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-

ous design techniques that have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for you to devise new and useful algorithms. Many of the chapters of this book are organized around what we believe are the major methods of algorithm design. The reader may now wish to glance back at the table of contents to see what these methods are called. Some of these techniques may already be familiar, and some have been found to be so useful that books have been written about them. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. In this book we can only hope to give an introduction to these many approaches to algorithm formulation. All of the approaches we consider have applications in a variety of areas including computer science. But some important design techniques such as linear, nonlinear, and integer programming are not covered here as they are traditionally covered in other courses.

2. *How to validate algorithms* — Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as *algorithm validation*. The algorithm need not as yet be expressed as a program. It is sufficient to state it in any precise way. The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as *program proving* or sometimes as *program verification*. A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a *specification*, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. *How to analyze algorithms* — This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data. *Analysis of algorithms* or *performance analysis* refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows you to make quantitative judgments about the value of one algorithm over another. Another result is that it allows you to predict whether the software will meet any efficiency constraints that exist.

Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical. For each algorithm in the text, an analysis is also given. Analysis is more fully described in Section 1.3.2.

4. *How to test a program* — Testing a program consists of two phases: debugging and profiling (or performance measurement). *Debugging* is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. However, as E. Dijkstra has pointed out, "debugging can only point to the presence of errors, but not to their absence." In cases in which we cannot verify the correctness of output on sample data, the following strategy can be employed: let more than one programmer develop programs for the same problem, and compare the outputs produced by these programs. If the outputs match, then there is a good chance that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization. A description of the measurement of timing complexity can be found in Section 1.3.5. For some of the algorithms presented here, we show how to devise a range of data sets that will be useful for debugging and profiling.

These four categories serve to outline the questions we ask about algorithms throughout this book. As we can't hope to cover all these subjects completely, we content ourselves with concentrating on design and analysis, spending less time on program construction and correctness.

EXERCISES

1. Look up the words algorism and algorithm in your dictionary and write down their meanings.

2. The name al-Khowarizmi (algorithm) literally means "from the town of Khowarazm." This city is now known as Khiva, and is located in Uzbekistan. See if you can find this country in an atlas.

3. Use the WEB to find out more about al-Khowarizmi, e.g., his dates, a picture, or a stamp.