Week 1: Introduction to Data Structures and Basic Algorithms

- Introduction to data structures and their importance in problem-solving
- Overview of algorithm analysis and complexity
- Arrays and their operations
- Basic searching and sorting algorithms
- In-class practical: Implementing and analyzing basic algorithms

Assignment 1: Implement and analyze different searching and sorting algorithms using arrays.

Introduction to Data Structures and their Importance in Problem-Solving

Data Structure and Algorithms Tutorial

PDF Version Quick Guide Resources Job Search Discussion

Data Structures are the programmatic way of storing data so that data can be used efficiently. Almost every enterprise application uses various types of data structures in one or the other way. This tutorial will give you a great understanding on Data Structures needed to understand the complexity of enterprise level applications and need of algorithms, and data structures.

Why to Learn Data Structure and Algorithms?

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(10^6) items of a store. If the application is to search an item, it has to search an 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 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.

To solve the above-mentioned 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 searched, and the required data can be searched almost instantly.

Applications of Data Structure and Algorithms

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.

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

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a 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.

The following computer problems can be solved using Data Structures −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Audience

This tutorial is designed for Computer Science graduates as well as Software Professionals who are willing to learn data structures and algorithm programming in simple and easy steps.

After completing this tutorial you will be at intermediate level of expertise from where you can take yourself to higher level of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of C programming language, text editor, and execution of programs, etc.

Data Structures & Algorithms - Overview

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.

Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the 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 that applications face now-a-days.

- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an 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 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.

To solve the above-mentioned 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 searched, and the required data can be searched almost instantly.

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).

Basic Terminology

- **Data** − Data are values or set of values.
- **Data Item** − Data item refers to single unit of values.
- **Group Items** − Data items that are divided into sub items are called as Group Items.
- **Elementary Items** − Data items 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.

Data Structures - Environment Setup

Local Environment Setup

If you are still willing to set up your environment for C programming language, you need the following two tools available on your computer, (a) Text Editor and (b) The C Compiler.

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

The name and the version of the text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on Windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for C programs are typically named with the extension "**.c**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, compile it, and finally execute it.

The C Compiler

The source code written in the source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program as per the given instructions.

This C programming language compiler will be used to compile your source code into a final executable program. We assume you have the basic knowledge about a programming language compiler.

Most frequently used and free available compiler is GNU C/C++ compiler. Otherwise, you can have compilers either from HP or Solaris if you have respective Operating Systems (OS).

The following section guides you on how to install GNU C/C++ compiler on various OS. We are mentioning C/C++ together because GNU GCC compiler works for both C and C++ programming languages.

Installation on UNIX/Linux

If you are using **Linux or UNIX**, then check whether GCC is installed on your system by entering the following command from the command line −

```
$ gcc -v
```

If you have GNU compiler installed on your machine, then it should print a message such as the following −

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix = /usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at https://gcc.gnu.org/install/

This tutorial has been written based on Linux and all the given examples have been compiled on Cent OS flavor of Linux system.

Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at developer.apple.com/technologies/tools/

Installation on Windows

To install GCC on Windows, you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing MinWG, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable, so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

Data Structures - Algorithms Basics

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.

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

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a 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 following characteristics −

- **Unambiguous** − 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

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
```

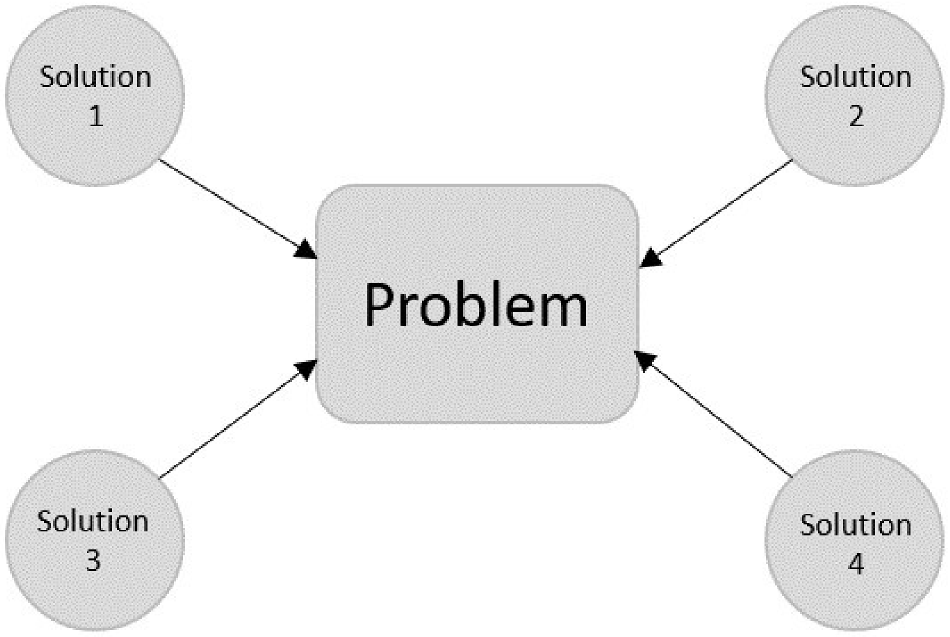
Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

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

In 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.

We design an algorithm to get a 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. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

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

Data Structures - Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

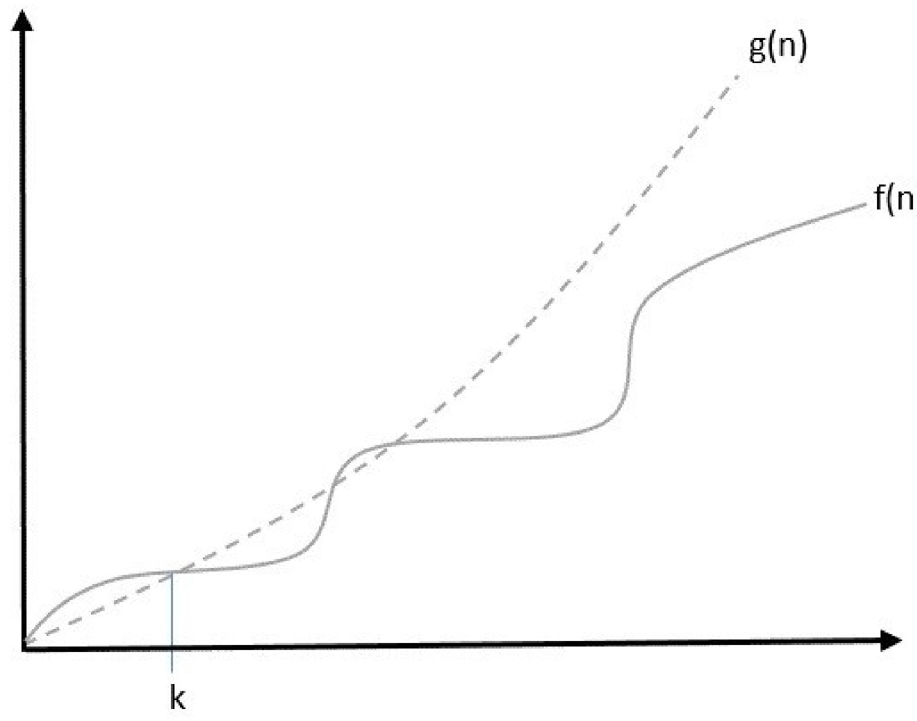
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- Ο − Big Oh Notation
- Ω − Big Omega Notation
- Θ − Theta Notation
- o − Little Oh Notation
- ω − Little Omega Notation

Big Oh Notation, Ο

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the **worst case time complexity** or the longest amount of time an algorithm can possibly take to complete.



For example, for a function ***f*(n)**

