Unit 1 – Introduction to Data Structure

A **data structure** is a way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. It defines the relationship between the data and the operations that can be performed on it. Data structures are essential in computer science because they provide the foundation for storing and organizing data in a way that allows for effective use in algorithms and real-world applications.

Basic Terminology :

- Data: Data can be defined as an elementary value or the collection of value. For example, student's name and its id are the data about the student.
- Information : It can be defined as meaningful data or processed data.
- Datatype : It refers to what kind of data may appear in computation.
- Group Items: Data items which have subordinate data items are called Group item. For example, name of a student can have first name and the last name.
- Record: Record can be defined as the collection of various data items. For example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.
- File: A File is a collection of various records of one type of entity, For example, if there are 60 students in the class, then there will be 60 records in the related file where each record contains the data about each student.
- ATTRIBUTE AND ENTITY: An entity represents the class of certain objects. It contains various attributes. Each attribute represents the particular property of that entity.

Need of Data Structures :

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

- Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

- Data Search: Consider an inventory size of 1000 items in a store, If our application needs to search for a particular item, it needs to traverse 1000 items every time, results in slowing down the search process.
- Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process. In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

An **Abstract Data Type (ADT)**

It is a high-level description or model of a data structure that defines a set of operations without specifying the details of how these operations are implemented. In other words, an ADT is a logical view of data, independent of any specific implementation. It defines what operations are to be performed on the data but not how the data should be structured or the operations executed.

Key Characteristics of an ADT:

1. **Encapsulation**: ADTs hide the internal details of how the data is organized and allow users to interact with it only through the defined operations.
2. **Operations**: An ADT specifies a set of operations that can be performed on the data, such as adding, removing, or accessing elements. These operations define the functionality of the ADT.
3. **Abstract Nature**: ADTs provide an abstraction of data, meaning they define what operations are possible but not how those operations are carried out.

Example:

Consider a **Stack** ADT:

- **Operations**:
    - push(element): Add an element to the top of the stack.
    - pop(): Remove the top element from the stack.
    - peek(): View the top element without removing it.
    - isEmpty(): Check whether the stack is empty.

While the ADT defines these operations, it does not specify how these operations are implemented. Internally, the stack could be implemented using an array or a linked list, but this detail is abstracted away from the user.

Definition of Data Structure :

A *data structure* D is a triplet, that is, D = ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) where $\mathcal{D}$ is a set of data object, $\mathcal{F}$ is a set of functions and $\mathcal{A}$ is a set of rules to implement the functions.

Data object or Domain D

The **domain** of a data structure refers to the set of values or elements that the data structure can hold or represent. It defines the collection of possible data items that can be stored in the data structure.

- For example, in the case of an **integer stack**, the domain would be the set of integers, i.e., {0, 1, -1, 100, -50, ...}.

2. Functions F

The **function** of a data structure describes the operations or actions that can be performed on the data structure. These functions define how data can be accessed, modified, or manipulated.

- For a **stack** data structure, the typical functions are:
  - push(element): Adds an element to the top of the stack.
  - pop(): Removes and returns the top element of the stack.
  - peek(): Returns the top element without removing it.
  - isEmpty(): Checks if the stack is empty.

Each of these functions operates on the elements in the domain (the stack) and may modify or return the data depending on the operation.

3. Axioms A

Axioms define the rules, properties, and constraints that govern the operations on the data structure. They describe the logical behavior that must be maintained when the functions are applied.

