CHAPTER ONE

BASIC CONCEPTS

> *Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.*
>
> — AUGUSTA ADA, Countess of Lovelace (1843)

> *Practice yourself, for heaven's sake, in little things;*
> *and thence proceed to greater.*
>
> — EPICTETUS (*Discourses* IV.i)

1.1. ALGORITHMS

THE NOTION of an *algorithm* is basic to all of computer programming, so we should begin with a careful analysis of this concept.

The word "algorithm" itself is quite interesting; at first glance it may look as though someone intended to write "logarithm" but jumbled up the first four letters. The word did not appear in *Webster's New World Dictionary* as late as 1957; we find only the older form "algorism" with its ancient meaning, the process of doing arithmetic using Arabic numerals. During the Middle Ages, abacists computed on the abacus and algorists computed by algorism. By the time of the Renaissance, the origin of this word was in doubt, and early linguists attempted to guess at its derivation by making combinations like *algiros* [painful]+*arithmos* [number]; others said no, the word comes from "King Algor of Castile." Finally, historians of mathematics found the true origin of the word algorism: It comes from the name of a famous Persian textbook author, Abū ʻAbd Allāh Muḥammad ibn Mūsā al-Khwārizmī (c. 825) — literally, "Father of Abdullah, Mohammed, son of Moses, native of Khwārizm." The Aral Sea in Central Asia was once known as Lake Khwārizm, and the Khwārizm region is located in the Amu River basin just south of that sea. Al-Khwārizmī wrote the celebrated Arabic text *Kitāb al-jabr wa'l-muqābala* ("Rules of restoring and equating"); another word, "algebra," stems from the title of that book, which was a systematic study of the solution of linear and quadratic equations. [For notes on al-Khwārizmī's life and work, see H. Zemanek, *Lecture Notes in Computer Science* **122** (1981), 1–81.]

Gradually the form and meaning of *algorism* became corrupted; as explained by the *Oxford English Dictionary*, the word “passed through many pseudo-etymological perversions, including a recent *algorithm*, in which it is learnedly confused” with the Greek root of the word *arithmetic*. This change from “algorism” to “algorithm” is not hard to understand in view of the fact that people had forgotten the original derivation of the word. An early German mathematical dictionary, *Vollständiges mathematisches Lexicon* (Leipzig: 1747), gave the following definition for the word *Algorithmus*: “Under this designation are combined the notions of the four types of arithmetic calculations, namely addition, multiplication, subtraction, and division.” The Latin phrase *algorithmus infinitesimalis* was at that time used to denote “ways of calculation with infinitely small quantities, as invented by Leibniz.”

By 1950, the word algorithm was most frequently associated with Euclid’s algorithm, a process for finding the greatest common divisor of two numbers that appears in Euclid’s *Elements* (Book 7, Propositions 1 and 2). It will be instructive to exhibit Euclid’s algorithm here:

Algorithm E (*Euclid’s algorithm*). Given two positive integers m and n, find their *greatest common divisor*, that is, the largest positive integer that evenly divides both m and n.

E1. [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Reduce.] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1. ▮

Of course, Euclid did not present his algorithm in just this manner. The format above illustrates the style in which all of the algorithms throughout this book will be presented.

Each algorithm we consider has been given an identifying letter (E in the preceding example), and the steps of the algorithm are identified by this letter followed by a number (E1, E2, E3). The chapters are divided into numbered sections; within a section the algorithms are designated by letter only, but when algorithms are referred to in other sections, the appropriate section number is attached. For example, we are now in Section 1.1; within this section Euclid’s algorithm is called Algorithm E, while in later sections it is referred to as Algorithm 1.1E.

Each step of an algorithm, such as step E1 above, begins with a phrase in brackets that sums up as briefly as possible the principal content of that step. This phrase also usually appears in an accompanying *flow chart*, such as Fig. 1, so that the reader will be able to picture the algorithm more readily.