```
O(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }
```

Example

Let us consider a given function, ***f*(*n*) = 4.n^3+10.n^2+5.*n*+1.**

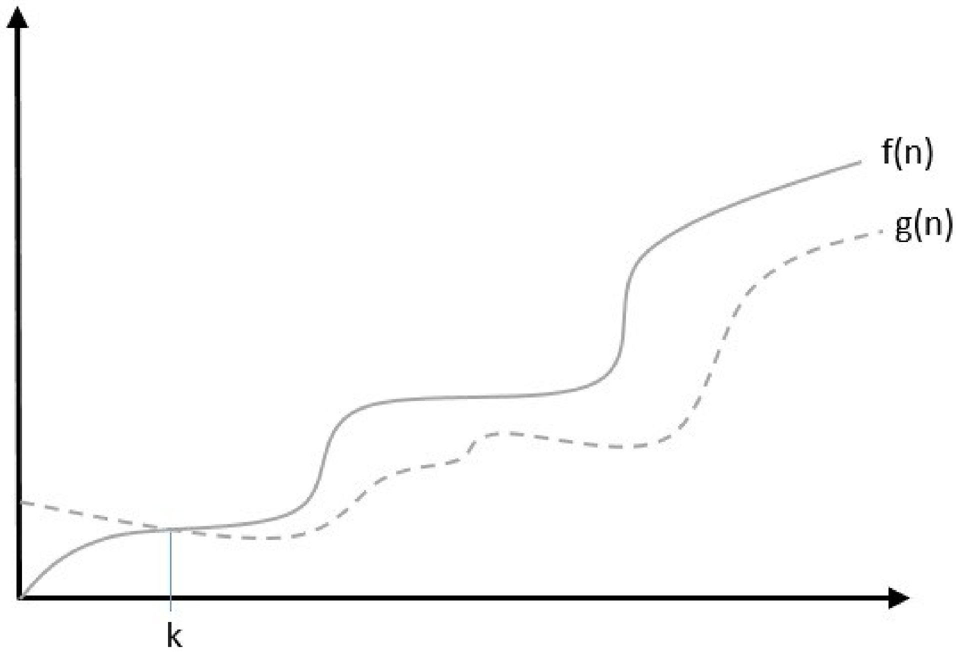
Considering ***g(n)* = n^3**

***f*(*n*) ≥ 5.*g*(*n*)** for all the values of ***n* > 2**.

Hence, the complexity of ***f*(n)** can be represented as ***O* (*g* (*n*)) ,i.e. *O* (n^3)**.

Big Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the **best case time complexity** or the best amount of time an algorithm can possibly take to complete.



For example, for a function ***f*(*n*)**

