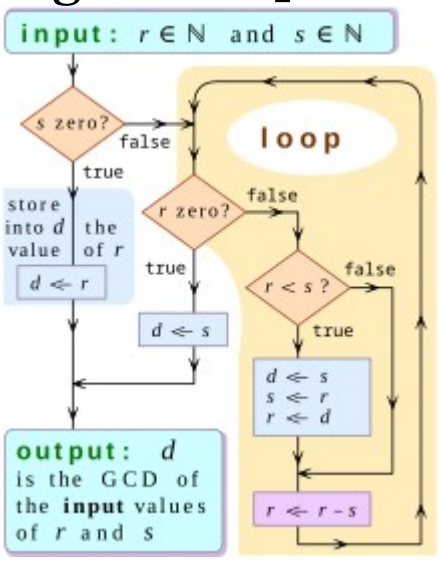
Algorithm [UNIT-II]



Flowchart of using successive subtractions to find the greatest common divisor of number *r* and *s*

In mathematics and computer science, an **algorithm** (/ˈælgərɪðəm/ ⓘ) is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation.[1] Algorithms are used as specifications for performing calculations and data processing. More advanced algorithms can use conditionals to divert the code execution through various routes (referred to as automated decision-making) and deduce valid inferences (referred to as automated reasoning).

In contrast, a heuristic is an approach to solving problems that do not have well-defined correct or optimal results.[2] For example, although social media recommender systems are commonly called "algorithms", they actually rely on heuristics as there is no truly "correct" recommendation.

As an effective method, an algorithm can be expressed within a finite amount of space and time[3] and in a well-defined formal language[4] for calculating a function.[5] Starting from an initial state and initial input (perhaps empty),[6] the instructions describe a computation that, when executed, proceeds through a finite[7] number of well-defined successive states, eventually producing "output"[8] 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.[9]

Etymology

, Persian scientist and polymath Muḥammad ibn Mūsā al-Khwārizmī wrote *kitāb al-ḥisāb al-hindī* ("Book of Indian computation") and *kitab al-jam' wa'l-tafriq al-ḥisāb al-hindī* ("Addition and subtraction in Indian arithmetic").[1] In the early 12th century, Latin translations of said al-Khwarizmi texts involving the Hindu–Arabic numeral system and arithmetic appeared, for example *Liber Alghoarismi de practica arismetrice*, attributed to John of Seville, and *Liber Algorismi de numero Indorum*, attributed

to Adelard of Bath.[10] Hereby, *alghoarismi* or *algorismi* is the Latinization of Al-Khwarizmi's name; the text starts with the phrase *Dixit Algorismi*, or "Thus spoke Al-Khwarizmi".[2] Around 1230, the English word *algorism* is attested and then by Chaucer in 1391, English adopted the French term.[3][4][*clarification needed*] In the 15th century, under the influence of the Greek word ἀριθμός (*arithmos*, "number"; *cf.* "arithmetic"), the Latin word was altered to *algorithmus*.[*citation needed*]

Definition

For a detailed presentation of the various points of view on the definition of "algorithm", see Algorithm characterizations.

One informal definition is "a set of rules that precisely defines a sequence of operations",[11][*need quotation to verify*] which would include all computer programs (including programs that do not perform numeric calculations), and any prescribed bureaucratic procedure[12] or cook-book recipe.[13] In general, a program is an algorithm only if it stops eventually[14]—even though infinite loops may sometimes prove desirable. Boolos, Jeffrey & 1974, 1999 define an algorithm to be an explicit set of instructions for determining an output, that can be followed by a computing machine or a human who could only carry out specific elementary operations on symbols.[15]

Most algorithms are intended to be implemented as computer programs. However, algorithms are also implemented by other means, such as in a biological neural network (for example, the human brain performing arithmetic or an insect looking for food), in an electrical circuit, or a mechanical device.

History

[Ancient algorithms

Step-by-step procedures for solving mathematical problems have been recorded since antiquity. This includes in Babylonian mathematics (around 2500 BC),[16] Egyptian mathematics (around 1550 BC),[16] Indian mathematics (around 800 BC and later),[17][18] the Ifa Oracle (around 500 BC),[19] Greek mathematics (around 240 BC),[20] and Arabic mathematics (around 800 AD).[21]

The earliest evidence of algorithms is found in ancient Mesopotamian mathematics. A Sumerian clay tablet found in Shuruppak near Baghdad and dated to c. 2500 BC describes the earliest division algorithm.[16] During the Hammurabi dynasty c. 1800 – c. 1600 BC, Babylonian clay tablets described algorithms for computing formulas.[22] Algorithms were also used in Babylonian astronomy. Babylonian clay tablets describe and employ algorithmic procedures to compute the time and place of significant astronomical events.[23]