After the summarizing phrase comes a description in words and symbols of some *action* to be performed or some decision to be made. Parenthesized *comments*, like the second sentence in step E1, may also appear. Comments are included as explanatory information about that step, often indicating certain invariant characteristics of the variables or the current goals. They do not specify

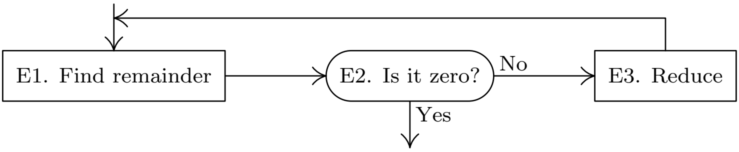

Fig. 1. Flow chart for Algorithm E.

actions that belong to the algorithm, but are meant only for the reader's benefit as possible aids to comprehension.

The arrow "$\leftarrow$" in step E3 is the all-important *replacement* operation, sometimes called *assignment* or *substitution*: "$m \leftarrow n$" means that the value of variable m is to be replaced by the current value of variable n. When Algorithm E begins, the values of m and n are the originally given numbers; but when it ends, those variables will have, in general, different values. An arrow is used to distinguish the replacement operation from the equality relation: We will not say, "Set $m = n$," but we will perhaps ask, "Does $m = n$?" The "$=$" sign denotes a condition that can be tested, the "$\leftarrow$" sign denotes an action that can be performed. The operation of *increasing n by one* is denoted by "$n \leftarrow n + 1$" (read "n is replaced by $n + 1$" or "n gets $n + 1$"). In general, "variable $\leftarrow$ formula" means that the formula is to be computed using the present values of any variables appearing within it; then the result should replace the previous value of the variable at the left of the arrow. Persons untrained in computer work sometimes have a tendency to say "n becomes $n + 1$" and to write "$n \to n + 1$" for the operation of increasing n by one; this symbolism can only lead to confusion because of its conflict with standard conventions, and it should be avoided.

Notice that the order of actions in step E3 is important: "Set $m \leftarrow n$, $n \leftarrow r$" is quite different from "Set $n \leftarrow r$, $m \leftarrow n$," since the latter would imply that the previous value of n is lost before it can be used to set m. Thus the latter sequence is equivalent to "Set $n \leftarrow r$, $m \leftarrow r$." When several variables are all to be set equal to the same quantity, we can use multiple arrows; for example, "$n \leftarrow r$, $m \leftarrow r$" may be written "$n \leftarrow m \leftarrow r$." To interchange the values of two variables, we can write "Exchange $m \leftrightarrow n$"; this action could also be specified by using a new variable t and writing "Set $t \leftarrow m$, $m \leftarrow n$, $n \leftarrow t$."

An algorithm starts at the lowest-numbered step, usually step 1, and it performs subsequent steps in sequential order unless otherwise specified. In step E3, the imperative "go back to step E1" specifies the computational order in an obvious fashion. In step E2, the action is prefaced by the condition "If $r = 0$"; so if $r \neq 0$, the rest of that sentence does not apply and no action is specified. We might have added the redundant sentence, "If $r \neq 0$, go on to step E3."

The heavy vertical line "▮" appearing at the end of step E3 is used to indicate the end of an algorithm and the resumption of text.

We have now discussed virtually all the notational conventions used in the algorithms of this book, except for a notation used to denote "subscripted" or "indexed" items that are elements of an ordered array. Suppose we have n quantities, $v_1, v_2, \ldots, v_n$; instead of writing v_j for the jth element, the notation $v[j]$ is often used. Similarly, $a[i,j]$ is sometimes used in preference to a doubly subscripted notation like a_{ij}. Sometimes multiple-letter names are used for variables, usually set in capital letters; thus `TEMP` might be the name of a variable used for temporarily holding a computed value, `PRIME[K]` might denote the `K`th prime number, and so on.

So much for the *form* of algorithms; now let us *perform* one. It should be mentioned immediately that the reader should *not* expect to read an algorithm as if it were part of a novel; such an attempt would make it pretty difficult to understand what is going on. An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it. The reader should always take pencil and paper and work through an example of each algorithm immediately upon encountering it in the text. Usually the outline of a worked example will be given, or else the reader can easily conjure one up. This is a simple and painless way to gain an understanding of a given algorithm, and all other approaches are generally unsuccessful.

