Chapter 1

INTRODUCTION

1.1 WHAT IS AN ALGORITHM?

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics. This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique, or method.

Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

Criteria 1 and 2 require that an algorithm produce one or more *outputs* and have zero or more *inputs* that are externally supplied. According to criterion 3, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as "add 6 or 7 to x" or "compute $5/0$" are not permitted because it is not clear which of the two possibilities should be done or what the result is.

The fourth criterion for algorithms we assume in this book is that they *terminate* after a finite number of operations. A related consideration is that the time for termination should be reasonably short. For example, an algorithm could be devised that decides whether any given position in the game of chess is a winning position. The algorithm works by examining all possible moves and countermoves that could be made from the starting position. The difficulty with this algorithm is that even using the most modern computers, it may take billions of years to make the decision. We must be very concerned with analyzing the efficiency of each of our algorithms.

Criterion 5 requires that each operation be *effective*; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithms that are definite and effective are also called *computational procedures*. One important example of computational procedures is the operating system of a digital computer. This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered. Though computational procedures include important examples such as this one, we restrict our study to computational procedures that always terminate.

To help us achieve the criterion of definiteness, algorithms are written in a programming language. Such languages are designed so that each legitimate sentence has a unique meaning. A *program* is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program. Most readers of this book have probably already programmed and run some algorithms on a computer. This is desirable because before you study a concept in general, it helps if you had some practical experience with it. Perhaps you had some difficulty getting started in formulating an initial solution to a problem, or perhaps you were unable to decide which of two algorithms was better. The goal of this book is to teach you how to make these decisions.

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. *How to devise algorithms* — Creating an algorithm is an art which may never be fully automated. A major goal of this book is to study vari-

1.2 ALGORITHM SPECIFICATION

1.2.1 Pseudocode Conventions

In computational theory, we distinguish between an algorithm and a program. The latter does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called *flowcharts* are another possibility, but they work well only if the algorithm is small and simple. In this text we present most of our algorithms using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces: { and }. A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.

3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, boolean, and so on. Compound data types can be formed with **record**s. Here is an example:

```
node = record
       {   datatype_1  data_1;
                 .
                 .
                 .
           datatype_n  data_n;
           node        *link;
       }
```

   In this example, *link* is a pointer to the record type *node*. Individual data items of a record can be accessed with $\rightarrow$ and period. For instance if p points to a record of type *node*, $p \rightarrow data\_1$ stands for the value of the first field in the record. On the other hand, if q is a record of type *node*, $q.data\_1$ will denote its first field.

4. Assignment of values to variables is done using the assignment statement

   $\langle variable \rangle := \langle expression \rangle$;

5. There are two boolean values **true** and **false**. In order to produce these values, the logical operators **and**, **or**, and **not** and the relational operators $<, \leq, =, \neq, \geq$, and $>$ are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the (i, j)th element of the array is denoted as $A[i, j]$. Array indices start at zero.

7. The following looping statements are employed: **for**, **while**, and **repeat-until**. The **while** loop takes the following form:

   while $\langle condition \rangle$ **do**
   {
   $\langle statement\ 1 \rangle$
   $\vdots$
   $\langle statement\ n \rangle$
   }

   As long as $\langle condition \rangle$ is **true**, the statements get executed. When $\langle condition \rangle$ becomes **false**, the loop is exited. The value of $\langle condition \rangle$ is evaluated at the top of the loop.

   The general form of a **for** loop is

   for *variable* := *value*1 **to** *value*2 **step** *step* **do**
   {
   $\langle statement\ 1 \rangle$
   $\vdots$
   $\langle statement\ n \rangle$
   }

   Here *value*1, *value*2, and *step* are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** *step*" is optional and taken as +1 if it does not occur. *step* could either be positive or negative. *variable* is tested for termination at the start of each iteration. The **for** loop can be implemented as a **while** loop as follows:

variable := *value*1;
fin := *value*2;
incr := *step*;
while ((*variable* − fin) ∗ *step* ≤ 0) **do**
{
⟨*statement* 1⟩
⋮
⟨*statement* n⟩
variable := *variable* + *incr*;
}

A **repeat-until** statement is constructed as follows:

repeat
⟨*statement* 1⟩
⋮
⟨*statement* n⟩
until ⟨*condition*⟩

The statements are executed as long as ⟨*condition*⟩ is **false**. The value of ⟨*condition*⟩ is computed after executing the statements.

The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.

8. A conditional statement has the following forms:

if ⟨*condition*⟩ **then** ⟨*statement*⟩
if ⟨*condition*⟩ **then** ⟨*statement* 1⟩ **else** ⟨*statement* 2⟩

Here ⟨*condition*⟩ is a boolean expression and ⟨*statement*⟩, ⟨*statement* 1⟩, and ⟨*statement* 2⟩ are arbitrary statements (simple or compound).

We also employ the following **case** statement:

case
{
:⟨*condition* 1⟩**:** ⟨*statement* 1⟩
⋮
:⟨*condition* n⟩**:** ⟨*statement* n⟩
:else: ⟨*statement* $n + 1$⟩
}

Here ⟨*statement* 1⟩, ⟨*statement* 2⟩, etc. could be either simple statements or compound statements. A **case** statement is interpreted as follows. If ⟨*condition* 1⟩ is **true**, ⟨*statement* 1⟩ gets executed and the **case** statement is exited. If ⟨*statement* 1⟩ is **false**, ⟨*condition* 2⟩ is evaluated. If ⟨*condition* 2⟩ is **true**, ⟨*statement* 2⟩ gets executed and the **case** statement exited, and so on. If none of the conditions ⟨*condition* 1⟩, ..., ⟨*condition* n⟩ are true, ⟨*statement* $n+1$⟩ is executed and the **case** statement is exited. The **else** clause is optional.

9. Input and output are done using the instructions **read** and **write**. No format is used to specify the size of input or output quantities.

10. There is only one type of procedure: **Algorithm**. An algorithm consists of a heading and a body. The heading takes the form

    Algorithm *Name* (⟨*parameter list*⟩)

    where *Name* is the name of the procedure and (⟨*parameter list*⟩) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type.

    As an example, the following algorithm finds and returns the maximum of n given numbers:

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

    In this algorithm (named Max), A and n are procedure parameters. *Result* and i are local variables.

Next we present two examples to illustrate the process of translating a problem into an algorithm.

Example 1.1 [Selection sort] Suppose we must devise an algorithm that sorts a collection of $n \geq 1$ elements of arbitrary type. A simple solution is given by the following