```
Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n >
n0. }
```

Example

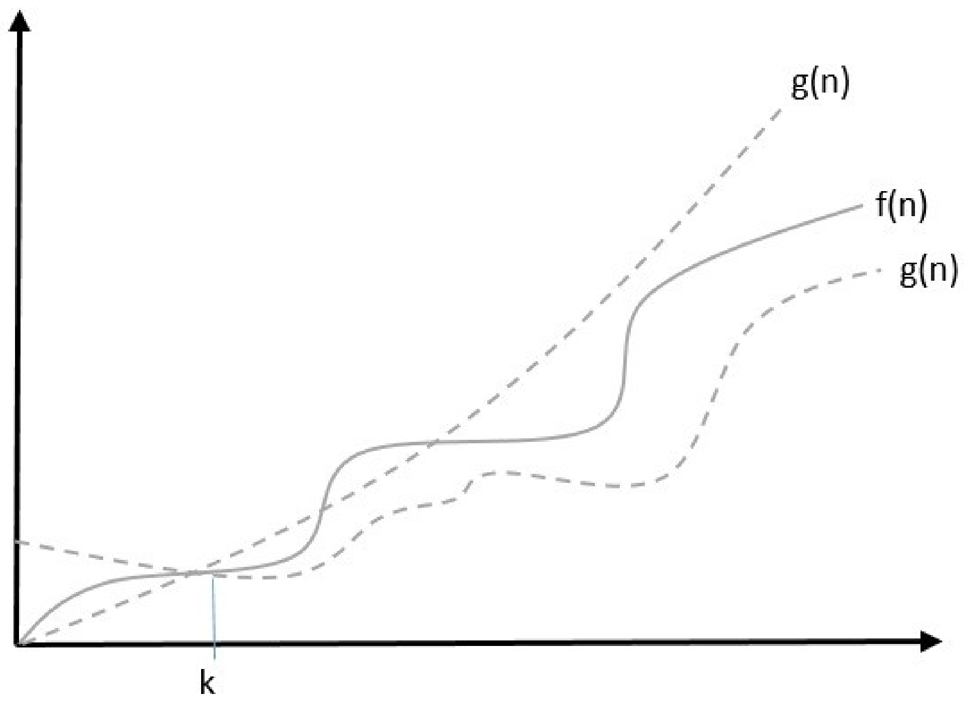
Let us consider a given function, ***f*(*n*) = 4.*n*^{3}+10.*n*^{2}+5.*n*+1**

Considering ***g*(*n*) = *n*^{3}** , ***f*(*n*) ≥ 4.*g*(*n*)** for all the values of ***n* > 0**.

Hence, the complexity of *f*(*n*) can be represented as ***Ω* (*g* (*n*)) ,i.e. *Ω* (*n*^{3})**.

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be *almost* accurately used to describe the average case, other notations could be used as well. It is represented as follows −



θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }

Example

Let us consider a given function, ***f*(*n*) = 4.*n*^{3}+10.*n*^{2}+5.*n*+1**

Considering ***g*(*n*) = *n*^{3}** , **4.*g*(*n*)≤ *f*(*n*)≤ 5.*g*(*n*)** for all the values of ***n***.

Hence, the complexity of ***f*(*n*)** can be represented as ***Θ* (*g* (*n*)) ,i.e. *Θ* (*n*^{3})**.

Little Oh (o) and Little Omega (ω) Notations

The Little Oh and Little Omega notations also represent the best and worst case complexities but they are not asymptotically tight in contrast to the Big Oh and Big Omega Notations. Therefore, the most commonly used notations to represent time complexities are Big Oh and Big Omega Notations only.

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	−	O(1)
logarithmic	−	O(log n)
linear	−	O(n)

n log n	−	O(n log n)
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

Data Structures - Greedy Algorithms

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

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

There are lots of similar problems that uses the greedy approach to find an optimum solution.

Data Structures - Divide and Conquer

To understand the divide and conquer design strategy of algorithms, let us use a simple real world example. Consider an instance where we need to brush a type C curly hair and remove all the knots from it. To do that, the first step is to section the hair in smaller strands to make the combing easier than combing the hair altogether. The same technique is applied on algorithms.

Divide and conquer approach breaks down a problem into multiple sub-problems recursively until it cannot be divided further. These sub-problems are solved first and the solutions are merged together to form the final solution.

