Chapter-1:

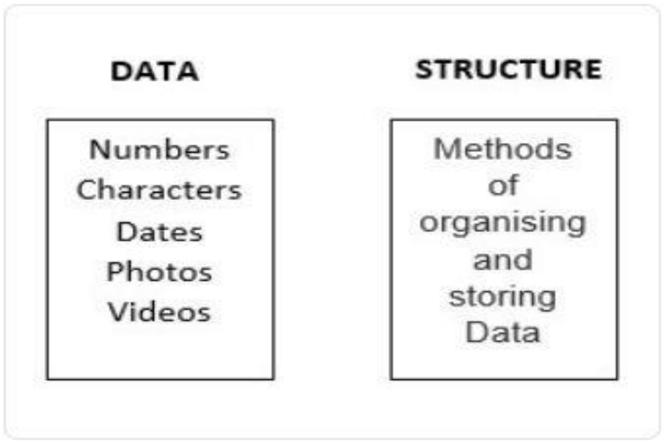
Introduction to Data Structure and Algorithms

1. Data Structures and Algorithms

The name "Data Structure" is a combination of two words: "Data" and "Structure". Let's go over each word individually:

Data : Data refers to facts, figures, or information that are collected and stored for analysis or reference. It can be anything that can be measured or recorded, such as numbers, text, images, or sounds.

Structure : A **structure** is simply an organized way of arranging or putting things together in a specific pattern or system.



Data Structure:

- Data Structure is a way to store, organize and retrieve data in a way that it can be used efficiently.
- Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other. (which tasks depend on each other, or how folders and files are organized on a disk.)

- Data structures are the building blocks of a program. Hence proper selection of data structure increases the productivity of programmers due to the proper designing and the use of efficient algorithms.
- **Example**: In the file path **/home/user/Documents,** the root / is at the top of the tree, with directories branching off as you navigate deeper.

Need of Data Structure:

1. **Improved Performance**: The right data structure enhances time and space efficiency, speeding up operations like searching and sorting.
2. **Memory Optimization**: Some structures, like linked lists, use memory more efficiently than arrays for certain tasks.
3. **Fast Data Retrieval**: Structures like hash tables allow quick access to data, saving time compared to linear searches.
4. **Data Management**: They make data handling easier, like using stacks for undo functionality.
5. **Solving Complex Problems**: Data structures enable efficient algorithms, crucial for solving problems like shortest paths in graphs.

Characteristics of Data Structure

The following are some of the main characteristics of data structures:

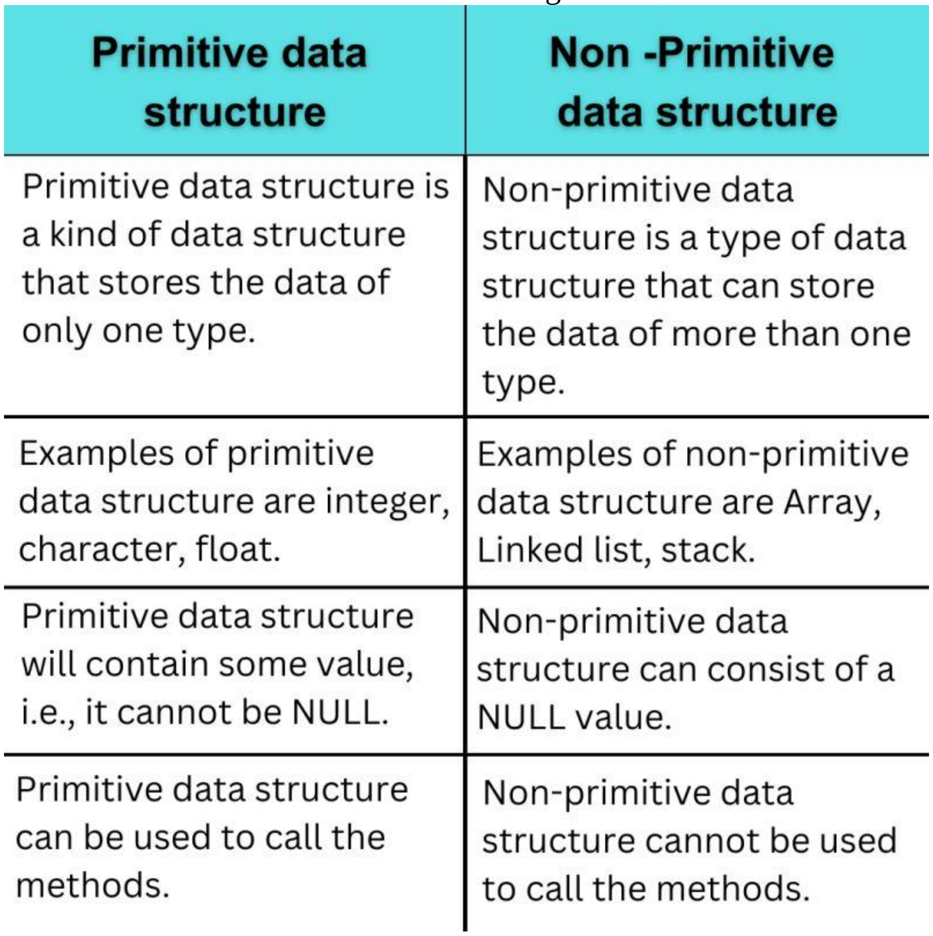
1. **Representation of Data:** Data structures define a way of representing data in a computer's memory, making it possible to store, manipulate, and access data efficiently.
2. **Access Techniques:** Different data structures provide different techniques for accessing data stored within them, such as random access or sequential access.
3. **Storage Organization:** Data structures define the organization of data in memory, such as linear or non-linear (hierarchical) organization.
4. **Insertion and Deletion Operations:** Different data structures support different methods for adding and removing elements, such as insertion at the end or deletion from the front.
5. **Time and Space Complexity:** Data structures can have different time and space complexities, depending on the operations they support and the way they organize data.
6. **Adaptability:** Some data structures are more adaptable to certain types of data and operations than others. For example, a stack is more suitable for problems that require
7. e Last-In-First-Out (LIFO) behavior, while a queue is better suited for problems that require First-In-First-Out (FIFO) behavior.