Let us therefore work out an example of Algorithm E. Suppose that we are given $m = 119$ and $n = 544$; we are ready to begin, at step E1. (The reader should now follow the algorithm as we give a play-by-play account.) Dividing m by n in this case is quite simple, almost too simple, since the quotient is zero and the remainder is 119. Thus, $r \leftarrow 119$. We proceed to step E2, and since $r \neq 0$ no action occurs. In step E3 we set $m \leftarrow 544$, $n \leftarrow 119$. It is clear that if $m < n$ originally, the quotient in step E1 will always be zero and the algorithm will always proceed to interchange m and n in this rather cumbersome fashion. We could insert a new step at the beginning:

E0. [Ensure $m \geq n$.] If $m < n$, exchange $m \leftrightarrow n$.

This would make no essential change in the algorithm, except to increase its length slightly, and to decrease its running time in about one half of all cases.

Back at step E1, we find that $544/119 = 4 + 68/119$, so $r \leftarrow 68$. Again E2 is inapplicable, and at E3 we set $m \leftarrow 119$, $n \leftarrow 68$. The next round sets $r \leftarrow 51$, and ultimately $m \leftarrow 68$, $n \leftarrow 51$. Next $r \leftarrow 17$, and $m \leftarrow 51$, $n \leftarrow 17$. Finally, when 51 is divided by 17, we set $r \leftarrow 0$, so at step E2 the algorithm terminates. The greatest common divisor of 119 and 544 is 17.

So this is an algorithm. The modern meaning for algorithm is quite similar to that of *recipe, process, method, technique, procedure, routine, rigmarole,* except that the word "algorithm" connotes something just a little different. Besides merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, an algorithm has five important features:

1) *Finiteness.* An algorithm must always terminate after a finite number of steps. Algorithm E satisfies this condition, because after step E1 the value of r is *less* than n; so if $r \neq 0$, the value of n *decreases* the next time step E1 is encountered. A decreasing sequence of positive integers must eventually terminate, so step E1 is executed only a finite number of times for any given original

value of n. Note, however, that the number of steps can become arbitrarily large; certain huge choices of m and n will cause step E1 to be executed more than a million times.

(A procedure that has all of the characteristics of an algorithm except that it possibly lacks finiteness may be called a *computational method.* Euclid originally presented not only an algorithm for the greatest common divisor of numbers, but also a very similar geometrical construction for the "greatest common measure" of the lengths of two line segments; this is a computational method that does not terminate if the given lengths are incommensurable. Another example of a nonterminating computational method is a *reactive process*, which continually interacts with its environment.)

2) *Definiteness.* Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. The algorithms of this book will hopefully meet this criterion, but they are specified in the English language, so there is a possibility that the reader might not understand exactly what the author intended. To get around this difficulty, formally defined *programming languages* or *computer languages* are designed for specifying algorithms, in which every statement has a very definite meaning. Many of the algorithms of this book will be given both in English and in a computer language. An expression of a computational method in a computer language is called a *program.*

In Algorithm E, the criterion of definiteness as applied to step E1 means that the reader is supposed to understand exactly what it means to divide m by n and what the remainder is. In actual fact, there is no universal agreement about what this means if m and n are not positive integers; what is the remainder of -8 divided by $-\pi$? What is the remainder of $59/13$ divided by zero? Therefore the criterion of definiteness means we must make sure that the values of m and n are always positive integers whenever step E1 is to be executed. This is initially true, by hypothesis; and after step E1, r is a nonnegative integer that must be nonzero if we get to step E3. So m and n are indeed positive integers as required.

3) *Input.* An algorithm has zero or more *inputs*: quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs. These inputs are taken from specified sets of objects. In Algorithm E, for example, there are two inputs, namely m and n, both taken from the set of *positive integers.*

4) *Output.* An algorithm has one or more *outputs*: quantities that have a specified relation to the inputs. Algorithm E has one output, namely n in step E2, the greatest common divisor of the two inputs.

