# Algorithms

An algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

From the data structure point of view, the following are some important categories of algorithms −

* Sort: Algorithm developed for sorting the items in a certain order.
* Search: Algorithm developed for searching the items inside a data structure.
* Delete: Algorithm developed for deleting the existing element from the data structure.
* Insert: Algorithm developed for inserting an item inside a data structure.
* Update: Algorithm developed for updating the existing element inside a data structure

## Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

* Unambiguous − The algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* Input − An algorithm should have 0 or more well-defined inputs.
* Output − An algorithm should have 1 or more well-defined outputs and should match the desired output.
* Finiteness − Algorithms must terminate after a finite number of steps.
* Feasibility − Should be feasible with the available resources.
* Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

## How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource-dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

### Example

Now we will look at an example of an algorithm in programming.

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START

Step 2 − declare three integers a, b & c

Step 3 − define values of a & b

Step 4 − add values of a & b

Step 5 − store output of step 4 to c

Step 6 − print c

Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD

Step 2 − get values of a & b

Step 3 − c ← a + b

Step 4 − display c

Step 5 − STOP

In the design and analysis of algorithms, usually, the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing. Writing step numbers is optional.

## Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

* Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
* Space Factor − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

### Space Complexity

Space complexity: An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

The space required by an algorithm is equal to the sum of the following two components −

* A fixed part is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

### Time Complexity

Time complexity: The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, the big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution.

For example, the addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.