Introduction

Algorithms, as demonstrated by their presence in the evolution of technology, mathematics and more generally of everyday life, play a catalytic role in our lives but also more generally in the world.

At same time they are very useful in our lives and their better exploit brings positive results. Although they are complex, they generate more results, in order to improve our lives.

Their history, which involves many disciplines of life and science, is great as their value. There will be a recursion in history to investigate their names, their original origins, their definition and their use in various machines and systems in different ages.

Two of the most important and most known algorithms of Mathematics will be included: the Euclidean algorithm for find the greatest common measure of two numbers and the approximation method of chord, with iterative process.

Generally

Algorithm is a sequence of clear rational operation aiming at the solution of the problem that is, the production of necessary output for every acceptable input in finite time. Algorithm is also a description of the way by which we can successfully perform a task or procedure.

The reference to instructions in the above definition implies the existence of something (a man or mechanism) which is able to understand and follow specific instructions. This entity is called computer, bearing in mind that in the pre-computer era, the term computer denoted someone who performed mathematical calculations.

Today, of course «computers» are invisible electronic devices that have become an indispensable part of any of our activities. However they are not necessary for the meaning of algorithm although the majority of algorithms is going to take shape in a computer. All over the world many field of science have incorporated algorithms. Some of them are Mathematics, Physics, Computer Science, Engineering and humanitarian studies with the emotional intelligence they use.

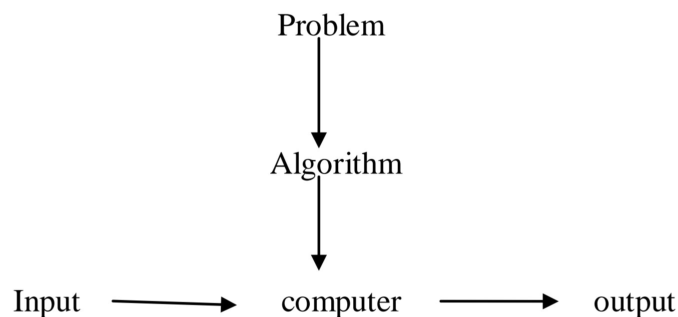
Algorithms are essential to the way computers process data. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system.

For some such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is,

any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by flow of control.

So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, "mechanical" means. Unique to this conception of formalized algorithms is the assignment operation, setting the value of a variable. It derives from the intuition of "memory" as a scratchpad. There is an example below of such an assignment.



The computational process and the algorithms terminate at some time and that is why by definitions they have their finite time as criterion. In the above diagram after using the appropriate algorithm the problem is solved and thus terminates

However, terminating or not terminating of a process and thus of an algorithm is one of its characteristics. A big source of errors in the creation of an algorithm is that under certain circumstances the process being described may not terminate while it is supposed to terminate.

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

For the computers' algorithms to function it is necessary to be a language for communication between the scientist (the human) and the machines so that the operations of the algorithm can be recorded. In the beginning they tried to use a natural language but then they noticed that it was difficult for the computers to comprehend it since today they process the algorithms more.

In order for a computer to execute the operations of an algorithm the language to be used must be as simple as possible. Such algorithms (that can function in the simplest possible way) are called programs and these programs are written in a programming language. Today there are hundreds of programming language and some of the most well-known are: Fortran, Cobol, Pl/C, Pascal, Ada, Basic, Logo and others. Many who want to note down the algorithm before they transfer it to the computer it is recorded in pseudocodes. But even this interim «language» or code does not differ from the natural language using its own alphabet. Today pseudocodes are use more for educational purposes and less in the use of scientific algorithms.

It's worth pointing out that every programming language uses its own grammatical and syntactic rules that dictate the use of this vocabulary. In most cases vocabulary consists of specific mathematical symbols and several English words and the grammar rules are simple enough to allow for the interpretation of programs by computer. Because of differences in vocabulary and grammar programming languages differ according to the kind of function they allow.

In order for an algorithm to be executed correctly it must be syntactically correct when there is an error involved in the syntax of the algorithms, this error is called syntactic correctness is a necessary requirement to interpret a computer program. Exceptions appear only when a processor (of any kind even the human mind) is so wise as to guess the form of operation at a syntactic error.

Apart from the syntactic errors there are errors of meaning relating to the meaning of specific forms of expressions in a language and the logical errors relating to a not so good description of a process.

An algorithms is correct when it includes steps executed the one after the other. Then we say that such an algorithm is a sequence of steps.

When an algorithm is a sequence of steps it does not have the ability to change this sequence according to circumstances. For this many algorithms have to ability to change according to circumstances with an order they contain. This ability is called selection and is realized under a condition. Every condition includes one "if" and one "else" so predicting the unstable factors.

A characteristics of the algorithms is the repetition of processes it has performed without been fed with new values and sums. This process is called iteration and it is very useful in an algorithm.

Simple Example

Algorithm LargestNumber

Input: A non-empty list of numbers L.

Output: The largest number in the list L.

largest ← L_0

for each item in the list (Length(L)≥1), do

if the item > largest, then

largest ← the item

return largest

History

In the beginning mainly in Middle Aged this term meant only the execution of four calculations with Arabic numbers and logical rules, connected to the decimal numbering system. Algorithmists were called those who followed the new method contrary to the abacists who continued to calculate using the abacus. Algorithmists followed the rules written in a book of the Arab Mathematician Al- Kuarismi who lived in Bagdad around 830 A.D. This book was known in Middle Ages as liber Algorismi and the and the word algorithm is a misspelling of the writer’s name.

