Contents

List of symbols vi
Preface ix

Chapter 1 Introduction to algorithms 1
  1.1 What is an algorithm? 1
  1.2 Control structures 3
  1.3 Further examples of algorithms 7
      Exercises 11

Chapter 2 Bases and number representation 15
  2.1 Real numbers and the decimal number system 15
  2.2 The binary number system 16
  2.3 Conversion from decimal to binary 18
  2.4 The octal and hexadecimal systems 22
  2.5 Arithmetic in non-decimal bases 24
      Exercises 26

Chapter 3 Computer representation and arithmetic 29
  3.1 Representing numbers in a computer 29
  3.2 Representing integers 29
  3.3 Arithmetic with integers 32
  3.4 Representing real numbers 35
  3.5 Arithmetic with real numbers 40
  3.6 Binary coded decimal representation 41
      Exercises 43

Chapter 4 Logic 45
  4.1 Logic and computing 45
  4.2 Propositions 45
  4.3 Connectives and truth tables 47
  4.4 Compound propositions 50
  4.5 Logical equivalence 53
  4.6 Laws of logic 55
  4.7 Predicate logic 60
  4.8 Proof techniques 65
      Exercises 68
## Chapter 5  Sets and relations

5.1 Sets  
5.2 Subsets, set operations and Venn diagrams  
5.3 Cardinality and Cartesian products  
5.4 Computer representation of sets  
5.5 Relations  

Exercises

## Chapter 6  Functions

6.1 Functions and computing  
6.2 Composite functions and the inverse of a function  
6.3 Functions in programming languages  

Exercises

## Chapter 7  Induction and recursion

7.1 Recursion and sequences  
7.2 Proof by induction  
7.3 Induction and recursion  
7.4 Solving linear recurrences  
7.5 Recursively defined functions and recursive algorithms  
7.6 Recursively defined functions in programming languages  

Exercises

## Chapter 8  Boolean algebra and digital circuits

8.1 Boolean algebra  
8.2 Simplifying Boolean expressions  
8.3 Digital circuits  
8.4 Disjunctive normal form and Karnaugh maps  

Exercises

## Chapter 9  Combinatorics

9.1 Combinatorics and computing  
9.2 The Principle of inclusion and exclusion  
9.3 The Multiplication principle  
9.4 Permutations  
9.5 Combinations  

Exercises

## Chapter 10  Introduction to graph theory

10.1 What is a graph?  
10.2 Basic concepts in graph theory
1 Introduction to algorithms

1.1 What is an algorithm?

A central theme in computing is the design of a process for carrying out a task. The task might be sorting names into alphabetical order, finding the cheapest way to link a set of computing sites into a network, converting a number to its representation in the binary system, encrypting a message for secure transmission, designing a digital circuit for a microchip, or determining the shortest path to be followed by a robotic arm. There are many occasions throughout this book when we find ourselves investigating problems of this nature, and asking: How can this task be performed by a person or a computer? In each case, the answer takes the form of a precise sequence of steps known as an algorithm.

An everyday example of an algorithm is provided by the steps you carry out when you withdraw cash from an automatic teller machine. We could write them out like this:

1. Insert your card into the slot.
2. Key in your personal identification number (PIN).
3. Select ‘Withdraw cash’ from the menu.
4. Enter the amount you want to withdraw.
5. Take your card, cash and transaction slip.

Steps 1–5 form a sequence of instructions to be carried out one after another. Each instruction is clear and unambiguous. (To some extent, this is a matter of opinion; for example, Step 1 does not specify which way round to insert the card, and Steps 2–4 do not tell us what to do if the machine rejects any of the input. Nevertheless, most people would probably agree that the instructions are clear enough for a person of normal intelligence to follow.) Finally, and importantly, the process is guaranteed to stop after a finite number of steps (five in this case). These are the essential features of an algorithm.
**Definition**

An algorithm is a finite sequence of steps for performing a task, such that:

- each step is a clear and unambiguous instruction that can be executed in a finite time;
- the sequence in which the steps are to be executed is clearly defined;
- the process is guaranteed to stop after a finite number of steps have been executed.

Notice that the algorithm in our example has some other properties that are not included in the definition, but that we would expect any useful algorithm to have. First, there is input (in this case the information keyed in), and output (the cash and the transaction slip). Second, the algorithm has been designed with a purpose in mind (to withdraw cash).

The definition of an algorithm may remind you of a computer program. A program contains the implementation of an algorithm in a particular programming language. We use the term ‘programming language’ here to include not only general purpose programming languages such as Pascal, C and Java, but more specialised languages such as the macro command languages used in spreadsheets and database management software. Designing an algorithm is one of the steps in writing a program. For the rest of this book, we will be primarily concerned with algorithms that are intended for implementation as programs on a computer.

In order to separate the process of designing an algorithm from the other aspects of programming in a particular language, the algorithms in this book will be written in a form of structured English called pseudocode. This will allow us to concentrate on the structure of the algorithm itself, without getting sidetracked by the details of a particular programming language.

Here is a simple example of an algorithm of the type that could be readily incorporated into a computer program.

**Example 1.1.1**

Write an algorithm to calculate the area of a circle, given the radius.

**Solution**

The area of a circle with radius \( r \) is \( \pi r^2 \). The algorithm is as follows:

1. Input \( r \) \{\( r \) is the radius of the circle.\}
2. \( area \leftarrow \pi r^2 \)
3. Output \( area \)

This example illustrates several basic points about the way algorithms are written in pseudocode. The steps are numbered consecutively for easy reference. Explanatory comments that are not part of the algorithm itself are written between braces \{\}. The symbol \( \leftarrow \) denotes assignment; thus in Step 2, the formula \( \pi r^2 \) is evaluated, and the result is assigned to the
variable area. (In computing terms, the result is stored in the memory at
an address specified by the identifier area.)

