every day. So the team proceeded to build and use a machine, Colossus, that could break the codes hundreds of times faster than mere humans.

The first general-purpose digital computers were built in the USA and Great Britain. You are probably familiar with the story of computers since then. Computers are now ubiquitous, whether as general-purpose computers or as components embedded in devices as diverse as digital watches, kitchen appliances, and aircraft. And every one of these computers is a machine designed to perform algorithms.

We are also surrounded by algorithms intended for performance by humans. Whether or not a device has a computer inside it, it probably comes with a user's manual. Such manuals contain algorithms necessary for us to use the devices effectively: to operate a washing machine, to set a digital watch to give an alarm signal or to record lap times, to change a car wheel, to change a light bulb, to assemble a piece of furniture from a flat-pack, and so on.

## 1.2 Algorithms and programs

So what exactly *is* an algorithm? We shall attempt a precise definition in Chapter 2. For the time being, we can think of an algorithm as a step-by-step procedure for solving a stated problem. An algorithm may be intended to be performed by a human or a machine. In either case, the algorithm must be broken down into steps that are simple enough to be performed by the human or machine.

Algorithms have many things in common with programs, but there are also important differences between them.

Firstly, algorithms are *more general* than programs. An algorithm may be suitable to be performed by a human, or by a machine, or by both. A program *must* be capable of being performed by a (suitable) machine. In this book, the machine will always be a general-purpose computer (as opposed to a robot or weaving machine, for example).

Secondly, algorithms are *more abstract* than programs. An algorithm can be expressed in any convenient language or notation. A program must be expressed in some programming language. If an algorithm is intended to be performed by a computer, it must first be coded in a programming language. There may be many ways of coding the algorithm and there is a wide choice of programming languages. Nevertheless, all the resulting programs are implementations of the same underlying algorithm. For example, Programs 1.7 and 1.8 are both implementations of Euclid's algorithm, coded in two different programming languages: compare them with Algorithm 1.3.

In this book we will express algorithms in English, and code them as methods in Java. The very same algorithms could be (and undoubtedly have been) equally well expressed in French or Chinese, and coded in C or Pascal or Ada. The point is that we can study these algorithms without paying too much attention to the language in which they are expressed.

Often there exist several different algorithms that solve the same problem. We are naturally interested in comparing such alternative algorithms. Which is fastest? Which needs least memory? Fortunately, we can answer such questions in terms of the qualities of the algorithms themselves, without being distracted by the languages in which they happen to be expressed. We shall study algorithm efficiency in Chapter 2.

Watt, D.A. & Brown, D.F., 2001, Java Collections. An Introduction to Abstract Data Types, Data Structures and Algorithms, Wiley.

# 2 Algorithms

In this chapter we shall study:

- fundamental principles of algorithms: problems that can or cannot be solved by algorithms, properties of algorithms themselves, and notation for algorithms (Section 2.1)
- analysis of algorithms to determine their time and space efficiency (Section 2.2)
- the notion of complexity of algorithms (Section 2.3)
- recursive algorithms and their complexity (Section 2.4).

## 2.1 Principles of algorithms

In Section 1.1 we encountered a variety of algorithms. In this section we briefly discuss some fundamental issues concerned with problems, algorithms, and notation.

### Problems

Concerning problems, we can state the following principles:

- An algorithm must be designed to solve a stated ***problem***, which is a well-defined task that has to be performed.
- The problem must be ***solvable*** by an algorithm.

We have already (in Section 1.1) encountered a number of problems that can be solved by algorithms. We can also pose some problems that are *not* solvable by algorithms. To say that a problem is unsolvable does not just mean that an algorithm has *not yet* been found to solve it. It means that such an algorithm can *never* be found. A human might eventually solve the problem, but only by applying insight and creativity, not by following a step-by-step procedure; moreover, there can be no guarantee that a solution will be found. Here is an example of a problem that is unsolvable by an algorithm.

**EXAMPLE 2.1** ***The halting problem***

The problem is to predict whether a given computer program, with given input data, will eventually halt.

This is a very practical problem for us programmers: we all occasionally write a program that gets into a never-ending loop. One of the most famous results in computer science is that this problem cannot be solved by any algorithm. It turns out that any 'algorithm' that purports to solve this problem will itself get into a never-ending loop, for at least some programs that might be given to it. As we shall see later in this section, we insist that every algorithm must eventually terminate.