Algorithm + Data Structure = Program

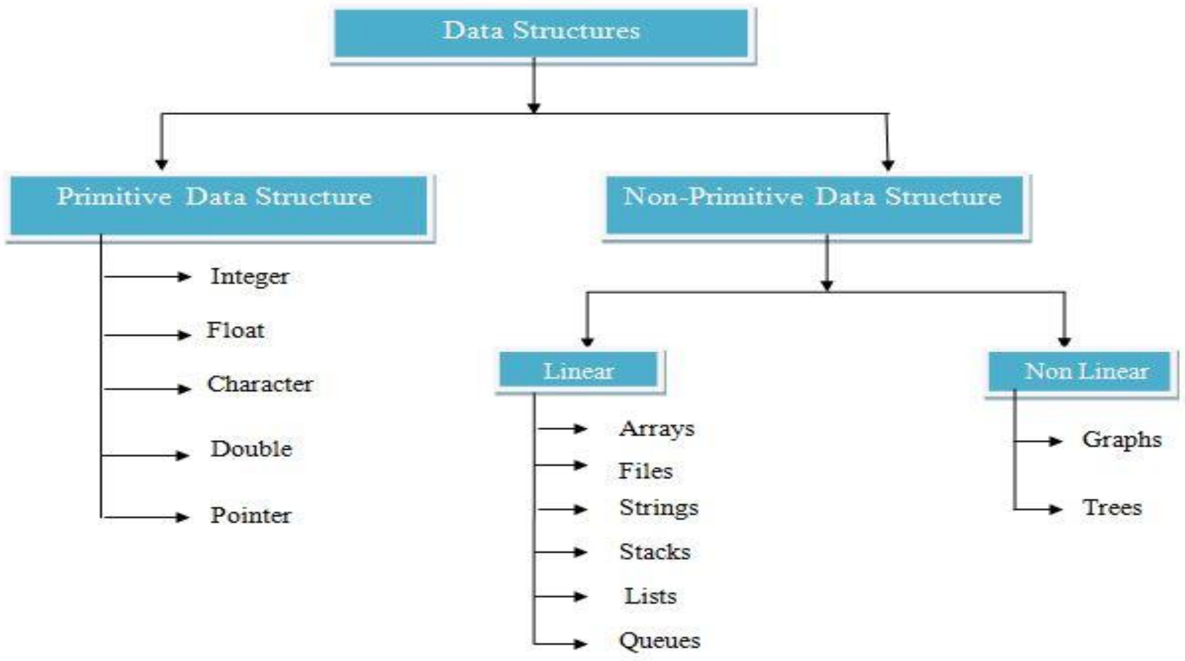
Classification of Data structure:

1. Primitive data structures:
2. Non-primitive data structures:

Data structure can be classified into 2 categories



Primitive data structure	Non -Primitive data structure
Primitive data structure is a kind of data structure that stores the data of only one type.	Non-primitive data structure is a type of data structure that can store the data of more than one type.
Examples of primitive data structure are integer, character, float.	Examples of non-primitive data structure are Array, Linked list, stack.
Primitive data structure will contain some value, i.e., it cannot be NULL.	Non-primitive data structure can consist of a NULL value.
Primitive data structure can be used to call the methods.	Non-primitive data structure cannot be used to call the methods.