Notice also that mathematical formulae such as \( \pi r^2 \) are written in the
usual mathematical notation. When we write pseudocode we are
interested only in the structure of the algorithm; we don’t want to be
concerned with the details of how a formula would be written in a
programming language. (In C, for example, Step 2 would appear as
area = pi * r * r; in the program. Another C statement would be
needed to assign the appropriate numerical value to pi. These kinds of
details vary from one language to another.)

### 1.2 Control structures

In Example 1.1.1, the steps are simply executed in order, one after the
other. However, most algorithms contain one or more control structures –
instructions that specify how many times other instructions are to be
executed, or under what conditions they are to be executed. The next
element illustrates a situation in which control structures are needed.

#### Example 1.2.1

Find the smallest number in a list of numbers.

#### Solution

The smallest number can be found by looking at each number in turn,
keeping track at each step of the smallest number so far.

1. Input the number of values \( n \)
2. Input the list of numbers \( x_1, x_2, \ldots, x_n \)
3. \( min \leftarrow x_1 \)
4. For \( i = 2 \) to \( n \) do
   4.1. If \( x_i < min \) then
      4.1.1. \( min \leftarrow x_i \)
5. Output \( min \)

Two control structures appear in this example; we examine each of them
in turn.

The **For-do** in Step 4 causes the following step (or steps) to be executed
a specified number of times. In this example, the index variable, \( i \), ranges
through the values 2, 3, 4, ..., \( n \), and Step 4.1 is executed with \( i \) set equal to
each of these values in turn. This type of structure is often referred to as a
**loop**. We will adopt the convention that the value of the index variable is
not defined after the loop has run to completion.

The other control structure in Example 1.2.1 is the **If-then** in Step 4.1.
Its operation is simply described – Step 4.1.1 is executed if \( x_i < min \),
otherwise it is ignored.

Notice how the logical structure of the algorithm is indicated by the use
of indenting, and given further emphasis by the way the steps are
numbered – Step 4.1.1 is part of Step 4.1, which in turn is part of Step 4.
A list of some commonly used control structures is given in Table 1.1. These structures or their equivalents are available in many programming languages. (In C, C++ and Java, there is a do-while structure that plays the role of the Repeat-until.)

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Example of use</th>
</tr>
</thead>
</table>
| If-then           | 1. If \( x \) < 0 then  
|                   | \( 1.1. \; x \leftarrow -x \)  |
| If-then-else      | 1. If \( x \geq 0 \) then  
|                   | \( 1.1. \; y \leftarrow x \)   |
|                   | \( \text{else} \)  
|                   | \( 1.2. \; \text{Output} \; \sqrt{x} \; \text{does not exist}. \)  |
| For-do            | 1. \( \text{sum} \leftarrow 0 \)  
|                   | 2. \( \text{For} \; i = 1 \text{ to } 10 \; \text{do} \)  
|                   | \( 2.1. \; \text{sum} \leftarrow \text{sum} + i^2 \)  |
| While-do          | 1. \( \text{While} \; \text{answer} \neq \text{‘y’ and answer} \neq \text{‘n’ do} \)  
|                   | \( 1.1. \; \text{Output} \; \text{‘Please answer y or n.’} \)  
|                   | \( 1.2. \; \text{Input} \; \text{answer} \)  |
| Repeat-until      | 1. \( i \leftarrow 0 \)  
|                   | 2. \( \text{Repeat} \)  
|                   | \( 2.1. \; i \leftarrow i + 1 \)  
|                   | \( 2.2. \; \text{Input} \; x_i \)  
|                   | \( \text{until} \; x_i = 0 \)  |

If we have designed an algorithm to perform a particular task, we would naturally like to find out whether it does the task correctly. One useful technique, known as tracing an algorithm, is carried out by choosing a particular set of inputs and recording the values of all the variables at each step of the algorithm.

**Example 1.2.2**  
Trace the algorithm in Example 1.2.1 with inputs \( n = 3, x_1 = 5, x_2 = 4, x_3 = 8 \).

**Solution**  
See Table 1.2.

1 In some programming languages, notably C, C++ and Java, a double equals-sign, `==`, is used to denote equality. We prefer to keep our pseudocode as close as possible to standard mathematical notation.
Since 4 is the smallest number in the list 5, 4, 8, the trace table confirms that the algorithm gives the correct output for this set of inputs.

Of course, we cannot claim on the strength of the trace in Example 1.2.2 that the algorithm is correct for every possible set of inputs. A trace can reveal that an algorithm is wrong, but it can never be used to prove that an algorithm is correct.

The next example illustrates a situation where a While-do or a Repeat-until is useful.

Example 1.2.3

Design an algorithm to check whether a string $c_1c_2...c_n$ of $n$ characters consists entirely of digits or whether non-digit characters are present, and output an appropriate message.

Solution

We could use a For-do loop to check every character in the string, but it would be more efficient to stop checking as soon as a non-digit character is encountered. One way of doing this is to use a Repeat-until, with a counter $i$ to keep track of the position in the string:

1. $i \leftarrow 1$; nondigit_detected $\leftarrow$ false
2. Repeat
   2.1. If $c_i$ is not a digit then
      2.1.1. nondigit_detected $\leftarrow$ true
   2.2. $i \leftarrow i + 1$
   until nondigit_detected $= \text{true}$

We will always write identifiers in algorithms as strings of characters without spaces, using the underscore character where necessary if an

---

2 Step 1 is really two steps, written on one line for convenience and separated by a semicolon. This is often done where several variables are to be initialised (given initial values), as is the case here.
identifier is made of two or more English words (as we have done with nondigit_detected above).