Algorithms for arithmetic are also found in ancient Egyptian mathematics, dating back to the Rhind Mathematical Papyrus c. 1550 BC.[16] Algorithms were later used in ancient Hellenistic mathematics. Two examples are the Sieve of Eratosthenes, which was described in the *Introduction to Arithmetic* by Nicomachus,[24][20]:Ch 9.2 and the Euclidean algorithm, which was first described in *Euclid's Elements* (c. 300 BC).[20]:Ch

9.1Examples of ancient Indian mathematics included the Shulba Sutras, the Kerala School, and the Brāhmasphutasiddhānta.[17]

The first cryptographic algorithm for deciphering encrypted code was developed by Al-Kindi, a 9th-century Arab mathematician, in *A Manuscript On Deciphering Cryptographic Messages*. He gave the first description of cryptanalysis by frequency analysis, the earliest codebreaking algorithm.[21]

Computers

Weight-driven clocks

Bolter credits the invention of the weight-driven clock as "the key invention [of Europe in the Middle Ages]," specifically the verge escapement mechanism[25] producing the tick and tock of a mechanical clock. "The accurate automatic machine"[26] led immediately to "mechanical automata" in the 13th century and "computational machines"—the difference and analytical engines of Charles Babbage and Ada Lovelace in the mid-19th century.[27] Lovelace designed the first algorithm intended for processing on a computer, Babbage's analytical engine, which is the first device considered a real Turing-complete computer instead of just a calculator. Although a full implementation of Babbage's second device was not realized for decades after her lifetime, Lovelace has been called "history's first programmer".

Electromechanical relay

Bell and Newell (1971) write that the Jacquard loom, a precursor to Hollerith cards (punch cards), and "telephone switching technologies" led to the development of the first computers.[28] By the mid-19th century, the telegraph, the precursor of the telephone, was in use throughout the world. By the late 19th century, the ticker tape (c. 1870s) was in use, as were Hollerith cards (c. 1890). Then came the teleprinter (c. 1910) with its punched-paper use of Baudot code on tape.

Telephone-switching networks of electromechanical relays were invented in 1835. These led to the invention of the digital adding device by George Stibitz in 1937. While working in Bell Laboratories, he observed the "burdensome" use of mechanical calculators with gears. "He went home one evening in 1937 intending to test his idea... When the tinkering was over, Stibitz had constructed a binary adding device".[29][30]

Formalization


Ada Lovelace's diagram from "Note G", the first published computer algorithm

In 1928, a partial formalization of the modern concept of algorithms began with attempts to solve the *Entscheidungsproblem* (decision problem) posed by David Hilbert. Later formalizations were framed as attempts to define "effective calculability"[31] or "effective method".[32] Those formalizations included the Gödel–Herbrand–Kleene recursive functions of 1930, 1934 and 1935, Alonzo Church's lambda calculus of 1936, Emil Post's Formulation 1 of 1936, and Alan Turing's Turing machines of 1936–37 and 1939.

Representations

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, drakon-charts, 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, drakon-charts, and control tables are structured expressions of algorithms that avoid common ambiguities of natural language. Programming languages are primarily for expressing algorithms in a computer-executable form, but are also used to define or document algorithms.

Turing machines

There are many possible representations and Turing machine programs can be expressed as a sequence of machine tables (see finite-state machine, state-transition table, and control table for more), as flowcharts and drakon-charts (see state diagram for more), as a form of rudimentary machine code or assembly code called "sets of quadruples", and more. Algorithm representations can also be classified into three accepted levels of Turing machine description: high-level description, implementation description, and formal description.[33] A high-level description describes qualities of the algorithm itself, ignoring how it is implemented on the Turing machine.[33] An implementation description describes the general manner in which the machine moves its head and stores data in order to carry out the algorithm, but does not give exact states.[33] In the most detail, a formal description gives the exact state table and list of transitions of the Turing machine.[33]

Flowchart representation

The graphical aid called a flowchart offers a way to describe and document an algorithm (and a computer program corresponding to it). It has four primary symbols: arrows showing program flow, rectangles (SEQUENCE, GOTO), diamonds (IF-THEN-ELSE), and dots (OR-tie). Sub-structures can "nest" in rectangles, but only if a single exit occurs from the superstructure.

Algorithmic analysis

It is often important to know how much time, storage, or other cost an algorithm may require. Methods have been developed for the analysis of algorithms to obtain such quantitative answers (estimates); for example, an algorithm that adds up the elements

of a list of n numbers would have a time requirement of , using big O notation. The

algorithm only needs to remember two values: the sum of all the elements so far, and its current position in the input list. If the space required to store the input numbers is not

counted, it has a space requirement of , otherwise is required.

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) outperforms a sequential search (cost) when used for table lookups on sorted lists or arrays.