If we can never find an algorithm to predict whether a given program halts with *given* input data, we clearly can never find an algorithm to prove whether a given program behaves correctly for *all possible* input data.

It may still be possible for a human to prove that a *particular* program is correct. Indeed, this has been done for some important small programs and subprograms. But we can never *automate* such proofs of correctness.

In fact, many problems in mathematics and computer science are unsolvable by algorithms. In a way, this is rather reassuring: we can be sure that mathematicians and computer scientists will never be made redundant by machines!

From now on, we shall consider only problems that are solvable by algorithms.

## Algorithms

Concerning algorithms themselves, we can state the following principles:

- The algorithm will be performed by some ***processor***, which may be a machine or a human.
- The algorithm must be expressed in steps that the processor is capable of performing.
- The algorithm must eventually terminate, producing the required answer.

Some algorithms, as we have already seen, are intended to be performed by humans rather than machines. But no algorithm is allowed to rely on qualities, such as insight and creativity, that distinguish humans from machines. This suggests a definition:

> An ***algorithm*** is an automatic procedure for solving a stated problem, a procedure that could (at least in principle) be performed by a machine.

The principle that the algorithm must be expressed in steps that can be performed by the processor should now be clear. If the processor has to work out for itself what steps to follow, then what we have is not an algorithm.

The principle that every algorithm must eventually terminate should also be clear. If it never terminates, it never produces an answer, therefore it is not an algorithm! So an algorithm must avoid getting into a never-ending loop.

## Notation

Concerning notation, we have one fundamental principle:

- The algorithm must be expressed in a language or notation that the processor 'understands'.

This principle should be self-evident. We cannot expect a weaving machine, or even a computer, to perform an algorithm expressed in natural language. A machine must be programmed in its own language.

On the other hand, an algorithm intended for humans need not necessarily be expressed in natural language. Special-purpose notations are commonly used for certain classes of algorithm. A musical score is an algorithm to be performed by a group of musicians, and is expressed in the standard musical notation. A knitting pattern is an algorithm for either a human or a knitting machine, and is generally expressed in a concise notation invented for the purpose.

Here we restrict our attention to computational algorithms. Even so, we have a variety of possible notations including natural language, programming language, mathematical notation, and combinations of these. In this book we shall express all algorithms in English, occasionally (and where appropriate) augmented by mathematical notation. The choice of a natural language gives us the greatest possible freedom of expression; both programming languages and mathematical notation are sometimes restrictive or inconvenient.

We should remember, however, that expressing an algorithm in a natural language always carries a risk of vagueness or even ambiguity. We must take great care to express the individual steps of the algorithm, and the order in which these steps are to be performed, as precisely as possible.

We have already seen several examples of algorithms, in Section 1.1, which you should now re-examine. Note the use of layout and numbering to show the structure of an algorithm. We number the steps consecutively, and arrange one below another in the intended order, e.g.:

```
1. Do this.
2. Do that.
3. Do the other.
```

We use indentation and the numbering system to show when one or more steps are to be performed only if some condition is satisfied:

```
1. If the condition is satisfied:
   1.1. Do this.
   1.2. Do that.
2. Carry on.
```

Likewise, we use indentation and the numbering system to show when one or more steps are to be performed repeatedly while (or until) some condition is satisfied:

```
1. While (or until) the condition is satisfied, repeat:
   1.1. Do this.
   1.2. Do that.
2. Carry on.
```

or when one or more steps are to be performed repeatedly as a variable $v$ steps through a sequence of values:

```
1. For $v$ = sequence of values, repeat:
   1.1. Do this.
   1.2. Do that.
2. Carry on.
```

## 2.2 Efficiency of algorithms

Given an algorithm, we are naturally interested in discovering how efficient it is. Efficiency has two distinct facets:

- ***Time efficiency*** is concerned with how much (processor) time the algorithm requires.
- ***Space efficiency*** is concerned with how much space (memory) the algorithm requires for storing data.

Often we have a choice of different algorithms that solve the same problem. How should we decide which of these algorithms to adopt? Naturally we tend to prefer the most efficient algorithm.

Sometimes one algorithm is faster, while an alternative algorithm needs less space. This is a classic space–time tradeoff, which can only be resolved with knowledge of the context in which the chosen algorithm will be used.

In this book we shall tend to pay more attention to time efficiency than to space efficiency. This is simply because time efficiency tends to be the critical factor in choosing between alternative algorithms.

