**Definition of Data Structure**

A data structure is a data organization, management, and storage format that enables efficient access and modification.

It is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data Structure form the basis for Abstract Data Types (ADT) An abstract data type is defined by its behavior (semantics) from the point of view of a *user*, of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services.

**Basic Terms**

**Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.

**Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

**Data** − Data are values or set of values.
**Data Item** − Data item refers to single unit of values.
**Group Items** − Data item that are divided into sub items are called as Group Items.
**Elementary Items** − Data item that cannot be divided are called as Elementary Items.
**Attribute and Entity** − An entity is that which contains certain attributes or properties which may be assigned values.
**Entity Set** − Entities of similar attributes form an entity set.
**Field** − Field is a single elementary unit of information representing an attribute of an entity.
**Record** − Record is a collection of field values of a given entity.
**File** − File is a collection of records of the entities in a given entity set.

**Characteristics of a Data Structure**

**Correctness** − Data Structure implementation should implement its interface correctly.
**Time Complexity** − Running time or execution time of operations of data structure must be as small as possible.
**Space Complexity** − Memory usage of a data structure operation should be as little as possible.

**Need for Data Structure**
As applications are getting complex and data rich, there are three common problems applications face now-a-days.

**Data Search** − Consider an inventory of 1 million($10^6$) items of a store. If application is to search an item. It has to search item in 1 million($10^6$) items every time slowing down the search. As data grows, search will become slower.
**Processor speed** − Processor speed although being very high, falls limited if data grows to billion records.
**Multiple requests** − As thousands of users can search data simultaneously on a web server,even very fast server fails while searching the data.

To solve above problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be search and required data can be searched almost instantly.

**Execution Time Cases**
There are three cases which are usual used to compare various data structure's execution time in relative manner.
**Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f$(n) then this operation will not take time more than $f$(n) time where $f$(n) represents function of n.
**Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes $f$(n) time in execution then m operations will take m$f$(n) time.
**Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes $f$(n) time in execution then actual operation may take time as random number which would be maximum as $f$(n).

**Definition of Algorithm**

An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of specific problems or to perform a computation. Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.

Some common Algorithms
**Search** − Algorithm to search an item in a data structure.
**Sort** − Algorithm to sort items in certain order
**Insert** − Algorithm to insert item in a data structure
**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 (or phases), 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.

**Writing 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

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

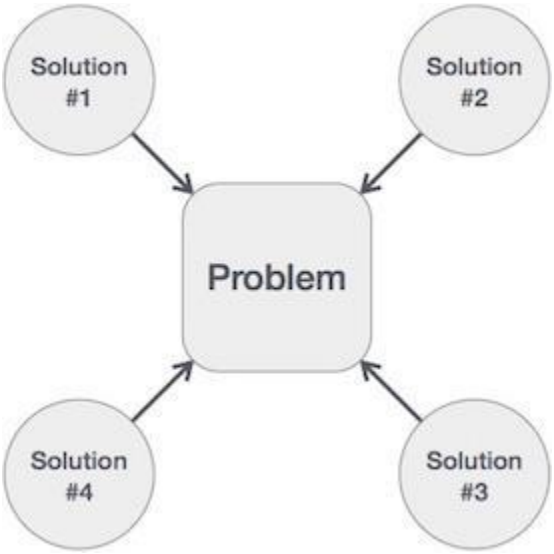
**step 1** − START
**step 2** − declare three integers **a**, **b** & **c**
**step 3** − define values of **a** & **b**
**step 4** − add values of **a** & **b**
**step 5** − store output of step 4 to **c**
**step 6** − print **c**
**step 7** – STOP

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.