The common procedure for the divide and conquer design technique is as follows −

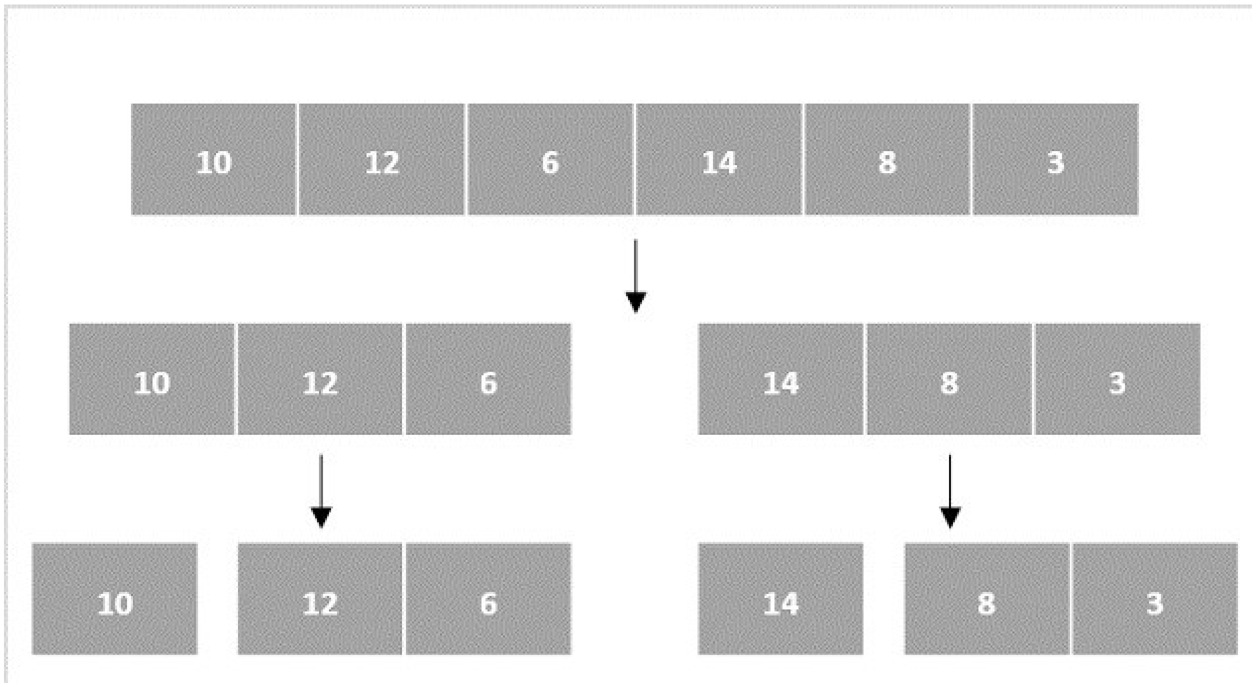
- **Divide** − We divide the original problem into multiple sub-problems until they cannot be divided further.
- **Conquer** − Then these subproblems are solved separately with the help of recursion
- **Combine** − Once solved, all the subproblems are merged/combined together to form the final solution of the original problem.

There are several ways to give input to the divide and conquer algorithm design pattern. Two major data structures used are − **arrays** and **linked lists**. Their usage is explained as

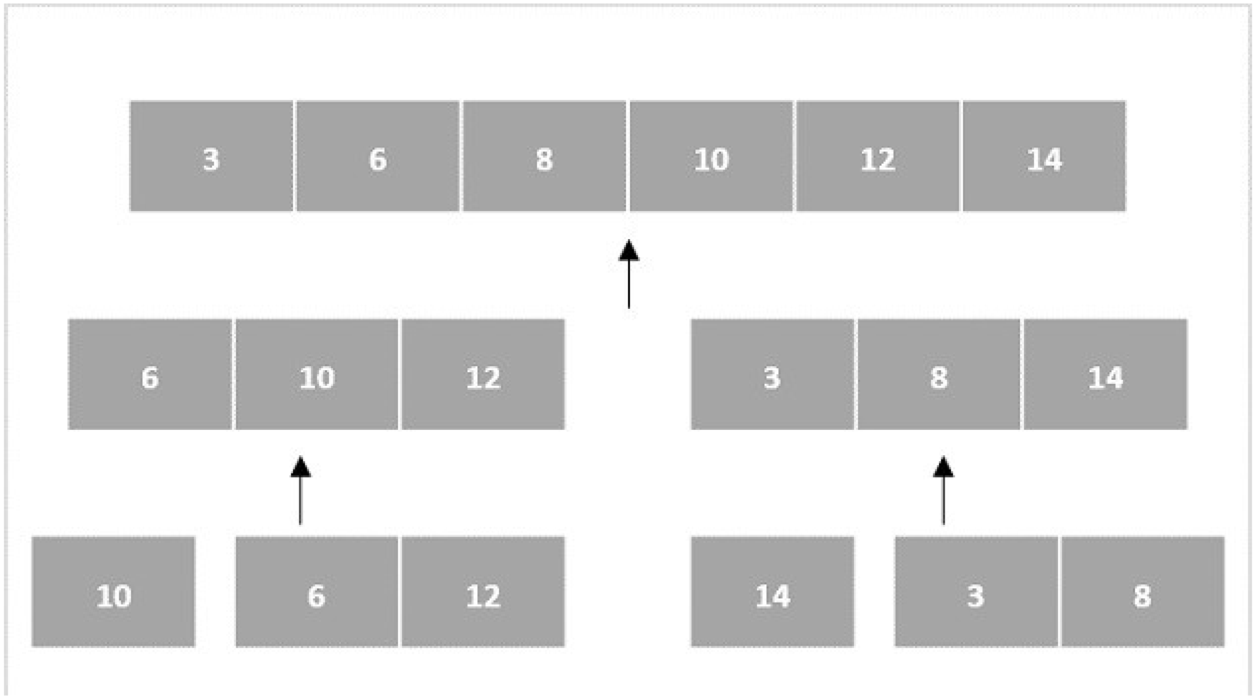
Arrays as Input

There are various ways in which various algorithms can take input such that they can be solved using the divide and conquer technique. Arrays are one of them. In algorithms that require input to be in the form of a list, like various sorting algorithms, array data structures are most commonly used.

In the input for a sorting algorithm below, the array input is divided into subproblems until they cannot be divided further.



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).

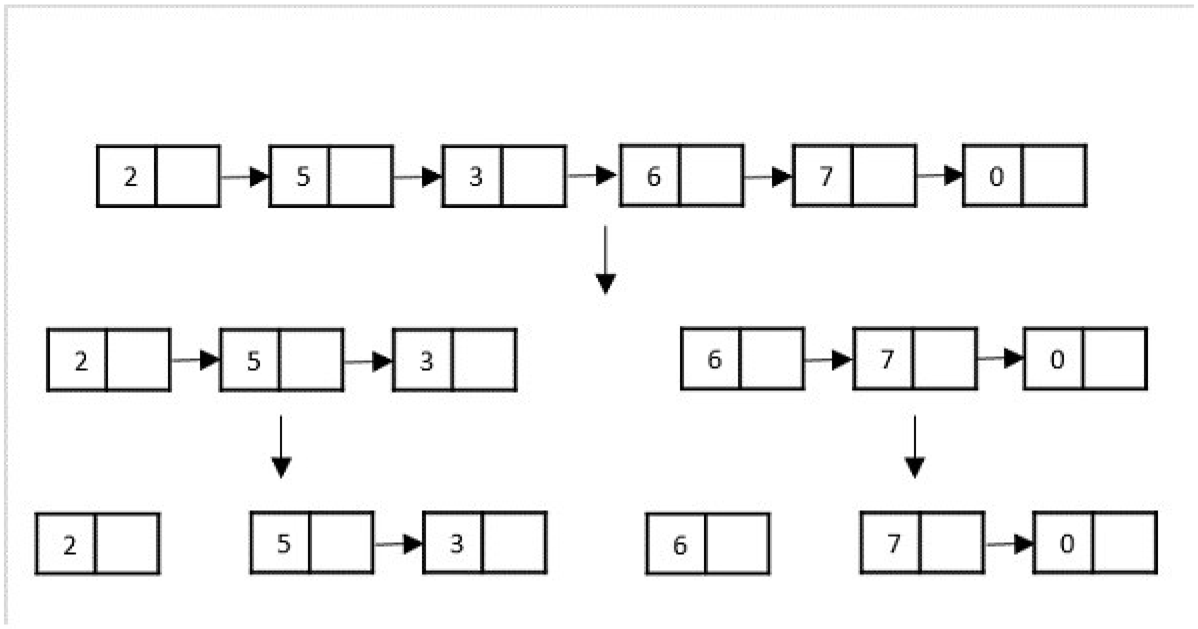


Since arrays are indexed and linear data structures, sorting algorithms most popularly use array data structures to receive input.

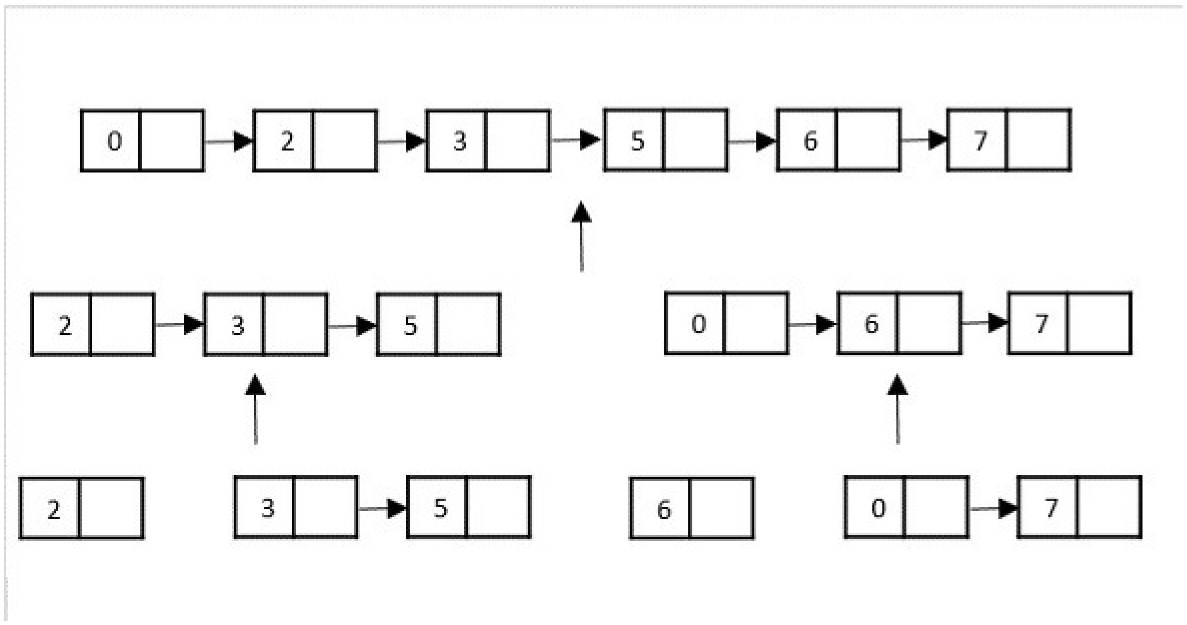
Linked Lists as Input

Another data structure that can be used to take input for divide and conquer algorithms is a linked list (for example, merge sort using linked lists). Like arrays, linked lists are also linear data structures that store data sequentially.

Consider the merge sort algorithm on linked list; following the very popular tortoise and hare algorithm, the list is divided until it cannot be divided further.



Then, the nodes in the list are sorted (conquered). These nodes are then combined (or merged) in recursively until the final solution is achieved.



Various searching algorithms can also be performed on the linked list data structures with a slightly different technique as linked lists are not indexed linear data structures. They must be handled using the pointers available in the nodes of the list.

Examples

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

