DATA STRUCTURES - ALGORITHMS BASICS

Algorithm is a step by step procedure, which defines a set of instructions to be executed in 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.

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

- **Search** − Algorithm to search an item in a datastructure.
- **Sort** − Algorithm to sort items in certain order
- **Insert** − Algorithm to insert item in a datastructure
- **Update** − Algorithm to update an existing item in a data structure
- **Delete** − Algorithm to delete an existing item from a data structure

Characteristics of an Algorithm

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

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps *orphases*, and their input/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 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

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

Problem − Design an algorithm to add two numbers and display 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
```

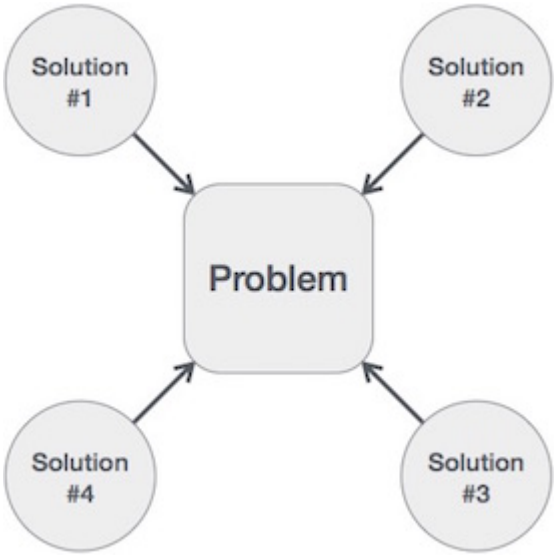
Algorithms tell the programmers how to code the program. Alternatively the algorithm can be written as −

```
step 1 − START ADD
step 2 − get values of a & b
step 3 − c ← a + b
step 4 − display c
step 5 − STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy of 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.

We design an algorithm to get solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. Next step is to analyze those proposed solution algorithms and implement the best suitable.

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below −

- ***A priori* analysis** − This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
- ***A posterior* analysis** − This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn here **a priori** algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

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** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** − The 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 storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant 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 of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.

For example, 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 addition of two bits. Here, we observe that **T(n)** grows linearly as input size increases.