MATHEMATICAL FUNCTIONS AND ALGORITHMS

AskarovMamit

Candidate of physical and Mathematical Sciences of the Department of mathematics teaching methodology of Nukus State Pedagogical Institute docent, Uzbekistan

ANNOTATION

The article about algorithms is used as specifications for performing calculations and data processing. More advanced algorithms can perform automated deductions (referred to as automated reasoning) and use mathematical and logical tests to divert the code execution through various routes (referred to as automated decision-making).

Key words:code execution, mathematics and computer science, algorithm

DOI Number: 10.48047/NQ.2022.20.20.NQ109123 **NeuroQuantology2022;20(20): 1199-1202**

INTRODUCTION

In mathematics and computer science, an algorithm (/ˈælgərɪðəm/ (listen)) is a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation. Algorithms are used as specifications for performing calculations and data processing. More advanced algorithms can perform automated deductions (referred to as automated reasoning) and use mathematical and logical tests to divert the code execution through various routes (referred to as automated decision-making). Using human characteristics as descriptors of machines in metaphorical ways was already practiced by Alan Turing with terms such as "memory", "search" and "stimulus".

In contrast, a heuristic is an approach to problem solving that may not be fully specified or may not guarantee correct or optimal results, especially in problem domains where there is no well-defined correct or optimal result.

As an effective method, an algorithm can be expressed within a finite amount of space and time, and in a well-defined formal language for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

"Elegant" (compact) programs, "good" (fast) programs : The notion of "simplicity and elegance" appears informally in Knuth and precisely in Chaitin:

Knuth: " we want good algorithms in some loosely defined aesthetic sense. One criterion ... is the length of time taken to perform the algorithm .... Other criteria are adaptability of the algorithm to computers, its simplicity, and elegance, etc."

Chaitin: " a program is 'elegant,' by which I mean that it's the smallest possible program for producing the output that it does"

Chaitin prefaces his definition with: "I'll show you can't prove that a program is 'elegant'"—such a proof would solve the Halting problem (ibid).

Algorithm versus function computable by an algorithm: For a given function multiple algorithms may exist. This is true, even without expanding the available instruction set available to the programmer. Rogers observes that "It is ... important to distinguish between the notion of algorithm, i.e. procedure and the notion of function computable by algorithm, i.e. mapping yielded by procedure. The same function may have several different algorithms".

Unfortunately, there may be a tradeoff between goodness (speed) and elegance (compactness)—an elegant program may take

eISSN 1303-5150

more steps to complete a computation than one less elegant. An example that uses Euclid's algorithm appears below.

Computers (and computors), models of computation: A computer (or human "computer") is a restricted type of machine, a "discrete deterministic mechanical device" that blindly follows its instructions. Melzak's and Lambek's primitive models reduced this notion to four elements: (i) discrete, distinguishable locations, (ii) discrete, indistinguishable counters (iii) an agent, and (iv) a list of instructions that are effective relative to the capability of the agent.

Minsky describes a more congenial variation of Lambek's "abacus" model in his "Very Simple Bases for Computability". Minsky's machine proceeds sequentially through its five (or six, depending on how one counts) instructions unless either a conditional IF-THEN GOTO or an unconditional GOTO changes program flow out of sequence. Besides HALT, Minsky's machine includes three assignment (replacement, substitution) operations: ZERO (e.g. the contents of location replaced by 0: L ← 0), SUCCESSOR (e.g. L ← L+1), and DECREMENT (e.g. L ← L − 1). Rarely must a programmer write "code" with such a limited instruction set. But Minsky shows (as do Melzak and Lambek) that his machine is Turing complete with only four general types of instructions: conditional GOTO, unconditional GOTO, assignment/replacement/substitution, and HALT. However, a few different assignment instructions (e.g. DECREMENT, INCREMENT, and ZERO/CLEAR/EMPTY for a Minsky machine) are also required for Turing-completeness; their exact specification is somewhat up to the designer. The unconditional GOTO is convenient; it can be constructed by initializing a dedicated location to zero e.g. the instruction " Z ← 0 "; thereafter the instruction IF Z=0 THEN GOTO xxx is unconditional.

