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 $0 \le r < n$.)

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. ▌

...

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 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.

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.

Πηγή : The Art of Computer Programming: Volume 1: Fundamental Algorithms