The variable nondigit_detected is an example of a logical (or Boolean) variable; it may take only the values 'true' and 'false'. Logical variables are often useful in control structures such as the Repeat-until in this example. (It would be permissible to omit ‘= true’ from the last line, and just write ‘until nondigit_detected’.)

The counter \( i \) in this algorithm works in much the same way as the index \( i \) of the For-do in Example 1.2.1; it is initialised to 1 before the loop is entered, and incremented by 1 at the end of each pass through the loop. An important difference is that the initialisation and incrementation are built into the structure of the For-do, and so it is not necessary to write the step \( i \leftarrow i + 1 \) explicitly there; in fact, it would be an error to do so.

The process we have just described will stop with nondigit_detected = true as soon as a non-digit is detected, but it fails if the string contains only digits. We can fix this by testing at the end of each pass through the loop whether the end of the string has been reached. After the last character \( c_n \) has been checked, \( i \) is incremented in Step 2.2 to \( n + 1 \), so this is what we need to test for:

1. \( i \leftarrow 1; \) nondigit_detected \( \leftarrow \) false
2. Repeat
   2.1. If \( c_i \) is not a digit then
       2.1.1. nondigit_detected \( \leftarrow \) true
   2.2. \( i \leftarrow i + 1 \)
   until nondigit_detected = true or \( i \leftarrow n + 1 \)

This looks better; if there are no non-digit characters, the loop is executed \( n \) times and finishes with nondigit_detected = false. It remains only to add suitable input and output steps to complete the algorithm:

1. Input \( n \)
2. Input \( c_1c_2 \ldots c_n \)
3. \( i \leftarrow 1; \) nondigit_detected \( \leftarrow \) false
4. Repeat
   4.1. If \( c_i \) is not a digit then
       4.1.1. nondigit_detected \( \leftarrow \) true
   4.2. \( i \leftarrow i + 1 \)
   until nondigit_detected = true or \( i \leftarrow n + 1 \)
5. If nondigit_detected = true then
   5.1. Output ‘The string contains non-digit characters.’
   else
   5.2. Output ‘The string consists entirely of digits.’

What happens if a null string (\( n = 0 \)) is input? In this case we would be in trouble, because the loop would be entered with \( i = 1 \), and Step 4.1 could not be executed. This illustrates a limitation of the Repeat-until construct – a loop of this type always executes at least once.

One way to avoid the problem is to use a While-do; because the test is performed at the beginning of the loop rather than at the end, the steps
within a **While-do** loop need not to be executed at all. Here is the algorithm again, this time rewritten using a **While-do**:

1. Input \( n \)
2. Input \( c_1 c_2 \ldots c_n \)
3. \( i \leftarrow 0; \text{nondigit\_detected} \leftarrow \text{false} \)
4. **While** \( \text{nondigit\_detected} = \text{false} \) and \( i < n \) **do**
   4.1. \( i \leftarrow i + 1 \)
   4.2. **If** \( c_i \) is not a digit **then**
       4.2.1. \( \text{nondigit\_detected} \leftarrow \text{true} \)
5. **If** \( \text{nondigit\_detected} = \text{true} \) **then**
   5.1. Output ‘The string contains non-digit characters.’
   **else**
   5.2. Output ‘The string consists entirely of digits.’

This algorithm works even if \( n = 0 \).

### 1.3 Further examples of algorithms

In this section, we present four more examples showing how algorithms are developed, beginning with the specification of a problem, and following through the steps to the final algorithm.

#### Example 1.3.1

Design an algorithm to evaluate \( x^n \), where \( x \) is any real number and \( n \) is a positive integer. (Assume that the operation of multiplication is available, but raising to a power is not; this is the case, for example, in the programming language Pascal.)

**Solution**

If \( n \) is a positive integer, \( x^n \) can be evaluated using the formula:

\[
x^n = x \times x \times \ldots \times x
\]

\( n \) times

In order to carry out this process, we will need a variable \( \text{answer} \), which is initially assigned the value of \( x \), and which is then multiplied by \( x \) the required number of times. The number of multiplications is fixed at \( n - 1 \), so a **For-do** loop is the most appropriate control structure to use here.

1. Input \( x, n \)
2. \( \text{answer} \leftarrow x \)
3. **For** \( i = 1 \) **to** \( n - 1 \) **do**
   3.1. \( \text{answer} \leftarrow \text{answer} \times x \)
4. Output \( \text{answer} \)

Is this sequence of steps an algorithm? The steps are clear and unambiguous, they are executed in a clearly defined sequence, and Step 3.1 in the **For-do** loop is executed a finite number of times, so the process
is guaranteed to terminate eventually. Therefore all the requirements of an algorithm are satisfied.

Is the algorithm correct? Table 1.3 is a trace table with inputs $x = 2$ and $n = 3$.

<table>
<thead>
<tr>
<th>Step</th>
<th>$i$</th>
<th>$answer$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>–</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>3.1</td>
<td>1</td>
<td>4</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>–</td>
</tr>
<tr>
<td>3.1</td>
<td>2</td>
<td>8</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The output is correct in this case, since $2^3 = 8$.

It is also a good idea to check the algorithm using input values that lie at the extremes of the allowed range of inputs, because in practice this is often where an algorithm will fail. In this example, $n$ must be greater than or equal to 1, so we should check that the algorithm gives the correct output when $n = 1$. If $n = 1$ then Step 3 will read For $i = 1$ to 0 do ..., which means that Step 3.1 is not executed at all. (In general, the steps within a For-do are not executed if the final value of the loop index is less than the initial value.) The output is therefore simply the original value of $answer$, namely $x$. This is exactly what it should be, because $x^1 = x$.

With a simple algorithm such as this, the testing we have done is enough for us to be reasonably confident that it is correct (although of course we have not proved that it is). More extensive testing is needed if the algorithm is more complex.