Classification of Data Structure

3. Algorithm:

An algorithm is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce some value, or set of values as output.

Algorithms Properties:

1. **Input(s)/output(s):** There must be some inputs from the standard set of inputs and an algorithm's execution must produce outputs(s).
2. **Definiteness**: Each step must be clear and unambiguous.
3. **Finiteness:** Algorithms must terminate after finite time or steps.
4. **Correctness:** Correct set of output values must be produced from the each set of inputs.
5. **Effectiveness:** Each step must be carried out in finite time.

Importance of Algorithm:

- **Solve Problems**: Provide a clear, systematic approach to solving complex problems.
- **Boost Efficiency**: Optimize time and memory use, especially important for large datasets.
- **Ensure Accuracy**: Process data precisely, reducing human error in critical fields like medicine and finance.

- **Enable Automation**: Perform repetitive tasks quickly and consistently.
- **Drive Innovation**: Power advancements in AI, machine learning, and data science.
- **Power Real-world Applications**: Used in search engines, recommendations, navigation, and more.

Abstract Data Type:

What is Abstraction?

- Abstraction is a fundamental principle in object-oriented programming. It allows programmers to hide irrelevant data about an object to make code more efficient and reduce complexity.
- Data abstraction refers to providing only essential information about the data to the outside world, ignoring unnecessary details or implementation.
- Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car.

The **Abstract Data Types** are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

Uses of ADT:

1. **Simplicity**: ADTs allow programmers to work with data structures without getting into implementation details.
2. **Reusability**: Since ADTs specify *what* a data structure does, they can be implemented in multiple ways and reused in different applications.
3. **Modularity**: Changes to the implementation of an ADT do not affect the rest of the program, as long as the operations remain the same.

Examples of ADTs

1. **Stack (LIFO)**:

- Operations: push (add an element), pop (remove an element), and peek (view the top element).
- Usage: Stacks are used in managing function calls in recursion and maintaining browsing history where the last page visited is the first to be returned to on "back."
- **Real-Life Example**: **A Stack of Plates**.

2. **Queue (FIFO)**:
   - Operations: enqueue (add an element at the end) and dequeue (remove an element from the front).
   - Usage: Used in situations where order matters, such as in print job management or task scheduling where documents are printed in the order they are sent to the printer.
   - **Real-Life Example**: **Bank Queue**

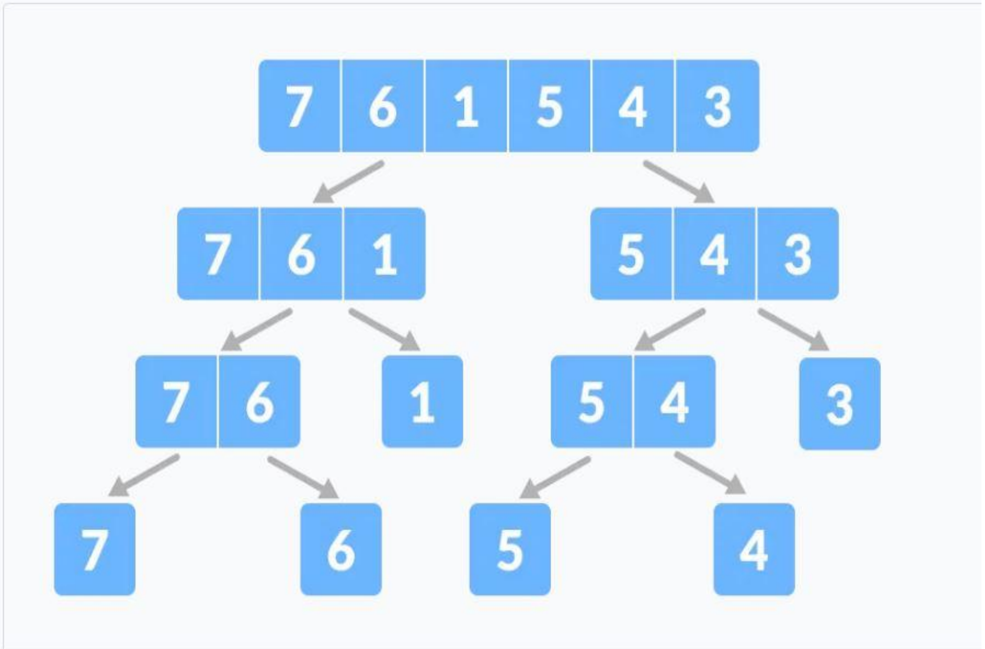
3. **List**:
   - Operations: insert, delete, and find.
   - Usage: Lists are used in creating arrays or linked lists in memory, where elements can be accessed randomly, such as in contact lists on a phone, where you can add, remove, or view any contact directly.
   - **Real-Life Example**: **To-Do List or Shopping List**

