$\ldots, a_1)^T$. Then

$$X A = \sum_{i=1}^{n+1} a_i x^{i-1}.$$

X has n nondependent columns, which implies that n multiplications are required. The result requires that the algorithm evaluate any polynomial, given its coefficients. Specific polynomials can often be evaluated with fewer multiplications. Similarly, if the polynomial is specified by parameters other than its coefficients, a saving in the number of multiplications is possible.

The one facet of problem complexity that is probably the most intriguing is the lack of nontrivial lower bounds for various problems. Almost all known lower bounds are either linear in the size of the problem or have been obtained by restricting the classes of algorithms. The notable exceptions are lower bounds obtained by the diagonalization techniques of recursive function theory. One of the major goals of computer scientists working in the analysis of algorithms is to close the gap in our knowledge of problem complexity. Hopefully, the next decade will provide powerful new tools in the area and startling improvements in the efficiency of algorithms.

Rererences

1968, 1969, 1973. Knuth, D. E. *The Art of Computer Programming* 1, 2, 3. Reading, MA: Addison-Wesley.

1974. Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley.

1976. Wirth, Niklaus, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice-Hall.

J. E. Hopcroft and J. E. Musinski

ALGORITHMS, THEORY OF

For articles on related subjects *see* Algorithm; Algorithms, Analysis of; Computability; Computational Complexity; Decidability; Formal Languages; NP-Complete Problems; and Turing Machine.

The meaning of the word *algorithm*, like the meaning of most other words commonly used in the English language, is somewhat vague. In order to have a *theory of algorithms*, we need a mathematically precise definition of an algorithm. However, in giving such a precise definition, we run the risk of not reflecting exactly the intuitive notion behind the word. The finding of a mathematically precise replacement of the notion of algorithm was the earliest problem in the theory of algorithms. Many authors have tried to capture the essence of the intuitive notion of an algorithm. We give four examples.

Hermes (1965). "An algorithm is a general procedure such that for any appropriate question the answer can be obtained by the use of a simple computation according to a specified method. . . . [A] general procedure [is] a process the execution of which is clearly specified to the smallest details."

Minsky (1967). " . . . an effective procedure is a set of rules which tells us, from moment to moment, precisely how to behave."

Rogers (1967). " . . . an algorithm is a clerical (i.e., deterministic, bookkeeping) procedure which can be applied to any of a certain class of symbolic *inputs* and which will eventually yield, for each such input, a corresponding symbolic *output*."

Hopcroft and Ullman (1969). "A *procedure* is a finite sequence of instructions that can be mechanically carried out, such as a computer program. . . . A procedure which always terminates is called an *algorithm*."

Note that what Hermes calles "a general procedure" is what Minsky calls an "effective procedure." Other terms are also used in the literature, and some authors use the word "algorithm" to denote any procedure whatsoever. In the remainder of this article the Hopcroft and Ullman terminology will be used.

An important fact to note is that the notion of a procedure cannot be divorced from the environment in which it operates. What may be a procedure in certain situations, may not be considered a procedure in other situations. For example, the instructions of a computer program are not usually understood by most people. Alternatively, the description of a chess game that appears in a newspaper is a perfectly clear algorithm for a chess player who wants to reproduce the game, but it is quite meaningless to people who do not play chess. Thus, when we talk about a procedure as a finite sequence of instructions, we assume that whoever is supposed to carry out those instructions, be it human or machine, understands them in the same way as whoever gave those instructions.

Another sense in which the environment influences the notions of procedure and algorithm is indicated by the following examples. If the instruction requires us to take the integral part of the square root of a number, such an instruction can be carried out if we are dealing with positive integers only, but it cannot always be carried out if we are dealing with both positive and negative integers.

from: A. Ralston (ed.), Encyclopedia of Computer Science & Engineering, 2nd ed.
(New York: Van Nostrand Reinhold, 1983)

Thus, the same set of instructions may or may not be a procedure, depending on the subset of integers for which it is intended. Alternatively, we can easily give a procedure that, given an integer x, keeps subtracting 1 until 0 is reached and then stops. Such a procedure will be an algorithm if we intend to use it for positive integers only, but it will not be an algorithm if we also intend to apply it to negative integers.