Before we leave this example, let us consider a slightly different version of the problem. Suppose we were asked to design an algorithm to evaluate $x^n$, where $x$ is any real number and $n$ is a non-negative integer. Could we use the same algorithm? In order to answer this question, we need to ascertain whether the algorithm works if $n = 0$ is input. If $n = 0$, Step 3.1 is not executed, and the value of $x$ is output. This is not the correct output, because $x^0 = 1$, so we conclude that the algorithm will need to be altered. You are asked to do this in Exercise 1.7.

**Example 1.3.2**

Design an algorithm to swap the values of two variables. (In practice, such an algorithm might form part of a larger algorithm for sorting data into numerical or alphabetical order.)
Solution

Let the two variables be $x$ and $y$. We could try something like this:

1. $x \leftarrow y$
2. $y \leftarrow x$

This looks quite plausible at first, but a trace quickly reveals that something is wrong!

Table 1.4 shows a trace with $x = 2$ and $y = 3$.

<table>
<thead>
<tr>
<th>Step</th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The problem occurs because the original value of $x$ is lost when Step 1 is executed. One way to prevent this from happening is to store the original value in another variable. This can be done in the following way:

1. $\text{temp} \leftarrow x$
2. $x \leftarrow y$
3. $y \leftarrow \text{temp}$

You should check that the trace now gives the correct result.

Example 1.3.3

The applicants for a certain position are given an aptitude test consisting of 20 multiple-choice questions. Design an algorithm to output a list of the applicants (identified by number), their scores, and a message stating whether they are to be short-listed for the position (those who score 16 or more) or reconsidered at a later date (those who score in the range from 12 to 15). The input consists of the answers provided by each applicant.

Solution

We will use `number_of_applicants` to denote the number of applicants. The same procedure will need to be applied to each applicant, so a `For-do` is the appropriate structure to use, with the index $i$ ranging from 1 to `number_of_applicants`.

We denote the 20 answers submitted by Applicant $i$ by $a_{i,1}, a_{i,2}, \ldots, a_{i,20}$. In order to calculate the applicant’s score, the answer $a_{i,q}$ provided by Applicant $i$ to Question $q$ will need to be compared with the correct answer for that question, which we denote by $c_q$. Another `For-do` loop, 'nested' within the first and with $q$ ranging from 1 to 20, will be needed to accomplish this.

In the final steps of the outer `For-do`, the output for Applicant $i$ will need to be generated according to the specifications in the problem.
1. Input `number_of_applicants`
2. For `i` from 1 to `number_of_applicants` do
   2.1. `score` $\leftarrow 0$
   2.2. For `q` from 1 to 20 do
      2.2.1. Input `ai,q` \{ `ai,q` is Applicant `i`’s answer to Question `q` \}
      2.2.2. If `ai,q = c_q` then  \{ `c_q` is the correct answer to Question `q` \}
          2.2.2.1. `score` $\leftarrow` `score` + 1
   2.3. Output `i`, `score`
2.4. If `score $\geq 16$` then
   2.4.1. Output 'Short-listed'
   else if `score $\geq 12$` then
   2.4.2. Output 'Reconsider later'

The layout of Step 2.4 needs some explanation. Strictly speaking, according to the rules we have laid down, the if-then after the else should be shown as a separate numbered statement, like this:

2.4. If `score $\geq 16$` then
   2.4.1. Output 'Short-listed'
   else
   2.4.2. If `score $\geq 12$` then
          2.4.2.1. Output 'Reconsider later'

In practice, however, we think of the conditions 'score $\geq 16$' and 'score $\geq 12$' as specifying two cases that require different actions to be taken. The layout we have used reflects this way of looking at it.

Example 1.3.4
Write an algorithm to locate the first occurrence of a given sequence of characters in a text. A suitable message should be output if the sequence of characters does not occur in the text.

Solution
The inputs to the algorithm are the text to be searched and the sequence of characters to be searched for. A simple way of searching for a sequence of characters is to step through the text, one character at a time, until either the sequence is found starting at that character or the end of the text is reached.

1. Input `c_1c_2...c_n` \{The text as a string of characters.\}
2. Input `a_1a_2...a_m` \{The sequence of characters to be searched for.\}
3. `location` $\leftarrow$ 1
4. `found` $\leftarrow$ false
5. While `location $\leq n - m + 1$` and `found` = false do
   5.1. If `c_{location}c_{location+1}...c_{location+m-1} = a_1a_2...a_m` then
       5.1.1. `found` $\leftarrow$ true
   else
       5.1.2. `location` $\leftarrow$ `location` + 1
6. If `found` = true then
   6.1. Output `location`
else

6.2. Output ‘The sequence of characters does not occur in the text.’

Note that in Step 5.1, we have simply tested two strings of characters for equality. We could have spelt this step out in more detail by comparing the two strings character by character.

String searching algorithms are implemented as a standard feature of text editing and word processing software. We remark that an algorithm known as the Boyer–Moore algorithm (or a variant of it) is commonly used in practice because it is more efficient than the simple algorithm described here.

The purpose of this chapter has been to present a brief introduction to algorithms and to define the pseudocode we will be using. Algorithms play a central role in computing and discrete mathematics, and they will reappear frequently in subsequent chapters.

Exercises

1. Modify the algorithm in Example 1.2.1 so that the output also includes the position in the list where the smallest number occurs.

2. Write an algorithm to input a period of time in hours, minutes and seconds, and output the time in seconds.

3. Write an algorithm to input a number \( n \), then calculate \( 1^2 + 2^2 + 3^2 + \ldots + n^2 \), the sum of the first \( n \) perfect squares, and output the result.

4. Write an algorithm to input the price of a purchase and the amount tendered, and calculate the change due. An appropriate message should be output if the amount tendered is less than the purchase price.