Algorithm Design Techniques:

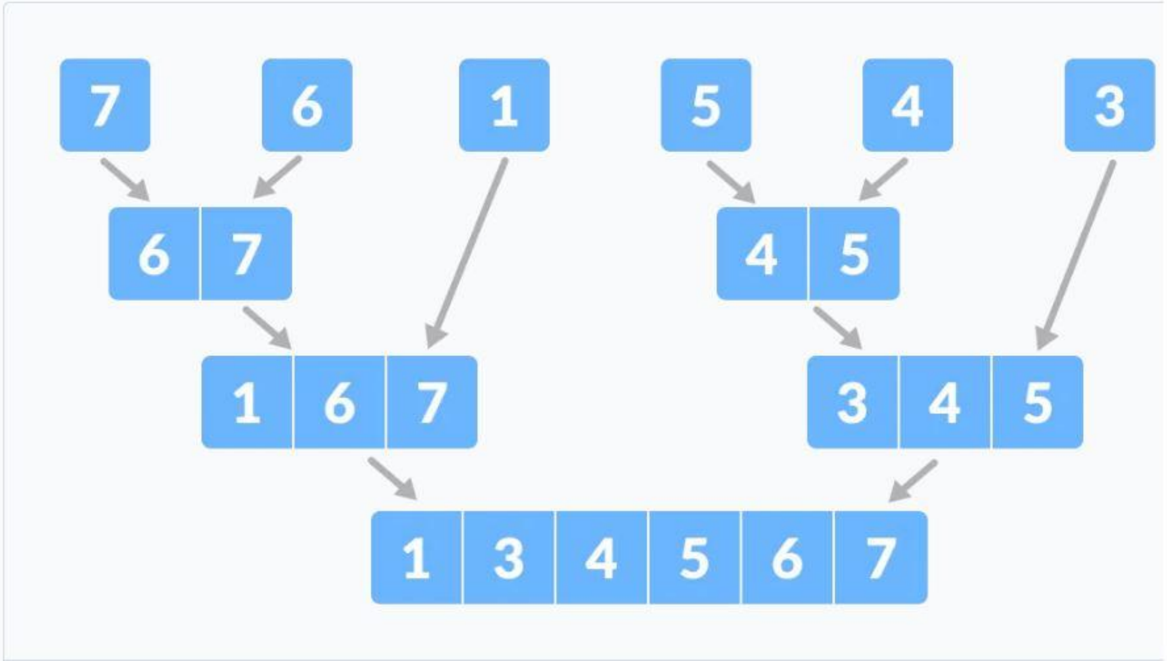
Algorithm Design refers to the process of creating efficient and effective step-by-step procedures to solve computational problems. It involves selecting appropriate mathematical models, data structures, and techniques to develop algorithmic solutions.

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.