Usually, the time taken by an algorithm depends on its input data. Figure 2.1 shows a hypothetical profile of two alternative sorting algorithms, showing how the time they take depends on $n$, the number of values to be sorted. Algorithm A is slightly faster for small $n$, but algorithm B wins more and more easily as $n$ increases.

How should we measure an algorithm's time efficiency? Perhaps the most obvious answer is to use real time, measured in seconds. Real time is certainly important in many practical situations. An interactive program that takes two minutes to respond to a user input will quickly fall into disuse. An aircraft control system that takes 30 seconds to respond to an abnormal altimeter reading will be eliminated by natural selection, along with the unfortunate crew and passengers.

Nevertheless, there are difficulties in using real time as a basis for comparing algorithms. An algorithm's real time requirement depends on the processor speed as well on the algorithm itself. Any algorithm can be made to run faster by using a faster processor,

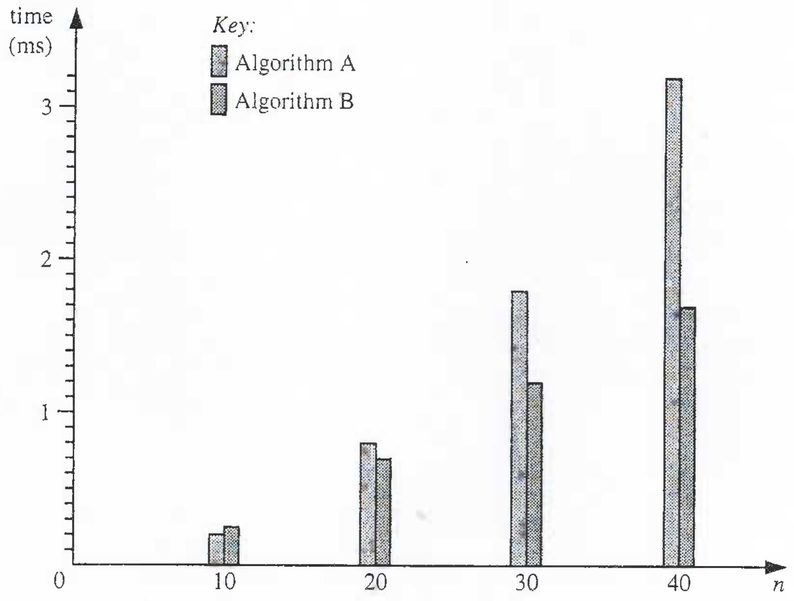

**Figure 2.1** Hypothetical profile of two sorting algorithms.

but this tells us nothing about the quality of the algorithm itself. And where the processor is a modern computer, the difficulty is compounded by the presence of software and hardware refinements – such as multiprogramming, pipelines, and caches – that increase the average speed of processing, but make it harder to predict the time taken by an individual algorithm.

We prefer to measure an algorithm's time efficiency in terms of the algorithm itself. One idea is simply to count the number of steps taken by the algorithm until it terminates. The trouble with this idea is that it depends on the granularity of the algorithm steps. Algorithms 2.2(a) and (b) solve the same problem in 3 and 7 steps, respectively. But they are just different versions of the same algorithm, one having course-grained (big) steps, while the other has fine-grained (small) steps.

(a)

```
To find the area of a triangle with sides $a$, $b$, $c$:
1. Let $s = (a + b + c)/2$.
2. Let $A = \sqrt{(s\,(s - a)\,(s - b)\,(s - c))}$.
3. Terminate with answer $A$.
```

(b)

```
To find the area of a triangle with sides $a$, $b$, $c$:
1. Let $s = (a + b + c)/2$.
2. Let $p = s$.
3. Multiply $p$ by $(s - a)$.
4. Multiply $p$ by $(s - b)$.
5. Multiply $p$ by $(s - c)$.
6. Let $A$ be the square root of $p$.
7. Terminate with answer $A$.
```

**Algorithm 2.2** Algorithms to find the area of a triangle.

The most satisfactory way to measure an algorithm's time efficiency is to count ***characteristic operations***. Which operations are characteristic depends on the problem to be solved. For an arithmetic algorithm it is natural to count arithmetic operations. For example, Algorithm 2.2(a) takes two additions, three subtractions, three multiplications, one division, and one square root; Algorithm 2.2(b) takes exactly the same number of arithmetic operations. In this book we shall see many examples of algorithms where comparisons or copies or other characteristic operations are the natural choice.