5. Write an algorithm to calculate the tax payable on a given taxable income, according to the following rules:

<table>
<thead>
<tr>
<th>Taxable income</th>
<th>Tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1–$5400</td>
<td>0</td>
</tr>
<tr>
<td>$5401–$20700</td>
<td>0 plus 20 cents for each $1 over $5400</td>
</tr>
<tr>
<td>$20701–$38000</td>
<td>$3060 plus 34 cents for each $1 over $20700</td>
</tr>
<tr>
<td>$38001–$50000</td>
<td>$8942 plus 43 cents for each $1 over $38000</td>
</tr>
<tr>
<td>$50001 and over</td>
<td>$14102 plus 47 cents for each $1 over $50000</td>
</tr>
</tbody>
</table>
Write an algorithm to input a list of numbers and test whether the numbers are in increasing order of magnitude, giving an appropriate message as output. The algorithm should be designed so that testing stops as soon as the answer is known.

Modify the algorithm in Example 1.3.1 so that it gives the correct output when \( n = 0 \) is input.

The following algorithm calculates a number known as the ‘digital root’ of a positive integer.

1. Input a positive integer \( n \)
2. \( d \leftarrow \) number of digits in \( n \)
3. While \( d > 1 \) do
   3.1. \( n \leftarrow \) sum of the digits of \( n \)
   3.2. \( d \leftarrow \) number of digits in \( n \)
4. Output \( n \)

(a) Trace the algorithm when 8678 is input.
(b) List all the possible values that the output of the algorithm could take.

Consider the following sequence of steps:

1. Input a non-negative integer \( n \)
2. \( i \leftarrow 0 \)
3. While \( n \) is even do
   3.1. \( n \leftarrow \) \( n \) / 2
   3.2. \( i \leftarrow i + 1 \)
4. Output \( i \)

(a) What is the output when 12 is input?
(b) What is the output when any odd number is input?
(c) What happens when 0 is input?
(d) Is this sequence of steps an algorithm?

Consider the following sequence of steps:

1. Input a positive integer \( n \)
2. \( answer \leftarrow n \)
3. While \( n > 1 \) do
   3.1. \( n \leftarrow n - 1 \)
   3.2. \( answer \leftarrow answer \times n \)
4. Output \( answer \)

(a) Construct a trace table to show what happens when 4 is input.
(b) Is this sequence of steps an algorithm? Give a reason for your answer.

Write an algorithm to input a string of characters and test whether the parentheses (round brackets) in the string are paired correctly. (Use a variable \( excess\_left \), which records the excess of the number of left parentheses over the number of right...
parentheses as the algorithm looks at each character in turn. If `excess_left` is never negative, and the end of the string is reached with `excess_left = 0`, then the parentheses are paired correctly.)

12 Write an algorithm that takes a passage of text (as a string of characters) as input, and outputs the number of words in the passage. Assume that each word is separated from the next word by one or more spaces. In particular, the algorithm must work correctly if the passage begins or ends with one or more spaces.

13 Consider the following algorithm:
   1. Input a positive integer $n$
   2. For $i = 1$ to $n$ do
      2.1. $a_i \leftarrow 0$
   3. For $i = 1$ to $n$ do
      3.1. For $j = 1$ to $n$ do
         3.1.1. If $j$ is divisible by $i$ then
            3.1.1.1. $a_j \leftarrow 1 - a_j$
            \{ $a_j$ is always either 0 or 1 \}
   4. For $i = 1$ to $n$ do
      4.1. Output $a_i$
   (a) List the values taken by $a_1, a_2, \ldots, a_n$ at the end of each pass through the outer For-do loop (Step 3) when the algorithm is run with $n = 10$.
   (b) Given any value of $n$, can you predict the final value of each of the $a_i$s without tracing through the algorithm? Justify your answer.

14 Consider the following algorithm, which rearranges the order of a list of numbers:
   1. Input $x_1, x_2, \ldots, x_n$ \{ numbers \}
   2. For $i = 1$ to $n-1$ do
      2.1. If $x_1 > x_i+1$ then
         2.1.1. Swap $x_i$ and $x_{i+1}$
   3. Output $x_1, x_2, \ldots, x_n$
   (a) For the input 6, 3, 2, write down the list of numbers after each swap has been performed.
   (b) Repeat part (a) with the input 5, 3, 8, 4.
   (c) Explain why, at the final step, the largest number in the list is always at the end of the list.

15 Consider the following algorithm:
   1. Input $x_1, x_2, \ldots, x_n$ \{ numbers \}
   2. For $i = 1$ to $n-1$ do
      2.1. $j \leftarrow n - 1$ \{ $j$ steps through $n-1, n-2, \ldots, 1$ \}
   2.2. For $k = 1$ to $j$ do
      2.2.1. If $x_k > x_{k+1}$ then
2.2.1.1. Swap $x_k$ and $x_{k+1}$

3. Output $x_1, x_2, \ldots, x_n$

(a) For the input 6, 3, 2, write down the list of numbers after each swap has been performed.

(b) Repeat part (a) with the input 5, 3, 8, 4.

(c) What is the purpose of the algorithm?

16 Consider the process defined by the following sequence of steps:

1. Input a positive integer $n$

2. While $n \neq 1$ do
   2.1. If $n$ is even then
      2.1.1. $n \leftarrow n / 2$
   else
      2.1.2. $n \leftarrow 3n + 1$

3. Output ‘Finished!’

(a) List the successive values taken by $n$ if an initial value of 7 is input.

(b) Explore the process for various other inputs.