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest Pair

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Data Structures - Dynamic Programming

Previous Page

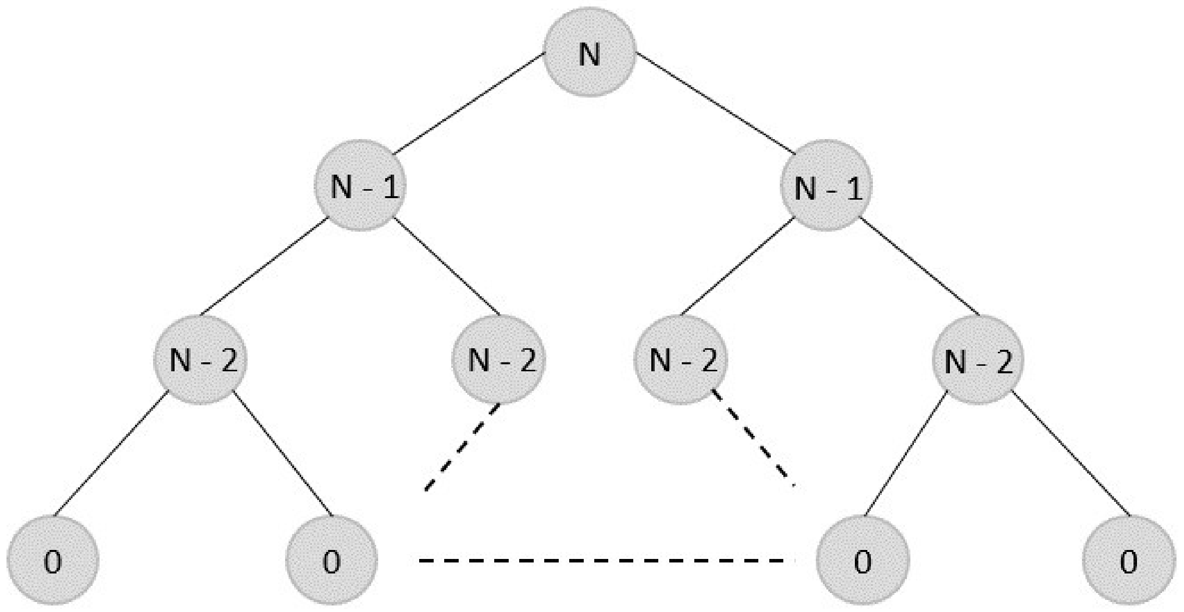
Next Page

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization.



Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach −

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

Data Structures & Algorithm Basic Concepts

This chapter explains the basic terms related to data structure.

Data Definition

Data Definition defines a particular data with the following characteristics.

- **Atomic** − Definition should define a single concept.
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations

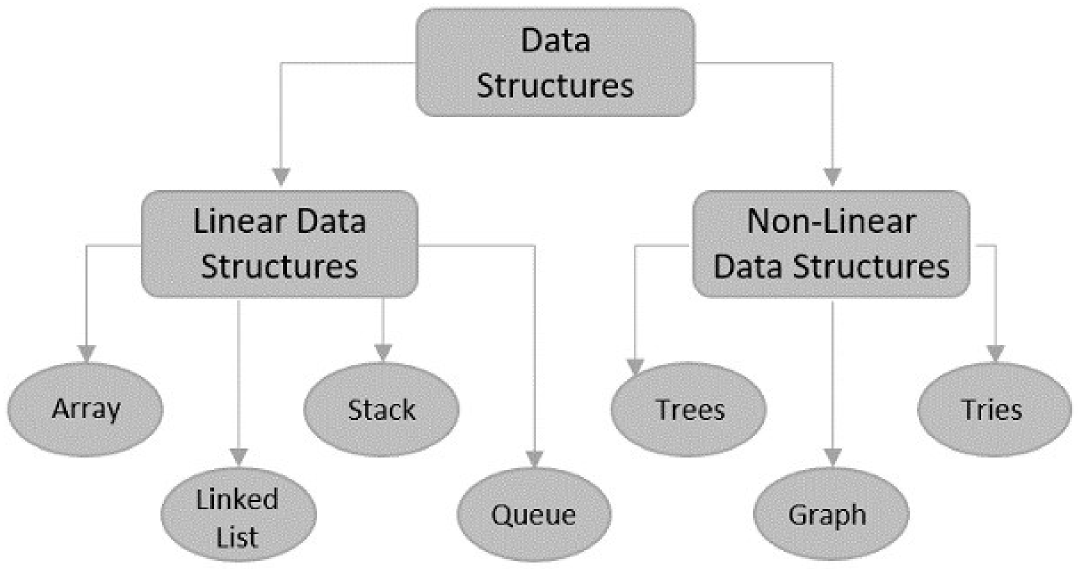
The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Data Structures and Types

Data structures are introduced in order to store, organize and manipulate data in programming languages. They are designed in a way that makes accessing and processing of the data a little easier and simpler. These data structures are not confined to one particular programming language; they are just pieces of code that structure data in the memory.

Data types are often confused as a type of data structures, but it is not precisely correct even though they are referred to as Abstract Data Types. Data types represent the nature of the data while data structures are just a collection of similar or different data types in one.

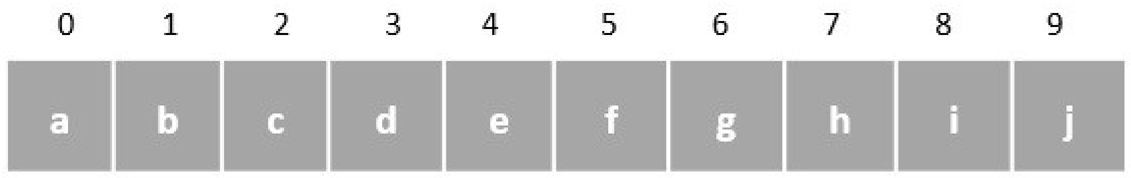


There are usually just two types of data structures −

- Linear
- Non-Linear

Linear Data Structures

The data is stored in linear data structures sequentially. These are rudimentary structures since the elements are stored one after the other without applying any mathematical operations.



Linear data structures are usually easy to implement but since the memory allocation might become complicated, time and space complexities increase. Few examples of linear data structures include −

- Arrays
- Linked Lists
- Stacks
- Queues

Based on the data storage methods, these linear data structures are divided into two sub-types. They are − **static** and **dynamic** data structures.

Static Linear Data Structures