Divide the array into smaller subparts



Combine the subparts

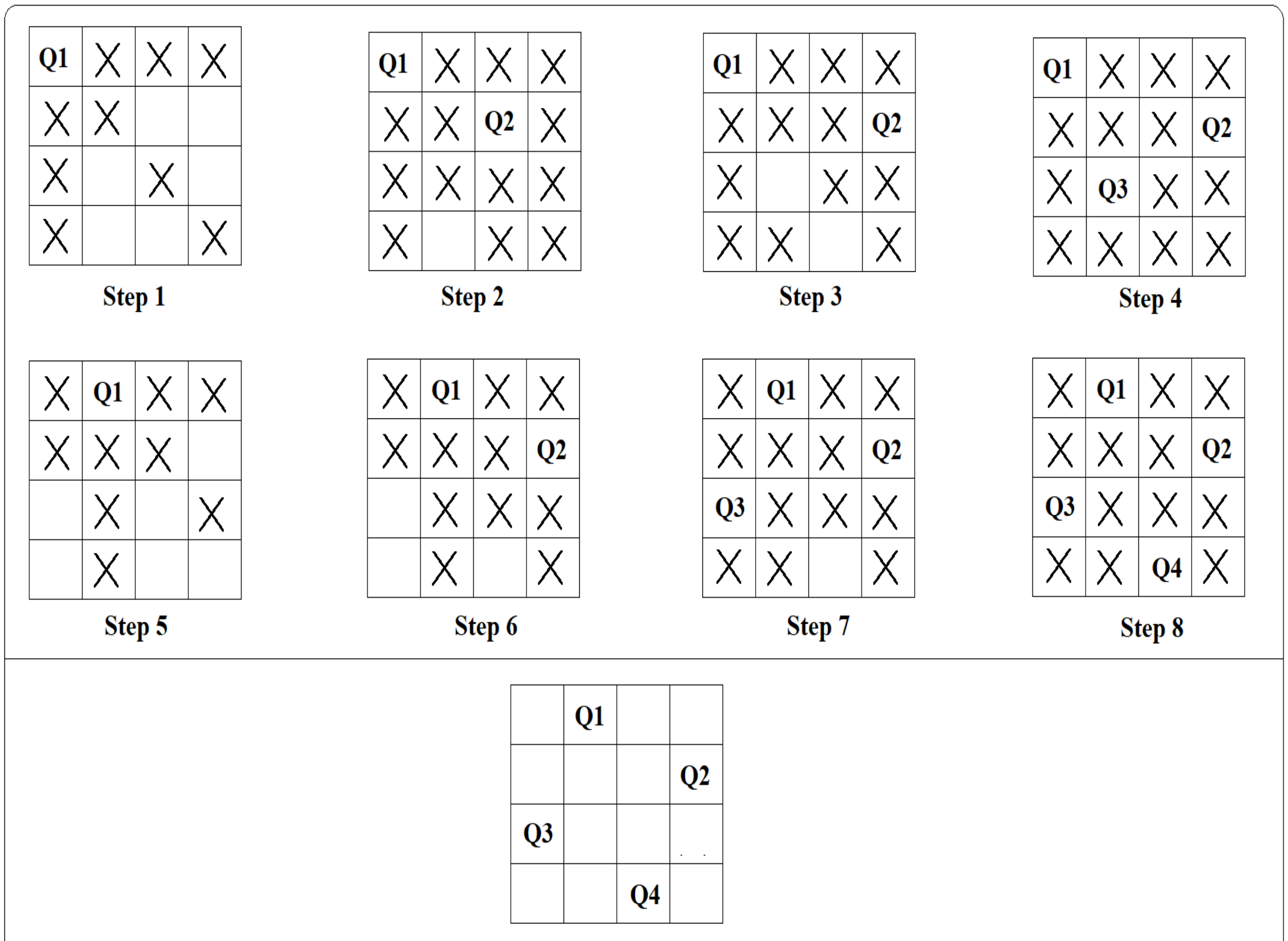
Fig: Merge Sort

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

- **Example:** N-queen problem.



4 Queens Problem Solution 1 - (2, 4, 1, 3)

Algorithm Analysis:

Algorithm analysis is the process of evaluating the performance of an algorithm, usually in terms of its time and space complexity.

The analysis of the algorithms gives a good insight of the algorithms under study.

Analysis of algorithms tries to answer few questions like; is the algorithm correct? i.e., the Algorithm generates the required result or not? does the algorithm terminate for all the inputs under problem domain? The other issues of analysis are efficiency, optimality, etc. So knowing the different aspects of different algorithms on the similar problem domain we can choose the better algorithm for our need. This can be done by knowing the resources needed for the algorithm
for its execution. Two most important resources are the **time** and the **space**. Both

of the resources are measures in terms of complexity for time instead of absolute time we consider growth

Time and Space Complexity
Best, Worst and Average case

Best case complexity :

- The best case scenario is the ideal situation where the algorithm takes the least amount of time or resources to complete its task.
- It gives the upper bound of the algorithm's performance. In other words, it's a way to determine how the algorithm would perform in the best possible scenario, allowing us to understand the lower bound of its performance.