(The problem of determining whether the process terminates for all possible inputs, and hence whether or not this sequence of steps is an algorithm, is unsolved. This problem is known as the Collatz conjecture (and also by various other names). A search on the Web should supply further information about the problem, including the current status of attempts to solve it.)
Index

1’s complement 31
2’s complement 30–1, 34
10’s complement of a decimal integer 43

absolute value 30
absorption laws
  in Boolean algebra 148
  in logic 56
  in sets 81
addition
  binary 24–5
  in Boolean algebra 145–6
  integer, on a computer 33–4, 236
  real, on a computer 40
adjacency matrix 186–7
adjacent 181
affine cipher 243–4
algorithm 2
  greedy 212
  iterative 117
  recursive 130–5
  for a sequence 115
  tracing 4
  tractable 271
ancestor 216
and (connective) 47, 48
AND gate 153
annihilation laws
  in Boolean algebra 148
  in logic 56
  in sets 81
antisymmetric relation 88
argument, validity of 59–60
arithmetic
  binary 24–6
  IEEE standard 36, 37, 40
  integer, on a computer 32–5, 235–6
  real number, on a computer 40–1
arrow diagram 100
assignment 2–3
associative axioms in Boolean algebra 146
associative laws
  in logic 56
  in sets 81
atomic proposition 47
axiom, Boolean 145–6
balanced ternary number system 28
base (in exponential notation) 35
base, number 15–17
bases, related 23
BCD representation 41–2
‘big O’ 262
binary arithmetic 24–6
binary coded decimal representation 41–2
binary exponential form, normalised 36
binary number system 17–18
arithmetic 24–6
conversion 18–21, 23–4
binary operation 145
binary relation 87
binary rooted tree 217
binary search 267
binomial coefficient 173, 175–6
bit 17
bitwise operation 85
Boolean algebra 145–6
  laws 148
Boolean expression 148
Boolean function 152
Boolean variable 6
  bound variable 61–2
Caesar cipher 243
cardinality 82
Cartesian product 83
characteristic 37
characteristic equation 127
child 216
circuit
  digital 151
  Eulerian 194, 269–70
  in a graph 193
  Hamiltonian 199, 270
CMY system 84, 86
codomain 98
colours, computer representation 84, 86
combination 172–3
comment (in pseudocode) 2
commutative axioms in Boolean algebra 146
commutative laws
  in logic 56
  in sets 81
complement
  1’s 31
Index

2's 30–1, 34
10's, of a decimal integer 43
of a graph 186
of a set 78
complement laws in Boolean algebra 148
complementation in Boolean algebra 145–6
complete graph 186
complexity of an algorithm 254
component of a graph 194
composite function 103–4
composite number 225
compound proposition 47, 50–1
compression, file 108
computer representation
  of colours 84, 86
  of numbers 29–31, 35–40
  of sets 85–6
conclusion 59
congruent 233
connected graph 181, 193
connective 47
  principal 51
contradiction 53
contrapositive 54–5
  proof using 67
control structures 3–4
converse 54–5
conversion
  binary to hexadecimal 24
  binary to octal 23
  decimal to binary 18–21
  decimal to hexadecimal 22–3
  decimal to octal 22
  hexadecimal to binary 24
  octal to binary 24
coprime 227
counterexample 67–8
cryptography 242
  public key 243, 246–8
cryptosystem, RSA 246–8
cybercycle 206
de Morgan's laws
  in Boolean algebra 148
  in logic 56
  in sets 81
decimal number system 15–16
decryption 242
degree 184
descendant 216
difference (of sets) 78–9
digital circuit 151
digital root 12, 250
Dijkstra's algorithm 214–15
direct proof 66
directed graph 87, 182
disjoint sets 77
disjunctive normal form 156–7
distance in a weighted graph 214
distributive axioms in Boolean algebra 146
distributive laws
  in logic 56
  in sets 81
divides 225
division
  binary 26
  real, on a computer 40
divisor 225
domain 98
double complement law
  in Boolean algebra 148
  in sets 81
double negation law in logic 56
dual (of a law of logic) 55
duality in Boolean algebra 147–8
degree 180
element 74, 75
encryption 108, 242–8
  public key 243, 246–8
enumerated form 74–5
equality (of sets) 77
equivalence class 90–2, 233–5
equivalence law in logic 56
equivalence, logical 54
error detection 111–12
Euclidean algorithm 228–9
Eulerian circuit 194, 269–70
Eulerian graph 194–5
Eulerian path 194
exclusive-or 48, 68–9
  bitwise 95
exponent 35
exponent bias 37
exponential notation 35
expression
  Boolean 148
  logical 51
expression tree 217
  logical 51
factorial 118, 130–1, 171, 270–1
factorisation, prime 225–6
Fibonacci sequence 117–18, 137, 140, 141
file compression 108
‘for all’ 61
for-do 3, 4
forest 207
fractional part 20
free variable 62
Index