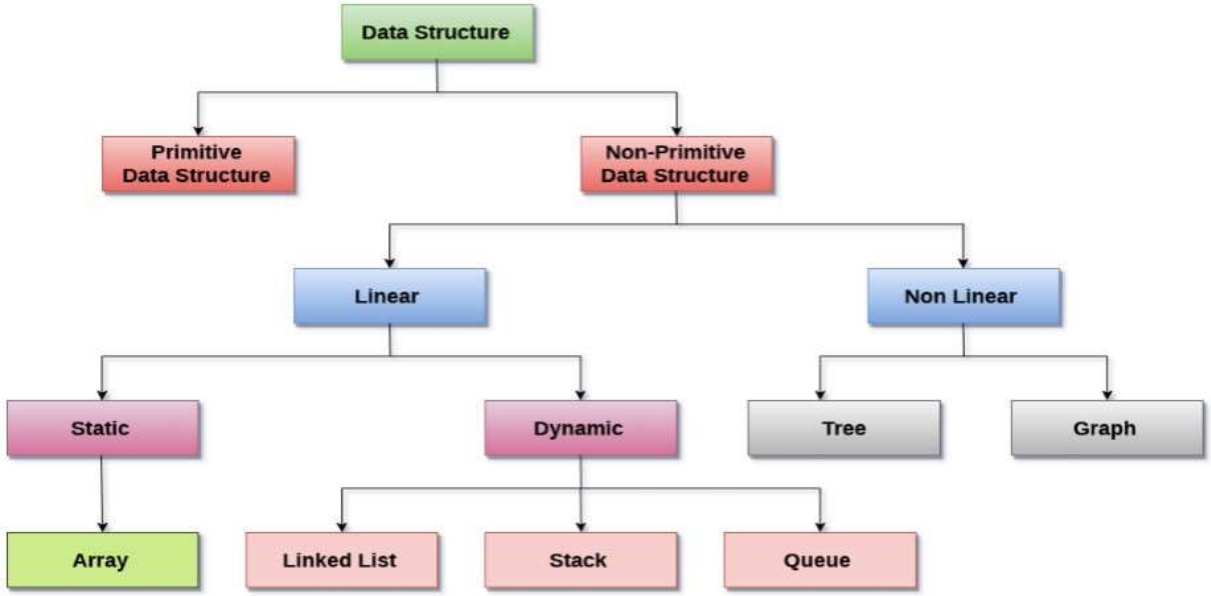
- For example, for a **stack** (which follows a Last-In-First-Out, or LIFO, principle), the axioms might include:

- pop(push(x)) = x: If an element x is pushed onto the stack, and then popped, it must return x.
- peek() always returns the element that is at the top of the stack.
- isEmpty() must return true only when the stack contains no elements.

Advantages of Data Structures

- Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
- Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.
- **Better Memory Management**: Data structures like linked lists allow dynamic memory allocation, ensuring that memory is used as needed and not wasted by predefined fixed sizes like arrays.

DATA STRUCTURE CLASSIFICATION



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures:

1. Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

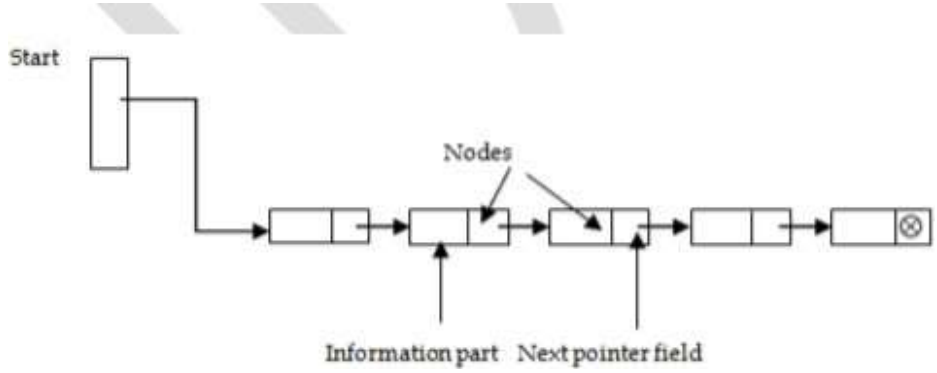
The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

```
int Num [5] = { 26, 7, 67, 50, 66 };
```

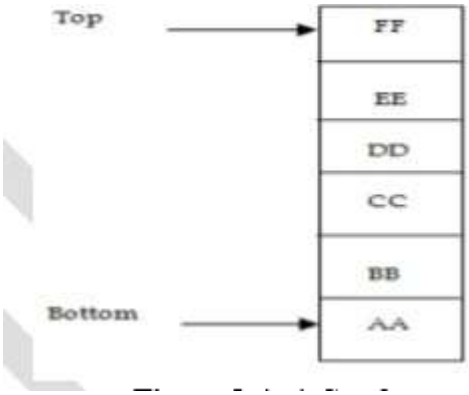
This declaration will create an array as shown below:

	0	1	2	3	4
Num	26	7	67	50	66

2. Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.



3. Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called top. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack. For example: - piles of plates or deck of cards etc.



4. Queue: Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

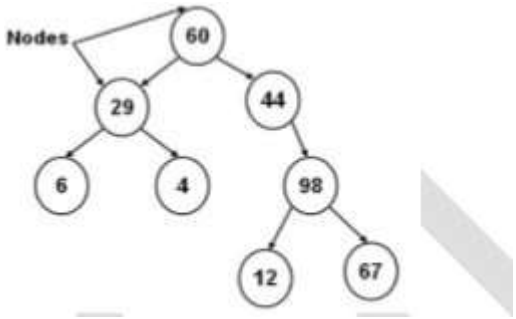
It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-InFirst-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures:

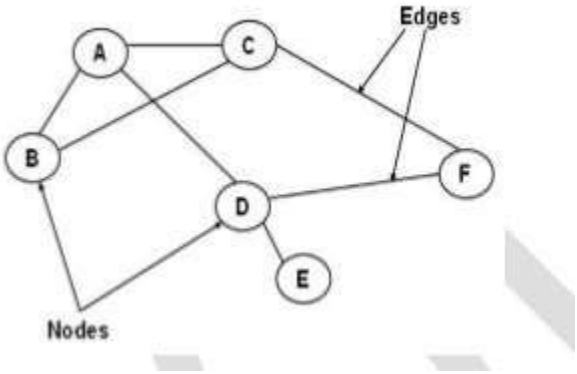
1. Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories.



2. Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph

is different from tree in the sense that a graph can have cycle while the tree cannot have the one.



Operations on data structure

1) Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting. Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is n then we can only insert n-1 data elements into it.

3) Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location. If we try to delete an element from an empty data structure then underflow occurs.

4) Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

5) Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

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.

Real Time Applications of Data Structure :

1. Databases

- **Data Structures Used**: B-Trees, Hash Tables, Arrays, Linked Lists.
- **Real-Time Application**: Databases use B-Trees or B+ Trees to organize large amounts of data efficiently for quick searching, insertion, and deletion. Hash tables are used for indexing and retrieving records quickly. Arrays and linked lists are used in various internal operations, such as caching or storing records.

2. Web Browsers

- **Data Structures Used**: Stacks, Queues, Hash Tables, Trees.
- **Real-Time Application**: Web browsers use stacks to manage the history of visited pages (back and forward buttons). Queues are used to manage tasks like the rendering of pages. Hash tables are used for storing the data for fast retrieval, such as in cookies or cache management. Trees are used in rendering the structure of web pages (DOM trees).

3. Operating Systems

- **Data Structures Used**: Linked Lists, Trees, Queues, Stacks, Hash Tables.
- **Real-Time Application**: Operating systems use queues for process scheduling (e.g., round-robin scheduling), stacks for managing function calls (call stack), and trees for managing file systems (e.g., directory structure). Linked lists and hash tables are used in memory management, file systems, and virtual memory handling.

4. Social Media Platforms

- **Data Structures Used**: Graphs, Hash Tables, Arrays, Queues.
- **Real-Time Application**: Social media platforms use graphs to represent user relationships (e.g., followers, friends) and to find connections (e.g., friend suggestions). Hash tables are used to quickly retrieve user data or posts. Queues are used to manage real-time notifications, and arrays are used for storing posts, comments, or media.

5. Search Engines

- **Data Structures Used**: Hash Tables, Trie, Arrays, Trees.
- **Real-Time Application**: Search engines like Google use trie data structures for fast word lookup and autocompletion of search queries. Hash tables are used for indexing and retrieving search results efficiently. Inverted indexes (built using trees or arrays) are used to map keywords to their location in the search database.