Worst case complexity :

- The worst case scenario is the worst possible situation where the algorithm takes the most amount of time or resources to complete its task.
- It's a way to determine how the algorithm would perform in the worst possible scenario, allowing us to understand the upper bound of its performance.

Average case complexity :

- The average case scenario is the most common or expected situation where the algorithm takes a reasonable amount of time or resources to complete its task.

Why best, average, and worst case scenarios matter ?

The best, average, and worst case scenarios help you evaluate and compare different algorithms based on their efficiency and scalability. They also help you identify the strengths and weaknesses of your algorithm, and how it behaves under different inputs and conditions. For example, if your algorithm has a low best case scenario but a high worst case scenario, it may be very fast for some inputs but very slow for others. You may want to optimize or modify your algorithm to improve its consistency and reliability.

Rate of Growth:
In algorithm analysis, rate of growth refers to how an algorithm's resource needs (like time or space) increase as the input size grows. It essentially describes the efficiency of an algorithm by quantifying how its performance changes with the scale of the problem it solves. This is often expressed using Big O notation, which provides a general upper bound on the growth rate.

Elaboration:

The rate of growth indicates how an algorithm's resource consumption (time, space, etc.) scales with the size of its input. A faster algorithm will have a slower growth rate, meaning its resource needs increase less dramatically as the input grows.

Examples:

- **O(1) (Constant):** The algorithm's performance is independent of the input size.
- **O(log n) (Logarithmic):** The algorithm's performance increases very slowly with the input size.
- **O(n) (Linear):** The algorithm's performance increases proportionally to the input size.
- **O(n log n):** A common growth rate for efficient sorting algorithms.
- **O(n^2) (Quadratic):** The algorithm's performance increases rapidly with the input size.
- **O(2^n) (Exponential):** The algorithm's performance grows extremely quickly with the input size.

Asymptotic Notations:

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

There are mainly three asymptotic notations:

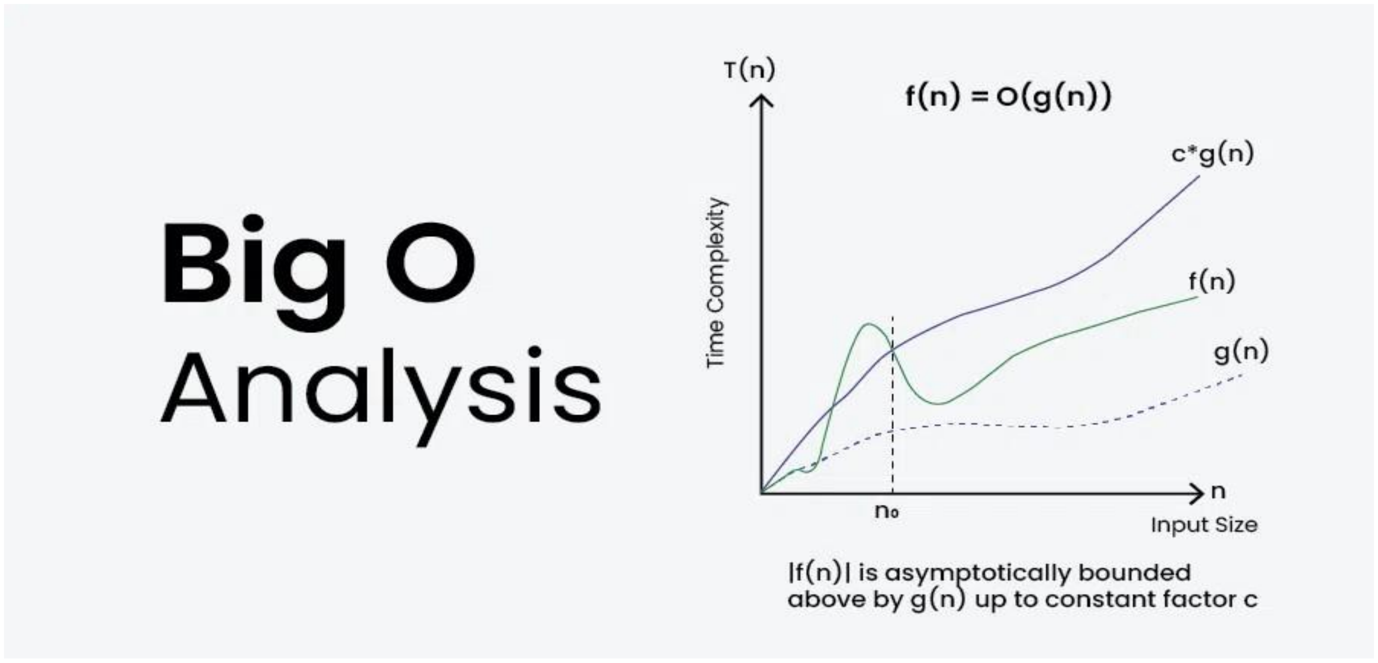
1. ***Big-O Notation (O-notation)***
2. ***Omega Notation (Ω-notation)***
3. ***Theta Notation (Θ-notation)***