The recognition of whether or not a sequence of instructions is a procedure or an algorithm is a subjective affair. No precise theory can be built on the vague definitions given above. In trying to build a precise theory, one must examine the situations in which the notion of algorithm is used. In the theory of computation, one is mainly concerned with algorithms that are used either for computing functions or for deciding predicates.

A *function* f with domain D and range R is a definite correspondence by which there is associated with each element x of the domain D (referred to as the "argument") a single element $f(x)$ of the range R (called the "value"). The function f is said to be "computable" (in the intuitive sense) if there exists an algorithm which, for any given x in D, provides us with the value $f(x)$.

A *predicate* P with domain D is a property of the elements of D which each particular element of D either has or does not have. If x in D has the property P, we say that $P(x)$ is true; otherwise, we say that $P(x)$ is false. The predicate P is said to be *decidable* (in the intuitive sense) if there exists an algorithm which, for any given x in D, provides us with a definite answer to the question of whether or not $P(x)$ is true.

The computability of functions and the decidability of predicates are very closely related notions because we can associate with each predicate P a function f with range $\{0,1\}$ such that, for all x in the common domain D of P and f, $f(x) = 0$ if $P(x)$ is true and $f(x) = 1$ if $P(x)$ is false. Clearly, P is decidable if and only if f is computable. For this reason we will hereafter restrict our attention to the computability of functions.

A further restriction customary in the theory of algorithms is to consider only functions whose domain and range are both the set of nonnegative integers. This is reasonable, since in those situations where the notion of a procedure makes any sense at all, it is usually possible to *represent* elements of the domain and the range by nonnegative integers. For example, if the domain comprises pairs of nonnegative integers, as in the case with an arithmetic function of two arguments, we can represent the pair (a,b) by the number 2^a3^b in an effective one-to-one fashion. If the domain comprises strings of symbols over an alphabet of 15 letters, we can consider the letters to be nonzero hexadecimal digits, and assign that nonnegative integer to a string that is denoted by the string in the hexadecimal notation. The device of representing elements of a set D by nonnegative integers is referred to as "arithmetization" or "Gödel numbering," after the logician K. Gödel, who used it to prove the undecidability of certain predicates about formal logic. From now on we will be exclusively concerned with functions whose domain and range are subsets of the set of nonnegative integers.

In order to show that a certain function is computable, it is sufficient to give an algorithm that computes it. But without a precise definition of an algorithm, all such demonstrations are open to question. The situation is even more uncertain if we want to show that a given function is uncomputable, i.e., that no algorithm whatsoever computes it. In order to avoid such uncertainty, we need a mathematically precise definition of a computable function.

It is clear from the way in which algorithms are discussed above that for a proper algorithm we ought to be able to construct a machine that carries out the instructions of the algorithm. One possible way of making precise the concept of a computable function is to define an appropriate type of machine, and then define a function to be computable if and only if it can be computed by such a machine. This has indeed been done. The machine usually used for this purpose is the so-called Turing machine (*q.v.*). This simple device has a tape and a read-write head, together with a control that may be in one of finitely many states. The tape is used to represent numbers. A function f is called computable if there exists a Turing machine that, given a tape representing an argument x, eventually halts with the tape representing the value $f(x)$. Since a precise definition of a Turing machine can be given, the notion of a computable function has become a precise mathematical notion.

The question arises whether or not it is indeed the case that a function is computable in the intuitive sense if and only if it is computable by a Turing machine. The claim that this is true is usually referred to as *Church's thesis* (sometimes as *Turing's thesis*). Such a claim can never by "proved," since one of the two notions whose equivalence is claimed is mathematically imprecise. However, there are many convincing arguments in support of Church's thesis, and an overwhelming majority of workers in the theory of algorithms accept its validity. One of the strongest arguments in support of Church's thesis is the fact that all of the many diverse attempts at precisely defining the concept of computable function have ended up with defining exactly the same set of functions.