Euclid’s Algorithm

Euclid’s algorithm appears as Proposition II in Book VII ("Elementary Number Theory") of his Elements. Euclid poses the problem: "Given two numbers not prime to one another, to find their greatest common measure". He defines "A number [to be] a multitude composed of units": a counting number, a positive integer not including 0. And to "measure" is to place a shorter measuring length s successively (q times) along longer length l until the remaining portion r is less than the shorter length s. In modern words, remainder r = l - q*s, q being the quotient, or remainder r is the "modulus", the integer-fractional part left over after the division.

For Euclid’s method to succeed, the starting lengths must satisfy two requirements: (i) the lengths must not be 0, AND (ii) the subtraction must be “proper”, a test must guarantee that the smaller of the two numbers is subtracted from the larger (alternately, the two can be equal so their subtraction yields 0).

Euclid's original proof adds a third: the two lengths are not prime to one another. Euclid stipulated this so that he could construct a reduction ad absurdum proof that the two numbers' common measure is in fact the greatest.

Computer (computor) language for Euclid's algorithm

Only a few instruction types are required to execute Euclid's algorithm—some logical tests (conditional GOTO), unconditional GOTO, assignment (replacement), and subtraction.

A location is symbolized by upper case letter(s), e.g. S, A, etc.

The varying quantity (number) in a location will be written in lower case letter(s) and (usually) associated with the location's name. For example, location L at the start might contain the number l = 3009.

An inelegant program for Euclid's algorithm

INPUT:

1 [Into two locations L and S put the numbers l and s that represent the two lengths]: INPUT L, S

2 [Initialize R: make the remaining length r equal to the starting/initial/input length l] R ← L

E: [Insure r ≥ s.]

3 [Insure the smaller of the two numbers is in S and the larger in R]: IF R > S THEN the contents of L is the larger number so skip over the exchange-steps 4, 5 and 6: GOTO step 6 ELSE swap the contents of R and S.]

4 L ← R (this first step is redundant, but will be useful for later discussion).

5 R ← S

6 S ← L

E1:[Find remainder]: Until the remaining length r in R is less than the shorter length s in S, repeatedly subtract the measuring number s in S from the remaining length r in R.

7 IF S > R THEN done measuring so GOTO 10 ELSE measure again,

8 R ← R - S

9 [Remainder-loop]: GOTO 7.

Is the remainder 0?]: EITHER (i) the last measure was exact and the remainder in R is 0 program can halt, OR (ii) the algorithm must continue: the last measure left a remainder in R less than measuring number in S.

10 IF R = 0 then done so GOTO step 15 ELSE continue to step 11,

Interchange s and r: The nut of Euclid's algorithm. Use remainder r to measure what was previously smaller number s:; L serves as a temporary location.

11 L ← R

12 R ← S

13 S ← L

14 Repeat the measuring process: GOTO 7

OUTPUT:

15 [Done. S contains the greatest common divisor]: PRINT S

DONE:

16 HALT, END, STOP.

An elegant program for Euclid's algorithm in Basic

REM Euclid's algorithm for greatest common divisor

PRINT "Type two integers greater than 0"

INPUT A,B

IF B=0 THEN GOTO 80

IF A > B THEN GOTO 60

LET B=B-A

GOTO 20

LET A=A-B

GOTO 20

PRINT A

END

Secant method

In numerical analysis, the secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The

secant method can be thought of as a finite difference approximation of Newton's method. However, the method was developed independently of Newton's method, and predated the latter by over 3000 years.

The method

The secant method is defined by the recurrence relation

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root.

Derivation of the method

Starting with initial values x_0 and x_1, we construct a line through the points $(x_0,f(x_0))$ and $(x_1,f(x_1))$. In point-slope form, this line has the equation

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1)$$

For y=0 to

$$0 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1)$$

The solution is

$$x = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

We then use this value of x as x_2 and repeat the process using x_1 and x_2 instead of x_0 and x_1. We continue this process, solving for x_3, x_4, etc., until we reach a sufficiently high level of precision (a sufficiently small difference between x_n and x_{n-1}).

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

...

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

Comparison with other root-finding methods

The secant method does not require that the root remain bracketed like the bisection method does, and hence it does not always converge. The false position method uses the same formula as the secant method. However, it does not apply the formula on x_{n-1} and x_n, like the secant method, but on x_n and on the last iterate x_k such that $f(x_k)$ and $f(x_n)$ have a different sign. This means that the false position method always converges.

The recurrence formula of the secant method can be derived from the formula for Newton's method

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

by using the finite difference approximation

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}.$$

If we compare Newton's method with the secant method, we see that Newton's method converges faster (order 2 against $\alpha \approx 1.6$). However, Newton's method requires the evaluation of both f and its derivative at every step, while the secant method only requires the evaluation of f. Therefore, the secant method may well be faster in practice. For instance, if we assume that evaluating f takes as much time as evaluating its derivative and we neglect all other costs, we can do two steps of the secant method (decreasing the logarithm of the error by a factor $\alpha^2 \approx 2.6$) for the same cost as one step of Newton's method (decreasing the logarithm of the error by a factor 2), so the secant method is faster. If however we consider parallel processing for the evaluation of the derivative, Newton's method proves its worth, being faster in time, though still spending more steps.