Formal versus empirical

The analysis, and study of algorithms is a discipline of computer science. Algorithms are often studied abstractly, without referencing any specific programming language or implementation. Algorithm analysis resembles other mathematical disciplines as it focuses on the algorithm's properties, not implementation. Pseudocode is typical for analysis as it is a simple and general representation. Most algorithms are implemented on particular hardware/software platforms and their algorithmic efficiency is tested using real code. The efficiency of a particular algorithm may be insignificant for many "one-off" problems but it may be critical for algorithms designed for fast interactive, commercial or long life scientific usage. Scaling from small n to large n frequently exposes inefficient algorithms that are otherwise benign.

Empirical testing is useful for uncovering 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 non-trivial to perform fairly.[34]

Execution efficiency

: Algorithmic efficiency

To illustrate the potential improvements possible even in well-established algorithms, a recent significant innovation, relating to FFT algorithms (used heavily in the field of image processing), can decrease processing time up to 1,000 times for applications like medical imaging.[35] In general, speed improvements depend on special properties of the problem, which are very common in practical applications.[36] Speedups of this magnitude enable computing devices that make extensive use of image processing (like digital cameras and medical equipment) to consume less power.

Design

: Algorithm § By design paradigm

Algorithm design is a method or mathematical process for problem-solving and engineering algorithms. The design of algorithms is part of many solution theories, such as divide-and-conquer or dynamic programming within operation research. Techniques for designing and implementing algorithm designs are also called algorithm design patterns,[37] with examples including the template method pattern and the decorator

pattern. One of the most important aspects of algorithm design is resource (run-time, memory usage) efficiency; the big O notation is used to describe e.g., an algorithm's run-time growth as the size of its input increases.[38]

Structured programming

Per the Church–Turing thesis, any algorithm can be computed by any Turing complete model. Turing completeness only requires four instruction types—conditional GOTO, unconditional GOTO, assignment, HALT. However, 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".[39] Tausworthe augments the three Böhm-Jacopini canonical structures:[40] SEQUENCE, IF-THEN-ELSE, and WHILE-DO, with two more: DO-WHILE and CASE.[41] An additional benefit of a structured program is that it lends itself to proofs of correctness using mathematical induction.[42]

Legal status

By themselves, algorithms are not usually patentable. In the United States, a claim consisting solely of simple manipulations of abstract concepts, numbers, or signals does not constitute "processes" (USPTO 2006), so algorithms are not patentable (as in *Gottschalk v. Benson*). However practical applications of algorithms are sometimes patentable. For example, in *Diamond v. Diehr*, the application of a simple feedback algorithm to aid in the curing of synthetic rubber was deemed patentable. The patenting of software is controversial,[43] and there are criticized patents involving algorithms, especially data compression algorithms, such as Unisys's LZW patent. Additionally, some cryptographic algorithms have export restrictions (see export of cryptography).

Classification

By implementation

Recursion

A recursive algorithm invokes itself repeatedly until meeting a termination condition, and is a common functional programming method. Iterative algorithms use repetitions such as loops or data structures like stacks to solve problems. Problems may be suited for one implementation or the other.Towers of Hanoi is a puzzle commonly solved using recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.

Serial, parallel or distributed

Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time on serial computers. Serial algorithms are designed for these environments, unlike parallel or distributed algorithms. Parallel algorithms take advantage of computer architectures where multiple processors can work on a problem at the same time. Distributed algorithms use

multiple machines connected via a computer network. Parallel and distributed algorithms divide the problem into subproblems and collect the results back together. Resource consumption in these algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Some sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable, but some problems have no parallel algorithms and are called inherently serial problems.

Deterministic or non-deterministic

Deterministic algorithms solve the problem with exact decision at every step; whereas non-deterministic algorithms solve problems via guessing. Guesses are typically made more accurate through the use of heuristics.

Exact or approximate

While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Such algorithms have practical value for many hard problems. For example, the Knapsack problem, where there is a set of items and the goal is to pack the knapsack to get the maximum total value. Each item has some weight and some value. The total weight that can be carried is no more than some fixed number X. So, the solution must consider weights of items as well as their value.[44]

Quantum algorithm

Quantum algorithms run on a realistic model of quantum computation. The term is usually used for those algorithms which seem inherently quantum or use some essential feature of Quantum computing such as quantum superposition or quantum entanglement.

By design paradigm

[edit]

Another way of classifying algorithms is by their design methodology or paradigm. Some common paradigms are:

Brute-force or exhaustive search

Brute force is a problem-solving method of systematically trying every possible option until the optimal solution is found. This approach can be very time-consuming, testing every possible combination of variables. It is often used when other methods are unavailable or too complex. Brute force can solve a variety of problems, including finding the shortest path between two points and cracking passwords.