6. Financial Systems (Trading Platforms)

- **Data Structures Used**: Queues, Priority Queues, Heaps, Hash Tables.
- **Real-Time Application**: In trading platforms, orders are often managed using queues or priority queues (to process buy and sell orders based on priority). Hash tables are used for quick lookup of stock prices, and heaps are used to maintain a sorted order of buy/sell orders

Algorithm and its Characteristics

Definition of Algorithm : An **algorithm** is a **step-by-step procedure** or a **set of well-defined rules** designed to solve a specific problem or perform a task. It is a finite sequence of instructions that, when executed, leads to the desired output from a given input in a finite amount of time.

Characteristics of an Algorithm:

Input: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.

Output: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.

Definiteness: All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.

Finiteness: An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.

Effectiveness: An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

1.9 ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation.

They are the following –

1) **Priori Analysis or Performance or Asymptotic Analysis :**

This is a theoretical analysis of an algorithm performed before its actual implementation and execution. It focuses on determining the efficiency of an algorithm based on its logical structure and the number of operations it performs, independent of specific hardware or programming language. It uses mathematical tools and concepts like Big O notation, Big Omega notation, and Big Theta notation to express the algorithm's time complexity (how runtime grows with input size) and space complexity (how memory usage grows with input size).

2) Posteriori Analysis or Performance Measurement:

This is an empirical analysis of an algorithm performed after its implementation and execution on a specific computing system. It aims to measure the actual performance of the algorithm in a real-world environment. It involves implementing the algorithm in a programming language, running it with various test cases on a target machine, and collecting actual statistics like execution time and memory consumption using profiling and benchmarking tools.

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.

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length

to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

Frequency count and **program steps** are important concepts in the analysis of algorithms, used to measure and understand the **time complexity** or **performance** of an algorithm. These techniques help in estimating how an algorithm's running time grows as the input size increases.

1. Frequency Count

Frequency count refers to the process of counting how many times specific operations or steps are executed during the execution of an algorithm. This helps to estimate the overall time complexity of an algorithm.

Program step: Program step is the syntactically or semantically meaningful segment of a program. And it has an execution time that is independent of the instance characteristics.

A program step refers to a single operation that is executed during the algorithm's runtime, such as an arithmetic operation, a comparison, or an assignment. These individual steps are considered when analyzing the efficiency of an algorithm. The total number of program steps helps estimate the algorithm's execution time and complexity

1.8 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 algorithm 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.

1.8.1 **Space Complexity : Space complexity of an algorithm represents the amount of memory space required by the algorithm as a function of the length of the input.**

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.

1.8.2 **Time Complexity** : **Time complexity of an algorithm represents the amount of time required by the algorithm to run as a function of the length of the input.** Time requirements can be defined as a numerical function f(n), where f(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 f(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that f(n) grows linearly as the input size increases.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. BigO notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

Asymptotic Notation:

A problem may have numerous (many) algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem. Asymptotic complexity is a way of expressing the main component of algorithms like

- Cost
- Time complexity
- Space complexity

Some Asymptotic notations are

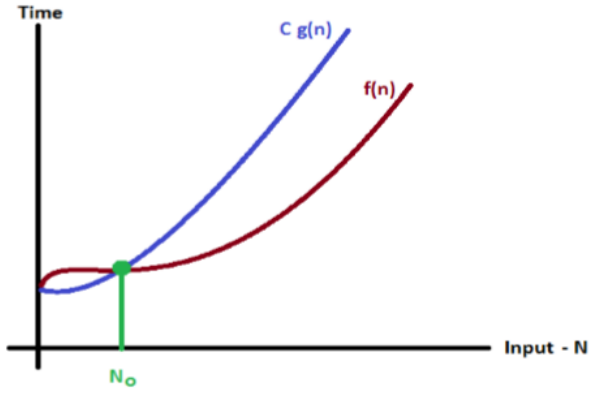
1. Big oh→O
2. Omega→Ω
3. Theta →θ
4. Little oh→o
5. Little Omega→ω

1. Big - Oh Notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

The function f(n) =O(g(n)) (read as "f of n is big oh of g of n) iff (if and only if) there exit positive constants c and n_0 such that f(n)<=c*g(n) for all n, n>=n_0 f(n)=O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C x g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Table 2.1: Common Orders			
Time complexity			Examples
1	O(1)	Constant	Adding to the front of a linked list
2	O(log n)	Logarithmic	Finding an entry in a sorted array
3	O(n)	Linear	Finding an entry in an unsorted array
4	O(n log n)	Linearithmic	Sorting 'n' items by 'divide-and-conquer'
5	O(n2)	Quadratic	Shortest path between two nodes in a graph
6	O(n3)	Cubic	Simultaneous linear equations
7	O(2n)	Exponential	The Towers of Hanoi problem