In Static Linear Data Structures, the memory allocation is not scalable. Once the entire memory is used, no more space can be retrieved to store more data. Hence, the memory is required to be reserved based on the size of the program. This will also act as a drawback since reserving more memory than required can cause a wastage of memory blocks.

The best example for static linear data structures is an array.

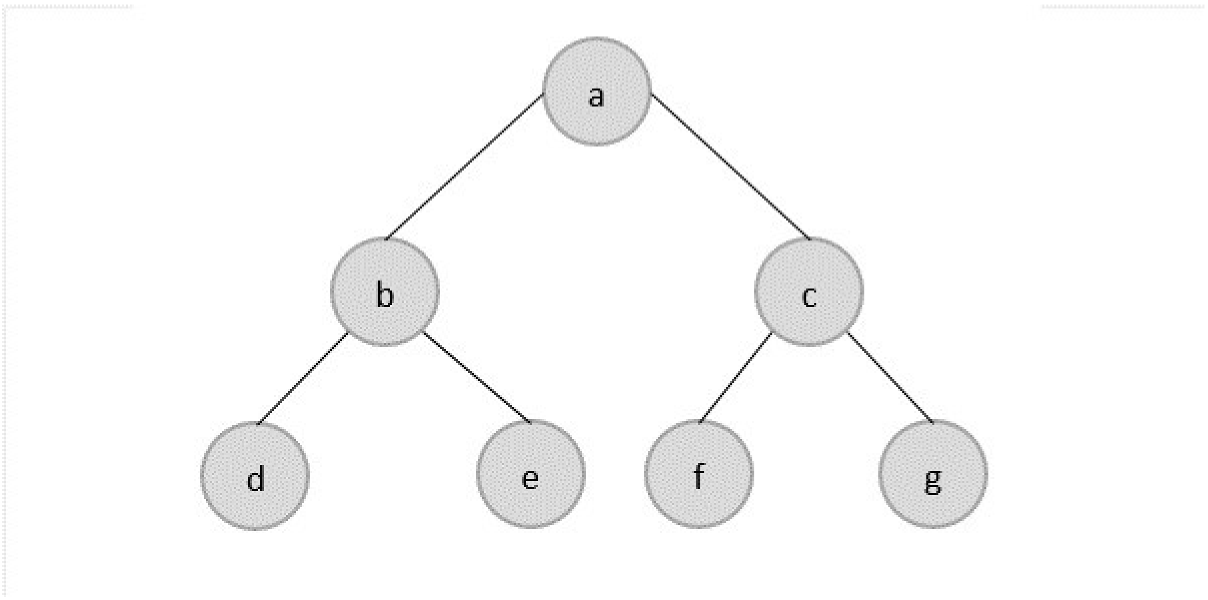
Dynamic Linear Data Structures

In Dynamic linear data structures, the memory allocation can be done dynamically when required. These data structures are efficient considering the space complexity of the program.

Few examples of dynamic linear data structures include: linked lists, stacks and queues.

Non-Linear Data Structures

Non-Linear data structures store the data in the form of a hierarchy. Therefore, in contrast to the linear data structures, the data can be found in multiple levels and are difficult to traverse through.



However, they are designed to overcome the issues and limitations of linear data structures. For instance, the main disadvantage of linear data structures is the memory allocation. Since the data is allocated sequentially in linear data structures, each element in these data structures uses one whole memory block. However, if the data uses less memory than the assigned block can hold, the extra memory space in the block is wasted.

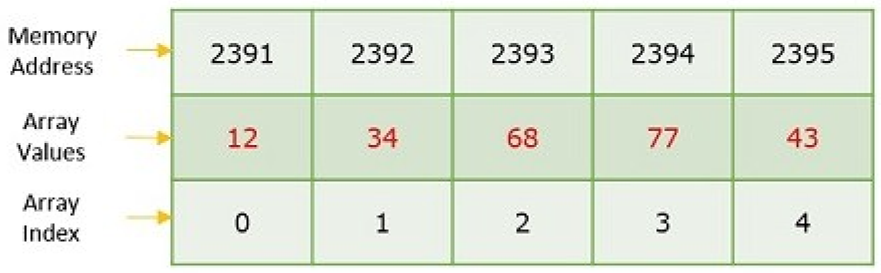
Therefore, non-linear data structures are introduced. They decrease the space complexity and use the memory optimally.

Few types of non-linear data structures are −

- Graphs
- Trees
- Tries
- Maps

Data Structures and Algorithms - Arrays

Array is a type of linear data structure that is defined as a collection of elements with same or different data types. They exist in both single dimension and multiple dimensions. These data structures come into picture when there is a necessity to store multiple elements of similar nature together at one place.



The difference between an array index and a memory address is that the array index acts like a key value to label the elements in the array. However, a memory address is the starting address of free memory available.

Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Syntax

Creating an array in **C** and **C++** programming languages −