full adder 166
function 98
Boolean 152
composite 103–4
identity 106
inverse 106–7
one-to-one 101, 107
onto 101, 107
in a programming language 109–10, 129–30, 135–6
recursively defined 130–2
Fundamental theorem of arithmetic 226, 248
general solution of a recurrence 127
graph 180
complete 186
connected 181, 193
directed 87, 182
Eulerian 194–5
Hamiltonian 199
null 181
semi-Eulerian 194–5, 199
simple 182
weighted 208–9
graphs, isomorphism of 182–3, 188–9
greatest common divisor 227–9
greedy algorithm 223
half adder 166
Hamiltonian circuit 199, 270
Hamiltonian graph 199
Hamiltonian path 199
Handshaking lemma 185
hexadecimal number system 17, 22–4
homogeneous recurrence 126
Horner’s method 27
Huffman code 223
idempotent laws
in Boolean algebra 148
in logic 56
in sets 81
identity axioms in Boolean algebra 146
identity function 106
identity laws
in logic 56
in sets 81
IEEE standard arithmetic 36, 37, 40
if-and-only-if (connective) 47, 50
if-then (connective) 47, 49–50
if-then (control structure) 4
if-then-else 4, 10
image 98
implication 54
implication law in logic 56
in-order traversal 218
incident 181
inclusion and exclusion, Principle of 168
inclusive-or 48
index variable 3
induction 119–23
and recursion 123–6
infix notation 218
initial condition 126
insertion sort 265–9
integer arithmetic on a computer 32–5, 235–6
integer part 20
integers 15, 75
computer representation 29–31
intersection 77–8
inverse axioms in Boolean algebra 146
inverse of a function 106–7
inverse laws
in logic 56
in sets 81
inverter 153
irrational numbers 15–16, 72
irreflexive relation 88
isolated vertex 184
isomorphic graphs 182–3, 188–9
isomorphic rooted trees 223
iterative algorithm 117
Karnaugh map 158–63
Königsberg bridge problem 191–2, 194–5
Kruskal’s algorithm 221–2
laws
of Boolean algebra 148
of logic 56, 144
of sets 81, 144
leaf 216
least common multiple 227, 232
length of path 214
linear congruential method 239–40
linear recurrence 126
local area network 208
logarithm (base 2) 268
logic
laws 56, 144
predicate 60–5
propositional 45–60, 143–4
logic gate 152–4
logic gate circuit 151
logical equivalence 54
logical expression 51
simplification 57–8
logical paradox 46
logical variable 6
loop
in an algorithm 3
in a graph 181–2
lower triangular matrix representation 188