Given a precise definition of a computable function, it is now possible to show for particular functions that they are computable. Conversely, it becomes possible to prove that certain functions are not computable. We will give two examples.

Example 1. Consider the following problem. Give an algorithm that, for any Turing machine, decides whether or not the machine eventually stops if it is started on an empty tape. This problem is called the "blank-tape halting problem." The required algorithm would be considered a *solution* of the problem. A proof that there is no such algorithm would be said to show the (effective) *unsolvability* of the problem.

The blank-tape halting problem is in fact unsolvable. This is proved by rephrasing the problem into a problem about the computability of a function, as follows: Turing machines can be Gödel-numbered in an effective manner; i.e., there exists an algorithm which for any Turing machine will give its Gödel number. Furthermore, this can be done in such a way that every nonnegative integer is the Gödel number of some Turing machine. Let f be the function defined as follows.

$$f(x) = \begin{cases} 0 & \text{if } n \text{ is the Gödel number of a Turing machine that eventually stops if started on the blank tape;} \\ 1 & \text{otherwise.} \end{cases}$$

It is easy to see that f is computable if and only if the blank-tape halting problem is solvable. The unsolvability of the blank-tape halting problem is proved by showing that the assumption that f is computable leads to a contradiction.

Example 2. Our second example indicates that there are unsolvable problems in classical mathematics. The followings problem is known as "Hilbert's tenth problem" (after the German mathematician David Hilbert, 1862–1943):

> Given a diophantine equation [an equation of the form $E = 0$, where E is a polynomial with integer coefficients; e.g., $xy^2 - 2x^2 + 3 = 0$] with any variables, give a procedure with which it is possible to decide after a finite number of operations whether or not the equation has a solution in integers.

Although this problem was stated by Hilbert in 1900 (long before there was such a thing as a theory of algorithms), it has only been recently that the Russian mathematician I. Matiajasevitch has shown it to be unsolvable.

That there are clearly defined problems, like the two given above, that cannot be solved by any computer-like device is probably the most striking aspect of the theory of algorithms. A whole superstructure has been built on such results, and there are methods to find out not only whether something is uncomputable, but also how badly it is uncomputable (see Rogers, 1967).

A typical question that one may ask is the following: Suppose we had a device which, for any given Turing machine, told us whether or not the Turing machine will eventually stop on the blank tape. Can we write an "algorithm" that makes use of this device and solves Hilbert's tenth problem? It has been known for some time that such an "algorithm" exists. In this sense, Hilbert's tenth problem is *reducible* to the blank-tape halting problem. It is the proof that the reverse is also true which gave us the unsolvability of Hilbert's tenth problem. Two problems that are both reducible to the other are said to be *equivalent*. Most of the theory of algorithms has, until recently, concerned itself with questions of the reducibility and equivalence of various unsolvable problems.

In recent years a new trend has developed. Much of the activity in the theory of algorithms began to concern itself with computable functions, decidable predicates, and solvable problems. Questions about the nature of the algorithms, the type of devices that can be used for the computation, and about the difficulty or complexity of the computation have been investigated and are discussed in other articles.

References

1965. Hermes, H. *Enumerability, Decidability, Computability.* Berlin, Germany: Springer-Verlag.

1967. Minsky, M. *Computation: Finite and Infinite Machines.* Englewood Cliffs, NJ: Prentice-Hall.

1967. Rogers, H. *Theory of Recursive Functions and Effective Computability.* New York: McGraw-Hill.

1969. Hopcroft, J. E. and Ullman, J. D. *Formal Lanuages and Their Relation to Automata.* Reading, MA: Addison-Wesley.

G. T. Herman

ALLOCATION, STORAGE.

See Storage Allocation.

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

For articles on related terms *see* American Society for Information Science; Association for Computing Machinery; Association for Educational Data Systems; International Federation for Information Processing; Institute of Electrical and Electronic Engineers—Computer Society; and Society for Industrial and Applied Mathematics.