```
data_type array_name[array_size] = {elements separated using commas}
or,
data_type array_name[array_size];
```

Creating an array in **JAVA** programming language −

```
data_type[] array_name = {elements separated by commas}
or,
data_type array_name = new data_type[array_size];
```

Need for Arrays

Arrays are used as solutions to many problems from the small sorting problems to more complex problems like travelling salesperson problem. There are many data structures other than arrays that provide efficient time and space complexity for these problems, so what makes using arrays better? The answer lies in the random access lookup time.

Arrays provide **O(1)** random access lookup time. That means, accessing the 1st index of the array and the 1000th index of the array will both take the same time. This is due to the fact that array comes with a pointer and an offset value. The pointer points to the right location of the memory and the offset value shows how far to look in the said memory.

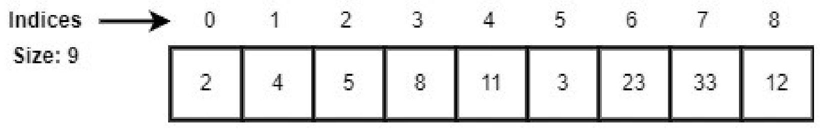
```
                    array_name[index]
                      |      |
                    Pointer   Offset
```

Therefore, in an array with 6 elements, to access the 1st element, array is pointed towards the 0th index. Similarly, to access the 6th element, array is pointed towards the 5th index.

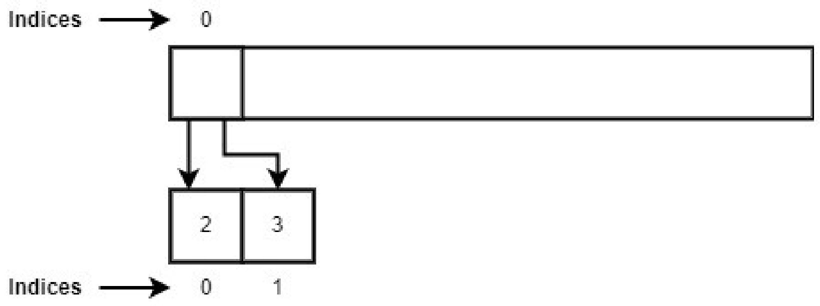
Array Representation

Arrays are represented as a collection of buckets where each bucket stores one element. These buckets are indexed from ‘0’ to ‘n-1’, where n is the size of that particular array. For example, an array with size 10 will have buckets indexed from 0 to 9.

This indexing will be similar for the multidimensional arrays as well. If it is a 2-dimensional array, it will have sub-buckets in each bucket. Then it will be indexed as array_name[m][n], where m and n are the sizes of each level in the array.



Single Dimensional Array



Multi Dimensional Array

As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 9 which means it can store 9 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 23.

Basic Operations in the Arrays

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.
- **Display** − Displays the contents of the array.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
bool	false

char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation

In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. This is done using input statements of the programming languages.

Algorithm

Following is an algorithm to insert elements into a Linear Array until we reach the end of the array −

