the sieve of Eratosthenes compared with its initial outline in Section 1.1. (Can you identify them?) You will do well if you keep in mind the following observation of Antoine de Saint-Exupéry, the French writer, pilot, and aircraft designer: “A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away.”[1]

## Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. They have developed special techniques for doing such proofs (see [Gri81]), but the power of these techniques of formal verification is limited so far to very small programs.

As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

Also note that throughout the book, we assume that inputs to algorithms belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm’s power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode. Still, you need to be aware of such standard tricks as computing a loop’s invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on. (See [Ker99] and [Ben00] for a good discussion of code tuning and other issues related to algorithm programming.) Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.

1. I found this call for design simplicity in an essay collection by Jon Bentley [Ben00]; the essays deal with a variety of issues in algorithm design and implementation and are justifiably titled *Programming Pearls*. I wholeheartedly recommend the writings of both Jon Bentley and Antoine de Saint-Exupéry.

A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. Such an analysis is based on timing the program on several inputs and then analyzing the results obtained. We discuss the advantages and disadvantages of this approach to analyzing algorithms in Section 2.6.

In conclusion, let us emphasize again the main lesson of the process depicted in Figure 1.2:

As a rule, a good algorithm is a result of repeated effort and rework.

Even if you have been fortunate enough to get an algorithmic idea that seems perfect, you should still try to see whether it can be improved.

Actually, this is good news since it makes the ultimate result so much more enjoyable. (Yes, I did think of naming this book *The Joy of Algorithms*.) On the other hand, how does one know when to stop? In the real world, more often than not a project's schedule or the impatience of your boss will stop you. And so it should be: perfection is expensive and in fact not always called for. Designing an algorithm is an engineering-like activity that calls for compromises among competing goals under the constraints of available resources, with the designer's time being one of the resources.

In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm's ***optimality***. Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves: What is the minimum amount of effort *any* algorithm will need to exert to solve the problem? For some problems, the answer to this question is known. For example, any algorithm that sorts an array by comparing values of its elements needs about $n \log_2 n$ comparisons for some arrays of size $n$ (see Section 11.2). But for many seemingly easy problems such as integer multiplication, computer scientists do not yet have a final answer.

Another important issue of algorithmic problem solving is the question of whether or not every problem can be solved by an algorithm. We are not talking here about problems that do not have a solution, such as finding real roots of a quadratic equation with a negative discriminant. For such cases, an output indicating that the problem does not have a solution is all we can and should expect from an algorithm. Nor are we talking about ambiguously stated problems. Even some unambiguous problems that must have a simple yes or no answer are "undecidable," i.e., unsolvable by any algorithm. An important example of such a problem appears in Section 11.3. Fortunately, a vast majority of problems in practical computing *can* be solved by an algorithm.

Before leaving this section, let us be sure that you do not have the misconception—possibly caused by the somewhat mechanical nature of the diagram of Figure 1.2—that designing an algorithm is a dull activity. There is nothing further from the truth: inventing (or discovering?) algorithms is a very creative and rewarding process. This book is designed to convince you that this is the case.

## Exercises 1.2

1. *Old World puzzle* A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)

2. *New World puzzle* There are four people who want to cross a rickety bridge; they all begin on the same side. You have 17 minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. Person 1 takes 1 minute to cross the bridge, person 2 takes 2 minutes, person 3 takes 5 minutes, and person 4 takes 10 minutes. A pair must walk together at the rate of the slower person's pace. (Note: According to a rumor on the Internet, interviewers at a well-known software company located near Seattle have given this problem to interviewees.)

3. Which of the following formulas can be considered an algorithm for computing the area of a triangle whose side lengths are given positive numbers $a$, $b$, and $c$?

   **a.** $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$

   **b.** $S = \frac{1}{2}bc \sin A$, where $A$ is the angle between sides $b$ and $c$

   **c.** $S = \frac{1}{2}ah_a$, where $h_a$ is the height to base $a$

4. Write pseudocode for an algorithm for finding real roots of equation $ax^2 + bx + c = 0$ for arbitrary real coefficients $a$, $b$, and $c$. (You may assume the availability of the square root function $sqrt(x)$.)

5. Describe the standard algorithm for finding the binary representation of a positive decimal integer

   **a.** in English.

   **b.** in pseudocode.

6. Describe the algorithm used by your favorite ATM machine in dispensing cash. (You may give your description in either English or pseudocode, whichever you find more convenient.)

7. **a.** Can the problem of computing the number $\pi$ be solved exactly?

   **b.** How many instances does this problem have?

   **c.** Look up an algorithm for this problem on the Internet.