Divide and conquer

A divide-and-conquer algorithm repeatedly reduces a problem to one or more smaller instances of itself (usually recursively) until the instances are small enough to solve easily. Merge sorting is an example of divide and conquer, where an unordered list can be divided into segments containing one item and sorting of entire list can be obtained by merging the segments. A simpler variant of divide and conquer is called a *decrease-and-conquer algorithm*, which solves one smaller instance of itself, and uses the solution to solve the bigger problem.

Divide and conquer divides the problem into multiple subproblems and so the conquer stage is more complex than decrease and conquer algorithms.[citation needed] An example of a decrease and conquer algorithm is the binary search algorithm.

Search and enumeration

Many problems (such as playing chess) can be modelled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration, and backtracking.

Randomized algorithm

Such algorithms make some choices randomly (or pseudo-randomly). They find approximate solutions when finding exact solutions may be impractical (see heuristic method below). For some problems the fastest approximations must involve some randomness.[45] Whether randomized algorithms with polynomial time complexity can be the fastest algorithm for some problems is an open question known as the P versus NP problem. There are two large classes of such algorithms:

1. Monte Carlo algorithms return a correct answer with high probability. E.g. RP is the subclass of these that run in polynomial time.
2. Las Vegas algorithms always return the correct answer, but their running time is only probabilistically bound, e.g. ZPP.

Reduction of complexity

This technique transforms difficult problems into better-known problems solvable with (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. For example, one selection algorithm finds the median of an unsorted list by first sorting the list (the expensive portion), then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as *transform and conquer*.

Back tracking

In this approach, multiple solutions are built incrementally and abandoned when it is determined that they cannot lead to a valid full solution.

Optimization problems

For optimization problems there is a more specific classification of algorithms; an algorithm for such problems may fall into one or more of the general categories described above as well as into one of the following:

Linear programming

When searching for optimal solutions to a linear function bound by linear equality and inequality constraints, the constraints can be used directly to produce

optimal solutions. There are algorithms that can solve any problem in this category, such as the popular simplex algorithm.[46] Problems that can be solved with linear programming include the maximum flow problem for directed graphs. If a problem also requires that any of the unknowns be integers, then it is classified in integer programming. A linear programming algorithm can solve such a problem if it can be proved that all restrictions for integer values are superficial, i.e., the solutions satisfy these restrictions anyway. In the general case, a specialized algorithm or an algorithm that finds approximate solutions is used, depending on the difficulty of the problem.

Dynamic programming

When a problem shows optimal substructures—meaning the optimal solution can be constructed from optimal solutions to subproblems—and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions. For example, Floyd–Warshall algorithm, the shortest path between a start and goal vertex in a weighted graph can be found using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. Unlike divide and conquer, dynamic programming subproblems often overlap. The difference between dynamic programming and simple recursion is the caching or memoization of recursive calls. When subproblems are independent and do not repeat, memoization does not help; hence dynamic programming is not applicable to all complex problems. Using memoization dynamic programming reduces the complexity of many problems from exponential to polynomial.

The greedy method

Greedy algorithms, similarly to a dynamic programming, work by examining substructures, in this case not of the problem but of a given solution. Such algorithms start with some solution and improve it by making small modifications. For some problems they always find the optimal solution but for others they may stop at local optima. The most popular use of greedy algorithms is finding minimal spanning trees of graphs without negative cycles. Huffman Tree, Kruskal, Prim, Sollin are greedy algorithms that can solve this optimization problem.

The heuristic method

In optimization problems, heuristic algorithms find solutions close to the optimal solution when finding the optimal solution is impractical. These algorithms get closer and closer to the optimal solution as they progress. In principle, if run for an infinite amount of time, they will find the optimal solution. They can ideally find a solution very close to the optimal solution in a relatively short time. These algorithms include local search, tabu search, simulated annealing, and genetic algorithms. Some, like simulated annealing, are non-deterministic algorithms while others, like tabu search, are deterministic. When a bound on the error of the non-optimal solution is known, the algorithm is further categorized as an approximation algorithm.

Examples

[edit]

Further information: List of algorithms

One of the simplest algorithms finds the largest number in a list of numbers of random order. Finding the solution requires looking at every number in the list. From this follows a simple algorithm, which can be described in plain English as:

High-level description:

1. If a set of numbers is empty, then there is no highest number.
2. Assume the first number in the set is the largest.
3. For each remaining number in the set: if this number is greater than the current largest, it becomes the new largest.
4. When there are no unchecked numbers left in the set, consider the current largest number to be the largest in the set.

(Quasi-)formal description: Written in prose but much closer to the high-level language of a computer program, the following is the more formal coding of the algorithm in pseudocode or pidgin code:

```
Algorithm LargestNumber
Input: A list of numbers L.
Output: The largest number in the list L.
if L.size = 0 return null
largest ← L[0]
for each item in L, do
    if item > largest, then
        largest ← item
return largest
```

- "←" denotes assignment. For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the following value.