Big - Omega Notation (Ω)

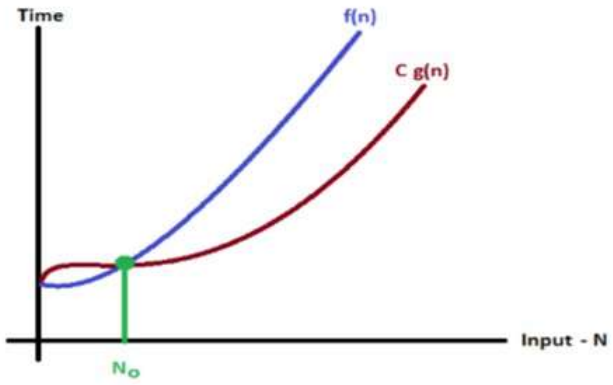
- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

The function f(n) = Ω(g(n)) (read as "f of n is omega of g of n) iff (if and only if) there exit positive constants c and n_0 such that f(n)>=c*g(n) for all n, n>=n_0

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

EXAMPLE

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**

f(n) >= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

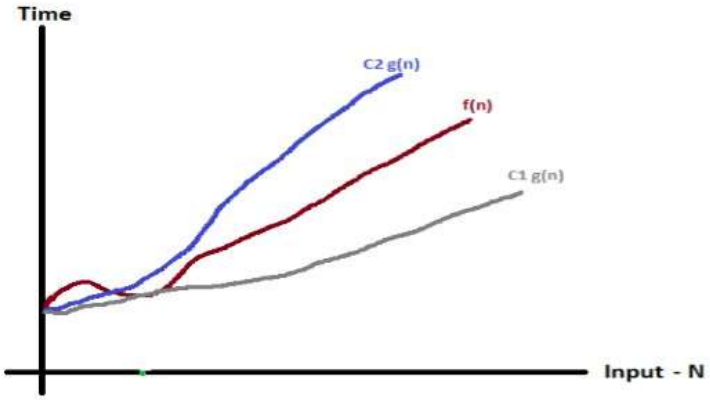
- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

The function f(n) = Θ (g(n)) (read as "f of n is theta of g of n) iff (if and only if) there exist positive constants c_1,c_2 and n_0 such thatc_1*g(n)<= f(n)<=c_2*g(n) for all n, n>=n_0

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always **C_1 g(n)** is less than **f(n)** and **C_2 g(n)** is greater than **f(n)** which indicates the algorithm's average bound.

EXAMPLE

Consider the following f(n) and g(n)...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy $C_1\, g(n) <= f(n) >= C_2\, g(n)$ for all values of C_1, $C_2 > 0$ and $n_0 >= 1$

$$C_1\, g(n) <= f(n) >= \Rightarrow C_2\, g(n)$$

$$C_1\, n <= 3n + 2 >= C_2\, n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n >= 1$.

By using Big - Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$

1.9.1 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

1.9.1.1 Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size n. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

1.9.1.2 Average Case Analysis

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size n to analyze the average case efficiency of algorithm.

1.9.1.3 Worst Case Analysis

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size n. The algorithm runs the longest among all the possible inputs of the similar size because of this input of size n.

Example: Searching an Element in an Unsorted Array (Linear Search)

Best Case

- The element is found at the **first position**.
- **Time Complexity: O(1)**
  Example: Search for 5 in `[5, 12, 7, 9]`
  You check only once → found!

Worst Case

- The element is at the **last position** OR **not present at all**.
- **Time Complexity: O(n)**
  Example: Search for 9 in `[5, 12, 7, 9]`
  You check all elements → found at last.

OR

Search for 100 in `[5, 12, 7, 9]`
You check all elements → not found.

Average Case

- Element may be anywhere in the array with equal probability.
- You check **half** the elements on average.
- **Time Complexity: O(n)**