(We can easily *prove* that this number is indeed the greatest common divisor, as follows. After step E1, we have

$$m = qn + r,$$

for some integer q. If $r = 0$, then m is a multiple of n, and clearly in such a case n is the greatest common divisor of m and n. If $r \neq 0$, note that any number that divides both m and n must divide $m - qn = r$, and any number that divides both n and r must divide $qn + r = m$; so the set of common divisors of m and n is the same as the set of common divisors of n and r. In particular, the *greatest* common divisor of m and n is the same as the greatest common divisor of n and r. Therefore step E3 does not change the answer to the original problem.)

5) *Effectiveness.* An algorithm is also generally expected to be *effective*, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. Algorithm E uses only the operations of dividing one positive integer by another, testing if an integer is zero, and setting the value of one variable equal to the value of another. These operations are effective, because integers can be represented on paper in a finite manner, and because there is at least one method (the "division algorithm") for dividing one by another. But the same operations would *not* be effective if the values involved were arbitrary real numbers specified by an infinite decimal expansion, nor if the values were the lengths of physical line segments (which cannot be specified exactly). Another example of a noneffective step is, "If 4 is the largest integer n for which there is a solution to the equation $w^n + x^n + y^n = z^n$ in positive integers w, x, y, and z, then go to step E4." Such a statement would not be an effective operation until someone successfully constructs an algorithm to determine whether 4 is or is not the largest integer with the stated property.

Let us try to compare the concept of an algorithm with that of a cookbook recipe. A recipe presumably has the qualities of finiteness (although it is said that a watched pot never boils), input (eggs, flour, etc.), and output (TV dinner, etc.), but it notoriously lacks definiteness. There are frequent cases in which a cook's instructions are indefinite: "Add a dash of salt." A "dash" is defined to be "less than 1/8 teaspoon," and salt is perhaps well enough defined; but where should the salt be added — on top? on the side? Instructions like "toss lightly until mixture is crumbly" or "warm cognac in small saucepan" are quite adequate as explanations to a trained chef, but an algorithm must be specified to such a degree that even a computer can follow the directions. Nevertheless, a computer programmer can learn much by studying a good recipe book. (The author has in fact barely resisted the temptation to name the present volume "The Programmer's Cookbook." Perhaps someday he will attempt a book called "Algorithms for the Kitchen.")

We should remark that the finiteness restriction is not really strong enough for practical use. A useful algorithm should require not only a finite number of steps, but a *very* finite number, a reasonable number. For example, there is an algorithm that determines whether or not the game of chess can always be won by White if no mistakes are made (see exercise 2.2.3–28). That algorithm can solve a problem of intense interest to thousands of people, yet it is a safe bet that we will never in our lifetimes know the answer; the algorithm requires fantastically large amounts of time for its execution, even though it is finite. See also Chapter 8 for a discussion of some finite numbers that are so large as to actually be beyond comprehension.

In practice we not only want algorithms, we want algorithms that are *good* in some loosely defined aesthetic sense. One criterion of goodness is the length of time taken to perform the algorithm; this can be expressed in terms of the number of times each step is executed. Other criteria are the adaptability of the algorithm to different kinds of computers, its simplicity and elegance, etc.

We often are faced with several algorithms for the same problem, and we must decide which is best. This leads us to the extremely interesting and all-important field of *algorithmic analysis*: Given an algorithm, we want to determine its performance characteristics.

For example, let's consider Euclid's algorithm from this point of view. Suppose we ask the question, "Assuming that the value of n is known but m is allowed to range over all positive integers, what is the *average* number of times, T_n, that step E1 of Algorithm E will be performed?" In the first place, we need to check that this question does have a meaningful answer, since we are trying to take an average over infinitely many choices for m. But it is evident that after the first execution of step E1 only the remainder of m after division by n is relevant. So all we must do to find T_n is to try the algorithm for $m = 1$, $m = 2$, …, $m = n$, count the total number of times step E1 has been executed, and divide by n.