Simulation of an algorithm: computer (computor) language: Knuth advises the reader that "the best way to learn an algorithm is to try it . . . immediately take pen and paper and work through an example". But what about a simulation or execution of the real thing? The programmer must translate the algorithm into a language that the simulator/computer/computor can effectively execute. Stone gives an example of this: when computing the roots of a quadratic equation the computer must know how to take a square root. If they don't, then the algorithm, to be effective, must provide a set of rules for extracting a square root.

This means that the programmer must know a "language" that is effective relative to the target computing agent (computer/computor).

But what model should be used for the simulation? Van Emde Boas observes "even if we base complexity theory on abstract instead of concrete machines, the arbitrariness of the choice of a model remains. It is at this point that the notion of simulation enters". When speed is being measured, the instruction set matters. For example, the subprogram in Euclid's algorithm to compute the remainder would execute much faster if the programmer had a "modulus" instruction available rather than just subtraction (or worse: just Minsky's "decrement").

Structured programming, canonical structures: Per the Church–Turing thesis, any algorithm can be computed by a model known to be Turing complete, and per Minsky's demonstrations, Turing completeness requires only four instruction types—conditional GOTO, unconditional GOTO, assignment, HALT. Kemeny and Kurtz observe that, while "undisciplined" use of unconditional GOTOs and conditional IF-THEN GOTOs can result in "spaghetti code", a programmer can write structured programs using only these instructions; on the other hand "it is also possible, and not too hard, to write badly structured programs in a structured language". Tausworthe augments the three Böhm-Jacopini canonical structures: SEQUENCE, IF-THEN-ELSE, and WHILE-DO, with two more: DO-WHILE and CASE. An additional benefit of a structured

eISSN 1303-5150

program is that it lends itself to proofs of correctness using mathematical induction.

Canonical flowchart symbols: The graphical aide called a flowchart offers a way to describe and document an algorithm (and a computer program corresponding to it). Like the program flow of a Minsky machine, a flowchart always starts at the top of a page and proceeds down. Its primary symbols are only four: the directed arrow showing program flow, the rectangle (SEQUENCE, GOTO), the diamond (IF-THEN-ELSE), and the dot (OR-tie). The Böhm–Jacopini canonical structures are made of these primitive shapes. Sub-structures can "nest" in rectangles, but only if a single exit occurs from the superstructure. The symbols and their use to build the canonical structures are shown in the diagram.

Euclid poses the problem thus: "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 zero. 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 zero, AND (ii) the subtraction must be "proper"; i.e., a test must guarantee that the smaller of the two numbers is subtracted from the larger (or the two can be equal so their subtraction yields zero).

Euclid's original proof adds a third requirement: the two lengths must not be prime to one another. Euclid stipulated this so that he could construct a reductio ad absurdum proof that the two numbers' common measure is in fact the greatest. While Nicomachus' algorithm is the same as Euclid's, when the numbers are prime to one another, it yields the number "1" for their common measure. So, to be precise, the following is really Nicomachus' algorithm.

```
// Euclid's algorithm for greatest common divisor
inteuclidAlgorithm (int A, int B) {
    A =abs(A);
    B =abs(B);
while (B !=0) {
while (A > B) {
          A = A-B;
       }
       B = B-A;
    }
return A;
}
```

Elegance (compactness) versus goodness (speed): With only six core instructions, "Elegant" is the clear winner, compared to "Inelegant" at thirteen instructions. However, "Inelegant" is faster (it arrives at HALT in fewer steps). Algorithm analysis indicates why this is the case: "Elegant" does two conditional tests in every subtraction loop, whereas "Inelegant" only does one. As the algorithm (usually) requires many loop-throughs, on average much time is wasted doing a "B = 0?" test that is needed only after the remainder is computed.

Can the algorithms be improved?: Once the programmer judges a program "fit" and "effective"—that is, it computes the function intended by its author—then the question becomes, can it be improved?

