**Link : http://www3.cs.stonybrook.edu/~algorith/**

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

**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.

## Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

**Data Search** − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) 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 the data grows to billion records.

**Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

## Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a 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 ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(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 ƒ(n) time in execution, then m operations will take mƒ(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 ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).

## Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

**A Priori Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

**A Posterior Analysis** − This is an 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 about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number 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** − 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 of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The 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 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

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

# Data Structures - Greedy Algorithms

## Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

**1** − Select one ₹ 10 coin, the remaining count is 8

**2** − Then select one ₹ 5 coin, the remaining count is 3

**3** − Then select one ₹ 2 coin, the remaining count is 1

**4** − And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

### Examples

Most networking algorithms use the greedy approach. Here is a list of few of them −

* Travelling Salesman Problem
* Prim's Minimal Spanning Tree Algorithm
* Kruskal's Minimal Spanning Tree Algorithm
* Dijkstra's Minimal Spanning Tree Algorithm
* Graph - Map Coloring
* Graph - Vertex Cover
* Knapsack Problem
* Job Scheduling Problem

# Data Structures - Divide and Conquer

### Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

* Merge Sort
* Quick Sort
* Binary Search