matrix
  adjacency 186–7
  relation 88
  weight 212–13
  merging 275
  mergesort 275–6
  minimal spanning tree 209
  minimum distance problem 214
  minterm 156
  modulus 233
  Moore’s law 272–3
  multiple 225
  multiplication
    binary 25–6
    in Boolean algebra 145–6
    real, on a computer 40
  Multiplication principle 169–70
  multiplicative congruential method 239–40
  nand (connective) 70–1
  NAND gate 167
  natural numbers 15, 75
  nearest neighbour algorithm 212
  negating a proposition 63–4
  non-decimal arithmetic 24–6
  non-recursive definition of a sequence 114–15
  normalised binary exponential form 36
  normalised representation of a real number 35–6
  not (connective) 47–9, 63
  NOT gate 153
  n-tuple, ordered 82–3
  null graph 181
  null set 75–6, 77
  number base 15–17
  number system
    balanced ternary 28
    binary 17–18
    decimal 15–16
    hexadecimal 17, 22–4
    octal 17, 22–4
    positional 16
  numbers
    computer representation 29–31, 35–40
    irrational 15–16, 72
    natural 15, 75
    rational 15, 75
    real 16, 75
    representation 15
  octal number system 17, 22–4
  one-to-one function 101, 107
  onto function 101, 107
  operation
    binary 145
    unary 145
    or (connective) 47, 48–9
    OR gate 153
    ordered n-tuple 82–3
    ordered pair 83
    overflow 34, 39
  paradox, logical 46
  parallel edges 182
  parent 216
  partial order relation 92–3
  partition 90
  Pascal’s triangle 175–6
  path 193
  Eulerian 194
  Hamiltonian 199
  permutation 171–2
  place value (of a digit) 16
  Polish notation 219
  positional number system 16
  post-order traversal 219–20
  power set 82
  powers-of-ten notation 35
  pre-order traversal 219
  predicate 46, 61
  predicate form 75
  predicate logic 60–5
  prefix notation 219
  premise 59
  Prim’s algorithm 210
  prime factorisation 225–6
  prime number 225
  principal connective 51
  Principle of inclusion and exclusion 168
  proof 65
    by contradiction 67
    using contrapositive 67
    direct 66
    disproof by counterexample 67–8
    by induction 119–23
    proper subset 77
  proposition 45–6
    atomic 47
    compound 47, 50–1
    propositional logic 45–60, 143–4
    pseudo-random number 239
    pseudo-random number generator 110, 239–40
    pseudocode 2
    public key encryption 243, 246–8
  quantifier 61–2
  quotient 18–19
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>random number</td>
<td>238–9</td>
</tr>
<tr>
<td>range</td>
<td>99</td>
</tr>
<tr>
<td>rational numbers</td>
<td>15, 75</td>
</tr>
<tr>
<td>real numbers</td>
<td>16, 75</td>
</tr>
<tr>
<td>computer representation</td>
<td>35–40</td>
</tr>
<tr>
<td>real number arithmetic on a computer</td>
<td>40–1</td>
</tr>
<tr>
<td>recurrence, linear</td>
<td>126</td>
</tr>
<tr>
<td>recursion</td>
<td>93, 114</td>
</tr>
<tr>
<td>and induction</td>
<td>123–6</td>
</tr>
<tr>
<td>recursive algorithm</td>
<td>130–5</td>
</tr>
<tr>
<td>recursive definition</td>
<td>118–19</td>
</tr>
<tr>
<td>of a sequence</td>
<td>115–18</td>
</tr>
<tr>
<td>recursively defined function</td>
<td>130–1</td>
</tr>
<tr>
<td>relation matrix</td>
<td>88</td>
</tr>
<tr>
<td>relation</td>
<td>86–7</td>
</tr>
<tr>
<td>antisymmetric relation</td>
<td>88</td>
</tr>
<tr>
<td>equivalence</td>
<td>89–90, 233</td>
</tr>
<tr>
<td>irreflexive relation</td>
<td>88</td>
</tr>
<tr>
<td>partial order</td>
<td>92–3</td>
</tr>
<tr>
<td>reflexive relation</td>
<td>88</td>
</tr>
<tr>
<td>symmetric relation</td>
<td>88</td>
</tr>
<tr>
<td>total order</td>
<td>93</td>
</tr>
<tr>
<td>transitive relation</td>
<td>88</td>
</tr>
<tr>
<td>relatively prime</td>
<td>227</td>
</tr>
<tr>
<td>remainder</td>
<td>18–19</td>
</tr>
<tr>
<td>repeat-until</td>
<td>4</td>
</tr>
<tr>
<td>representation</td>
<td></td>
</tr>
<tr>
<td>binary coded decimal (BCD)</td>
<td>41–2</td>
</tr>
<tr>
<td>of integers, computer</td>
<td>29–31</td>
</tr>
<tr>
<td>of numbers</td>
<td>15</td>
</tr>
<tr>
<td>of real numbers, computer</td>
<td>35–40</td>
</tr>
<tr>
<td>of sets, computer</td>
<td>85–6</td>
</tr>
<tr>
<td>reverse Polish notation</td>
<td>220</td>
</tr>
<tr>
<td>RGB system</td>
<td>84, 86</td>
</tr>
<tr>
<td>rooted tree</td>
<td>215–16</td>
</tr>
<tr>
<td>binary</td>
<td>217</td>
</tr>
<tr>
<td>RSA cryptosystem</td>
<td>246–8</td>
</tr>
<tr>
<td>second-order recurrence</td>
<td>126</td>
</tr>
<tr>
<td>seed</td>
<td>239</td>
</tr>
<tr>
<td>self-reference</td>
<td>46</td>
</tr>
<tr>
<td>semi-Eulerian graph</td>
<td>194–5, 199</td>
</tr>
<tr>
<td>sequence</td>
<td>114</td>
</tr>
<tr>
<td>set</td>
<td>74</td>
</tr>
<tr>
<td>sets</td>
<td></td>
</tr>
<tr>
<td>computer representation</td>
<td>85–6</td>
</tr>
<tr>
<td>laws</td>
<td>81, 144</td>
</tr>
<tr>
<td>Sieve of Eratosthenes</td>
<td>248–9</td>
</tr>
<tr>
<td>sign bit</td>
<td>30</td>
</tr>
<tr>
<td>significand</td>
<td>35</td>
</tr>
<tr>
<td>simple graph</td>
<td>182</td>
</tr>
<tr>
<td>solution of recurrence</td>
<td>127</td>
</tr>
<tr>
<td>sorting</td>
<td>265–9</td>
</tr>
<tr>
<td>spanning tree</td>
<td></td>
</tr>
<tr>
<td>subset</td>
<td>76–7</td>
</tr>
<tr>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>25</td>
</tr>
<tr>
<td>integer, on a computer</td>
<td>34–5</td>
</tr>
<tr>
<td>real, on a computer</td>
<td>40</td>
</tr>
<tr>
<td>subtree</td>
<td>217</td>
</tr>
<tr>
<td>summation notation</td>
<td>120</td>
</tr>
<tr>
<td>symmetric relation</td>
<td>88</td>
</tr>
<tr>
<td>tautology</td>
<td>52, 58–9</td>
</tr>
<tr>
<td>term</td>
<td>114</td>
</tr>
<tr>
<td>‘there exists’</td>
<td>61</td>
</tr>
<tr>
<td>time complexity</td>
<td>254</td>
</tr>
<tr>
<td>total order relation</td>
<td>93</td>
</tr>
<tr>
<td>Tower of Hanoi</td>
<td>142</td>
</tr>
<tr>
<td>trace table</td>
<td>4–5</td>
</tr>
<tr>
<td>tracing an algorithm</td>
<td>4</td>
</tr>
<tr>
<td>tractable algorithm</td>
<td>271</td>
</tr>
<tr>
<td>transitive relation</td>
<td>88</td>
</tr>
<tr>
<td>Travelling sales representative problem</td>
<td>212</td>
</tr>
<tr>
<td>traversal</td>
<td></td>
</tr>
<tr>
<td>in-order</td>
<td>218</td>
</tr>
<tr>
<td>post-order</td>
<td>219–20</td>
</tr>
<tr>
<td>pre-order</td>
<td>219</td>
</tr>
<tr>
<td>of a tree</td>
<td>218</td>
</tr>
<tr>
<td>tree</td>
<td>180, 207</td>
</tr>
<tr>
<td>binary rooted</td>
<td>217</td>
</tr>
<tr>
<td>decision</td>
<td>216</td>
</tr>
<tr>
<td>expression</td>
<td>217</td>
</tr>
<tr>
<td>rooted</td>
<td>215–16</td>
</tr>
<tr>
<td>truth table</td>
<td>48–55, 151–2</td>
</tr>
<tr>
<td>truth value</td>
<td>45</td>
</tr>
<tr>
<td>unary operation</td>
<td>145</td>
</tr>
<tr>
<td>underflow</td>
<td>39</td>
</tr>
<tr>
<td>union</td>
<td>78</td>
</tr>
<tr>
<td>universal set</td>
<td>76</td>
</tr>
<tr>
<td>validity of an argument</td>
<td>56–60</td>
</tr>
<tr>
<td>variable</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>6</td>
</tr>
<tr>
<td>bound</td>
<td>61–2</td>
</tr>
<tr>
<td>free</td>
<td>62</td>
</tr>
<tr>
<td>logical</td>
<td>6</td>
</tr>
<tr>
<td>Venn diagram</td>
<td>76</td>
</tr>
<tr>
<td>vertex</td>
<td>180</td>
</tr>
<tr>
<td>isolated</td>
<td>184</td>
</tr>
<tr>
<td>weight of edge</td>
<td>209</td>
</tr>
<tr>
<td>weight matrix</td>
<td>206–7</td>
</tr>
<tr>
<td>weighted graph</td>
<td>212–13</td>
</tr>
<tr>
<td>while-do</td>
<td>4</td>
</tr>
<tr>
<td>xor</td>
<td>48, 68–9</td>
</tr>
<tr>
<td>bitwise</td>
<td>95</td>
</tr>
</tbody>
</table>