The compactness of "Inelegant" can be improved by the elimination of five steps. But Chaitin proved that compacting an algorithm cannot be automated by a generalized algorithm;rather, it can only be done heuristically; i.e., by exhaustive search (examples to be found at Busy beaver), trial and error, cleverness, insight, application of inductive reasoning, etc. Observe that steps 4, 5 and 6 are repeated in steps 11, 12 and 13. Comparison with "Elegant" provides a hint that these steps, together with steps 2 and 3, can be

eISSN 1303-5150

eliminated. This reduces the number of core instructions from thirteen to eight, which makes it "more elegant" than "Elegant", at nine steps.

The speed of "Elegant" can be improved by moving the "B=0?" test outside of the two subtraction loops. This change calls for the addition of three instructions (B = 0?, A = 0?, GOTO). Now "Elegant" computes the example-numbers faster; whether this is always the case for any given A, B, and R, S would require a detailed analysis.

It is frequently important to know how much of a particular resource (such as time or storage) is theoretically required for a given algorithm. Methods have been developed for the analysis of algorithms to obtain such quantitative answers (estimates); for example, an algorithm which adds up the elements of a list of n numbers would have a time requirement of O(n), using big O notation. At all times the algorithm only needs to remember two values: the sum of all the elements so far, and its current position in the input list. Therefore, it is said to have a space requirement of O(1), if the space required to store the input numbers is not counted, or O(n) if it is counted.

Different algorithms may complete the same task with a different set of instructions in less or more time, space, or 'effort' than others. For example, a binary search algorithm (with cost O(log n)) outperforms a sequential search (cost O(n)) when used for table lookups on sorted lists or arrays.

Formal versus empirical

Main articles: Empirical algorithmics, Profiling (computer programming), and Program optimization

The analysis, and study of algorithms is a discipline of computer science, and is often practiced abstractly without the use of a specific programming language or implementation. In this sense, algorithm analysis resembles other mathematical disciplines in that it focuses on the underlying properties of the algorithm and not on the specifics of any particular implementation. Usually pseudocode is used for analysis as it is the simplest and most general representation. However, ultimately, most algorithms are usually implemented on particular hardware/software platforms and their algorithmic efficiency is eventually put to the test using real code. For the solution of a "one off" problem, the efficiency of a particular algorithm may not have significant consequences (unless n is extremely large) but for algorithms designed for fast interactive, commercial or long life scientific usage it may be critical. Scaling from small n to large n frequently exposes inefficient algorithms that are otherwise benign.

Empirical testing is useful because it may uncover unexpected interactions that affect performance. Benchmarks may be used to compare before/after potential improvements to an algorithm after program optimization. Empirical tests cannot replace formal analysis, though, and are not trivial to perform in a fair manner.

REFERENCES

1.Ашурова З. Р., Жураева Н. Ю., Жураева У. Ю. Функция Карлемана для полигармонических функций определенных в некоторых областей лежащих в некоторых четном n-мерном евклидовом пространстве //Операторные алгебры и смежные проблемы. – 2012. – С. 100-101.

2.Raximovna A. Z., Yunusovna J. N., Yunusalievna J. U. Task Cauchy and Carleman Function //Texas Journal of Multidisciplinary Studies. – 2021. – Т. 1. – №. 1. – С. 228-231.

3.Raximovna A. Z., Yunusovna J. N. Some Estimates For TheCarleman Function //The American Journal of Applied sciences. – 2021. – Т. 3. – №. 06. – С. 77-81.

4.Ашурова З. Р. и др. ОЦЕНИВАЕТСЯ ФУНКЦИЯ КАРЛЕМАНА ДЛЯ ПОЛИГАРМОНИЧЕСКИХ ФУНКЦИЙ ВТОРОГО ПОРЯДКА, ОПРЕДЕЛЕННЫХ В ОБЛАСТИ ТРЕХМЕРНОГО ПРОСТРАНСТВА //ScienceandEducation. – 2021. – Т. 2. – №. 2. – С. 17-21.

5.Yunusovna J. N., Juraeva A. Z. R., Yunusalievna U. Growing polyharmonic functions and the Cauchy problem //Journal of Critical Reviews. – 2020. – Т. 7. – №. 7. – С. 371-378.

eISSN 1303-5150