- The **upper bound** of an algorithm represents the **maximum** amount of time or resources the algorithm will require to solve a problem for a given input size.
- The **lower bound** represents the **minimum** amount of time or resources an algorithm will need to solve a problem for a given input size.

1. **Big O Notation (O)**

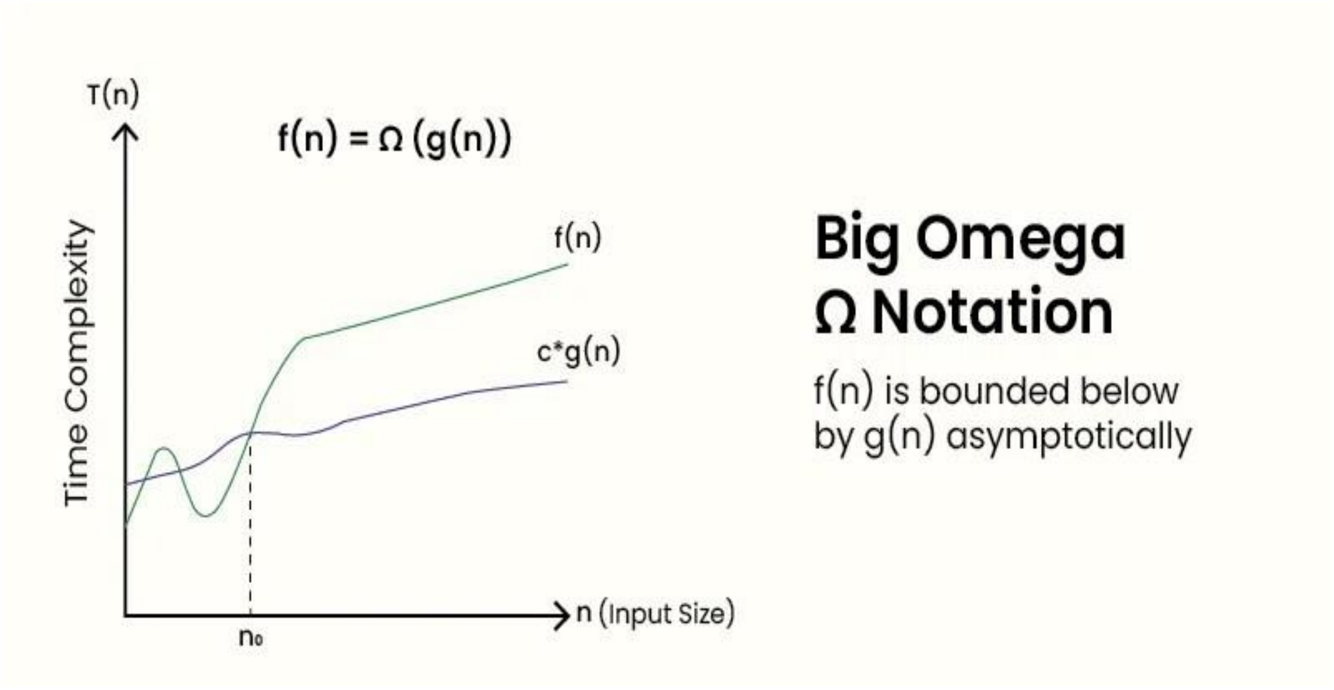
- **Definition**: Big O notation describes the **upper bound** of an algorithm's running time. It gives the worst-case time complexity, indicating the maximum amount of time an algorithm can take for large input sizes.
-
- Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
- Can be used to compare the efficiency of different algorithms or data structures.
- It provides an **upper limit** on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O.
- It’s denoted as **O(f(n))**, where **f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.

- **Use Case**: Used to guarantee that an algorithm will not take more than a certain time, even in the worst case.
- **Example**: If an algorithm’s time complexity is O(n^2), it means that as the input size n grows, the execution time will increase at a rate proportional to n^2 in the worst-case scenario.