```
1. Start
2. Create an Array of a desired datatype and size.
3. Initialize a variable ‘i’ as 0.
4. Enter the element at ith index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop
```

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

C C++Java

```
#include <stdio.h>

int main(){

   int LA[3], i;
   printf("Array Before Insertion:\n");
   for(i = 0; i < 3; i++)
      printf("LA[%d] = %d \n", i, LA[i]);
   printf("Inserting Elements.. ");
   printf("The array elements after insertion :\n"); // prints array
values
   for(i = 0; i < 3; i++) {
      LA[i] = i + 2;
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   return 0;
}
```

Output

```
Array Before Insertion:
LA[0] = 587297216
LA[1] = 32767 LA[2] = 0
Inserting Elements.. The array elements after insertion :
LA[0] = 2
LA[1] = 3
LA[2] = 4
```

For other variations of array insertion operation, click here.

Deletion Operation

In this array operation, we delete an element from the particular index of an array. This deletion operation takes place as we assign the value in the consequent index to the current index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the K^th position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++Java

```
#include <stdio.h>
void main(){
   int LA[] = {1,3,5};
   int n = 3;
   int i;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++)
      printf("LA[%d] = %d \n", i, LA[i]);
   for(i = 1; i<n; i++) {
      LA[i] = LA[i+1];
      n = n – 1;
   }
   printf("The array elements after deletion :\n");
   for(i = 0; i<n-1; i++)
      printf("LA[%d] = %d \n", i, LA[i]);
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
The array elements after deletion :
LA[0] = 1
LA[1] = 5
```

Search Operation

Searching an element in the array using a key; The key element sequentially compares every value in the array to check if the key is present in the array or not.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++Java

```
#include <stdio.h>
void main(){
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   for(i = 0; i<n; i++) {
      if( LA[i] == item ) {
         printf("Found element %d at position %d\n", item, i+1);
      }
   }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
```

```
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Traversal Operation

This operation traverses through all the elements of an array. We use loop statements to carry this out.

Algorithm

Following is the algorithm to traverse through all the elements present in a Linear Array −

```
1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable ‘i’ with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End
```

Example

Following are the implementations of this operation in various programming languages −

C C++Java

```c
#include <stdio.h>
int main(){
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

Output

```
The original array elements are :
LA[0] = 1
```

```
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++Java

```c
#include <stdio.h>
void main(){
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   LA[k-1] = item;
   printf("The array elements after updation :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5 LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Display Operation

This operation displays all the elements in the entire array using a print statement.

Algorithm

```
1. Start
2. Print all the elements in the Array
3. Stop
```

Example

Following are the implementations of this operation in various programming languages −

C C++Java

```
#include <stdio.h>

int main(){

   int LA[] = {1,3,5,7,8};

   int n = 5;

   int i;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {

      printf("LA[%d] = %d \n", i, LA[i]);

   }

}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

Implementation and Analysis of Basic Algorithms Practical

Objective: The objective of this practical is to implement and analyze basic algorithms, including searching and sorting algorithms, and gain hands-on experience in measuring their efficiency.

1. Linear Search Algorithm:
- Implement the linear search algorithm in a programming language of your choice.
- Create a test case with a large array and search for a specific element.
- Measure the runtime of the linear search algorithm using appropriate timing techniques.
- Analyze the time complexity of the linear search algorithm and compare it with the theoretical time complexity.

2. Binary Search Algorithm:
- Implement the binary search algorithm in a programming language of your choice.
- Create a test case with a sorted array and search for a specific element.
- Measure the runtime of the binary search algorithm using appropriate timing techniques.
- Analyze the time complexity of the binary search algorithm and compare it with the theoretical time complexity.

3. Bubble Sort Algorithm:
- Implement the bubble sort algorithm in a programming language of your choice.
- Create a test case with an unsorted array.
- Measure the runtime of the bubble sort algorithm using appropriate timing techniques.
- Analyze the time complexity of the bubble sort algorithm and compare it with the theoretical time complexity.
- Compare the measured runtime with the runtime of other sorting algorithms, such as insertion sort or selection sort.

4. Selection Sort Algorithm:

- Implement the selection sort algorithm in a programming language of your choice.
- Create a test case with an unsorted array.
- Measure the runtime of the selection sort algorithm using appropriate timing techniques.
- Analyze the time complexity of the selection sort algorithm and compare it with the theoretical time complexity.
- Compare the measured runtime with the runtime of other sorting algorithms, such as bubble sort or insertion sort.

5. Insertion Sort Algorithm:

- Implement the insertion sort algorithm in a programming language of your choice.
- Create a test case with an unsorted array.
- Measure the runtime of the insertion sort algorithm using appropriate timing techniques.
- Analyze the time complexity of the insertion sort algorithm and compare it with the theoretical time complexity.
- Compare the measured runtime with the runtime of other sorting algorithms, such as bubble sort or selection sort.

6. Analysis and Comparison:

- Compare the measured runtimes of the implemented algorithms with their theoretical time complexities.
- Discuss any variations observed and possible reasons behind them.
- Compare the efficiency of different algorithms for various input sizes.
- Analyze the advantages and disadvantages of each algorithm based on their time complexities and runtime measurements.

Note: Document your implementations, runtime measurements, analysis, and comparisons in a report or presentation format. Include any observations, insights, and conclusions drawn from the practical exercise.