Now the important question is to determine the *nature* of T_n; is it approximately equal to $\frac{1}{3}n$, or $\sqrt{n}$, for instance? As a matter of fact, the answer to this question is an extremely difficult and fascinating mathematical problem, not yet completely resolved, which is examined in more detail in Section 4.5.3. For large values of n it is possible to prove that T_n is approximately $\bigl(12(\ln 2)/\pi^2\bigr)\ln n$, that is, proportional to the *natural logarithm* of n, with a constant of proportionality that might not have been guessed offhand! For further details about Euclid's algorithm, and other ways to calculate the greatest common divisor, see Section 4.5.2.

Analysis of algorithms is the name the author likes to use to describe investigations such as this. The general idea is to take a particular algorithm and to determine its quantitative behavior; occasionally we also study whether or not an algorithm is optimal in some sense. The *theory of algorithms* is another subject entirely, dealing primarily with the existence or nonexistence of effective algorithms to compute particular quantities.

So far our discussion of algorithms has been rather imprecise, and a mathematically oriented reader is justified in thinking that the preceding commentary makes a very shaky foundation on which to erect any theory about algorithms. We therefore close this section with a brief indication of one method by which the concept of algorithm can be firmly grounded in terms of mathematical set theory. Let us formally define a *computational method* to be a quadruple (Q, I, Ω, f), in which Q is a set containing subsets I and Ω, and f is a function from Q into itself. Furthermore f should leave Ω pointwise fixed; that is, $f(q)$ should equal q for all elements q of Ω. The four quantities Q, I, Ω, f are intended to represent respectively the states of the computation, the input, the output, and the computational rule. Each input x in the set I defines a *computational* *sequence*, $x_0, x_1, x_2, \ldots$, as follows:

$$x_0 = x \qquad \text{and} \qquad x_{k+1} = f(x_k) \quad \text{for} \quad k \ge 0. \tag{1}$$

The computational sequence is said to *terminate in k steps* if k is the smallest integer for which x_k is in Ω, and in this case it is said to produce the output x_k from x. (Notice that if x_k is in Ω, so is x_{k+1}, because $x_{k+1} = x_k$ in such a case.) Some computational sequences may never terminate; an *algorithm* is a computational method that terminates in finitely many steps for all x in I.

Algorithm E may, for example, be formalized in these terms as follows: Let Q be the set of all singletons (n), all ordered pairs (m,n), and all ordered quadruples $(m,n,r,1)$, $(m,n,r,2)$, and $(m,n,p,3)$, where m, n, and p are positive integers and r is a nonnegative integer. Let I be the subset of all pairs (m,n) and let Ω be the subset of all singletons (n). Let f be defined as follows:

$$\begin{aligned} f\bigl((m,n)\bigr) &= (m,n,0,1); \qquad f\bigl((n)\bigr) = (n); \\ f\bigl((m,n,r,1)\bigr) &= (m,\ n,\ \text{remainder of } m \text{ divided by } n,\ 2); \\ f\bigl((m,n,r,2)\bigr) &= (n) \quad \text{if} \quad r = 0, \qquad (m,n,r,3) \quad \text{otherwise}; \\ f\bigl((m,n,p,3)\bigr) &= (n,p,p,1). \end{aligned} \tag{2}$$

The correspondence between this notation and Algorithm E is evident.

This formulation of the concept of an algorithm does not include the restriction of effectiveness mentioned earlier. For example, Q might denote infinite sequences that are not computable by pencil and paper methods, or f might involve operations that mere mortals cannot always perform. If we wish to restrict the notion of algorithm so that only elementary operations are involved, we can place restrictions on Q, I, Ω, and f, for example as follows: Let A be a finite set of letters, and let A^* be the set of all strings on A (the set of all ordered sequences $x_1 x_2 \ldots x_n$, where $n \ge 0$ and x_j is in A for $1 \le j \le n$). The idea is to encode the states of the computation so that they are represented by strings of A^*. Now let N be a nonnegative integer and let Q be the set of all (σ, j), where σ is in A^* and j is an integer, $0 \le j \le N$; let I be the subset of Q with $j = 0$ and let Ω be the subset with $j = N$. If θ and σ are strings in A^*, we say that θ occurs in σ if σ has the form $\alpha\theta\omega$ for strings α and ω. To complete our definition, let f be a function of the following type, defined by the strings θ_j, ϕ_j and the integers a_j, b_j for $0 \le j < N$:

$$\begin{aligned} &f\bigl((\sigma,j)\bigr) = (\sigma, a_j) && \text{if } \theta_j \text{ does not occur in } \sigma; \\ &f\bigl((\sigma,j)\bigr) = (\alpha\phi_j\omega, b_j) && \text{if } \alpha \text{ is the shortest possible string for which } \sigma = \alpha\theta_j\omega; \\ &f\bigl((\sigma,N)\bigr) = (\sigma, N). \end{aligned} \tag{3}$$

Every step of such a computational method is clearly effective, and experience shows that pattern-matching rules of this kind are also powerful enough to do anything we can do by hand. There are many other essentially equivalent ways to formulate the concept of an effective computational method (for example, using Turing machines). The formulation above is virtually the same as that given by A. A. Markov in his book *The Theory of Algorithms* [*Trudy Mat. Inst. Akad. Nauk* **42** (1954), 1–376], later revised and enlarged by N. M. Nagorny (Moscow: Nauka, 1984; English edition, Dordrecht: Kluwer, 1988).

EXERCISES

1. [*10*] The text showed how to interchange the values of variables m and n, using the replacement notation, by setting $t \leftarrow m$, $m \leftarrow n$, $n \leftarrow t$. Show how the values of *four* variables (a, b, c, d) can be rearranged to (b, c, d, a) by a sequence of replacements. In other words, the new value of a is to be the original value of b, etc. Try to use the minimum number of replacements.

2. [*15*] Prove that m is always greater than n at the beginning of step E1, except possibly the first time this step occurs.

3. [*20*] Change Algorithm E (for the sake of efficiency) so that all trivial replacement operations such as "$m \leftarrow n$" are avoided. Write this new algorithm in the style of Algorithm E, and call it Algorithm F.

4. [*16*] What is the greatest common divisor of 2166 and 6099?

▶ **5.** [*12*] Show that the "Procedure for Reading This Set of Books" that appears after the preface actually fails to be a genuine algorithm on at least three of our five counts! Also mention some differences in format between it and Algorithm E.

6. [*20*] What is T_5, the average number of times step E1 is performed when $n = 5$?

▶ **7.** [*M21*] Suppose that m is known and n is allowed to range over all positive integers; let U_m be the average number of times that step E1 is executed in Algorithm E. Show that U_m is well defined. Is U_m in any way related to T_m?

8. [*M25*] Give an "effective" formal algorithm for computing the greatest common divisor of positive integers m and n, by specifying θ_j, ϕ_j, a_j, b_j as in Eqs. (3). Let the input be represented by the string $a^m b^n$, that is, m a's followed by n b's. Try to make your solution as simple as possible. [*Hint:* Use Algorithm E, but instead of division in step E1, set $r \leftarrow |m - n|$, $n \leftarrow \min(m, n)$.]

▶ **9.** [*M30*] Suppose that $C_1 = (Q_1, I_1, \varOmega_1, f_1)$ and $C_2 = (Q_2, I_2, \varOmega_2, f_2)$ are computational methods. For example, C_1 might stand for Algorithm E as in Eqs. (2), except that m and n are restricted in magnitude, and C_2 might stand for a computer program implementation of Algorithm E. (Thus Q_2 might be the set of all states of the machine, i.e., all possible configurations of its memory and registers; f_2 might be the definition of single machine actions; and I_2 might be the set of initial states, each including the program that determines the greatest common divisor as well as the particular values of m and n.)

Formulate a set-theoretic definition for the concept "C_2 is a representation of C_1" or "C_2 simulates C_1." This is to mean intuitively that any computation sequence of C_1 is mimicked by C_2, except that C_2 might take more steps in which to do the computation and it might retain more information in its states. (We thereby obtain a rigorous interpretation of the statement, "Program X is an implementation of Algorithm Y.")