**EXAMPLE 2.2** ***Power algorithms***

Given a nonnegative integer $n$, the $n$th power of a number $b$, written $b^n$, is defined by:

$$b^n = b \times \cdots \times b \qquad (2.1)$$

(where $n$ copies of $b$ are multiplied together). For example:

$$\begin{aligned} b^3 &= b \times b \times b \\ b^2 &= b \times b \\ b^1 &= b \\ b^0 &= 1 \end{aligned}$$

Algorithm 2.3 (the 'simple' power algorithm) is based directly on definition (2.1). The variable $p$ successively takes the values 1, $b$, $b^2$, $b^3$, and so on – in other words, the successive powers of $b$. Program 2.4 is a Java implementation of Algorithm 2.3.

Let us now analyze Algorithm 2.3. The characteristic operations are obviously multiplications. The algorithm performs one multiplication for each iteration of the loop, and there will be $n$ iterations, therefore:

$$\text{No. of multiplications} = n \qquad (2.2)$$

Algorithm 2.3 is fine if we want to compute small powers like $b^2$ and $b^3$, but it is very time-consuming if we want to compute larger powers like $b^{20}$ and $b^{100}$.

Fortunately, there is a better algorithm. It is easy to see that $b^{20} = b^{10} \times b^{10}$. So once we know $b^{10}$, we can compute $b^{20}$ with only one more multiplication, rather than ten more multiplications. This shortcut is even more effective for still larger powers: once we know $b^{50}$, we can compute $b^{100}$ with only one more multiplication, rather than fifty.

Likewise, it is easy to see that $b^{21} = b^{10} \times b^{10} \times b$. So once we know $b^{10}$, we can compute $b^{21}$ with only two more multiplications, rather than eleven.

Algorithm 2.5 (the 'smart' power algorithm) takes advantage of these observations. The variable $q$ successively takes the values $b$, $b^2$, $b^4$, $b^8$, and so on. At the same time, the variable $m$ successively takes the values $n$, $n/2$, $n/4$, and so on (neglecting any remainders) down to 1. Whenever $m$ has an odd value, $p$ is multiplied by the current value of $q$. Program 2.6 is a Java implementation of Algorithm 2.5.

This algorithm is not easy to understand, but that is not the issue here. Instead, let us focus on analyzing its efficiency.

First of all, note that steps 2.1 and 2.2 each performs a multiplication, but the multiplication in step 2.1 is conditional. Between them, these steps perform *at most* two multiplications.

Next, note that these steps are contained within a loop, which is iterated as often as we can halve the value of $n$ (neglecting any remainder) until we reach zero. It can be shown (see Appendix A.2) that the number of iterations is floor($\log_2 n$) + 1, where floor($r$) is the function that converts a real number $r$ to an integer by discarding its fractional part.

Putting these points together:

$$\begin{aligned}\text{Maximum no. of multiplications} &= 2\,(\text{floor}(\log_2 n) + 1) \\ &= 2\,\text{floor}(\log_2 n) + 2 \qquad (2.3)\end{aligned}$$

The *exact* number of multiplications depends on the value of $n$ in a rather complicated way. For $n = 15$ the actual number of multiplications corresponds to (2.3), since halving 15 repeatedly gives a series of odd numbers; while for $n = 16$ the actual number of multiplications is smaller, since halving 16 repeatedly gives a series of even numbers. Equation (2.3) gives us the *maximum* number of multiplications for any given $n$, which is a pessimistic estimate.

Figure 2.7 plots (2.2) and (2.3) for comparison. The message should be clear. For small values of $n$, there is little to choose between the two algorithms. For larger values of $n$, the smart power algorithm is clearly better; indeed, its advantage grows as $n$ grows.

## 2.3 Complexity of algorithms

If we want to understand the efficiency of an algorithm, we first choose characteristic operations, and then analyze the algorithm to determine the number of characteristic

```
To compute $b^n$ (where $n$ is a nonnegative integer):

1.  Set $p$ to 1.
2.  For $i = 1, \ldots, n$, repeat:
    2.1.  Multiply $p$ by $b$.
3.  Terminate with answer $p$.
```

**Algorithm 2.3** Simple power algorithm.

```
static int power (int b, int n) {
// Return the value of b raised to the n'th power (where n is a nonnegative
// integer).
  int p = 1;
  for (int i = 1; i <= n; i++)
    p *= b;
  return p;
}
```

**Program 2.4** Java implementation of the simple power algorithm.