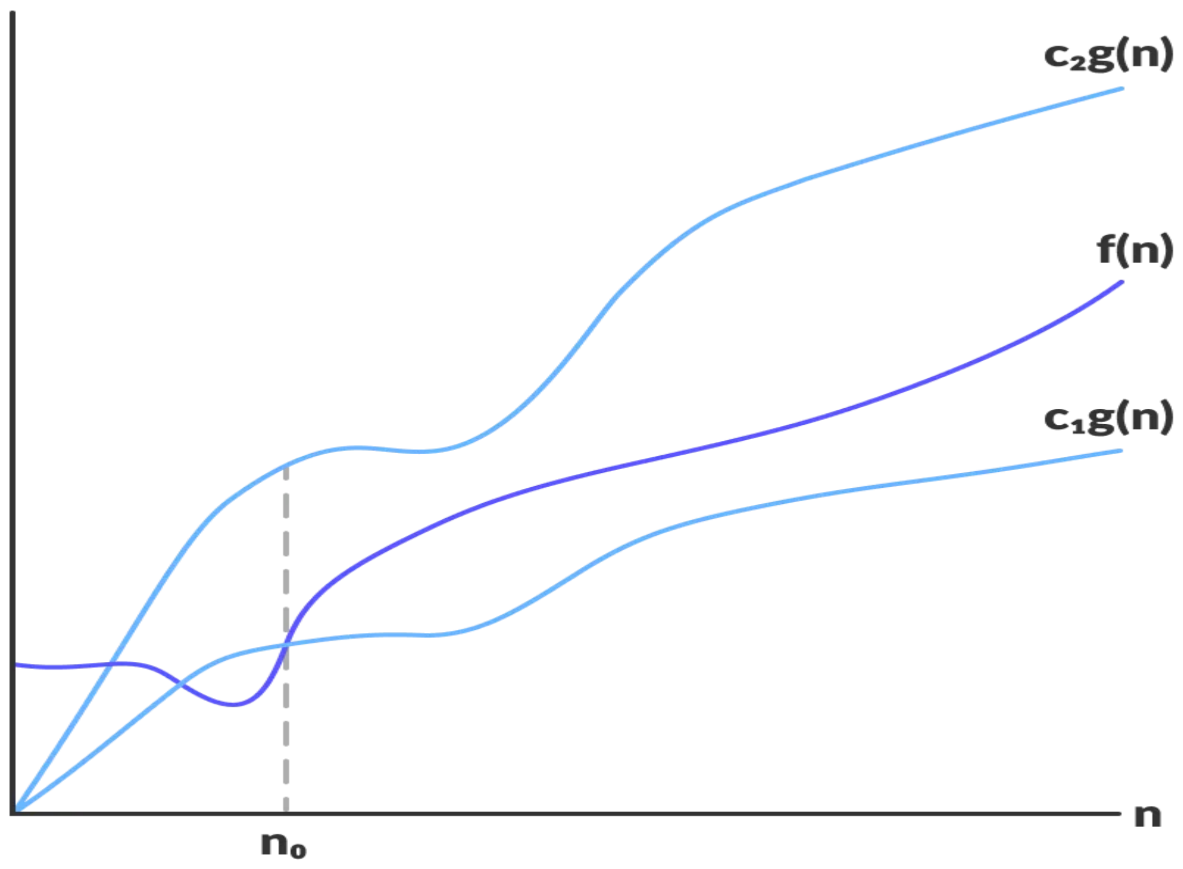
2. **Omega Notation (Ω)**

- **Definition**: Omega notation represents the **lower bound** of an algorithm's running time. It describes the best-case time complexity, or the minimum time an algorithm can take for large inputs.
- **Use Case**: Used to establish a minimum performance level, showing that the algorithm will not run faster than this bound in the best case.
- **Example**: If an algorithm’s time complexity is Ω(n), it means that, at a minimum, the algorithm will require time proportional to n as the input size n grows.



3. **Theta Notation (Θ)**

- **Definition**: Theta notation provides a **tight bound** for an algorithm's running time, meaning it describes both the upper and lower bounds. It's the most precise measure of an algorithm's efficiency, as it defines the exact asymptotic behavior.
- **Use Case**: Used when the upper and lower bounds are the same, allowing you to accurately predict the algorithm's performance.
- **Example**: If an algorithm's time complexity is Θ(nlogn), it means the execution time will always grow at a rate proportional to nlogn for large inputs, regardless of best or worst case.



$f(n) = \Theta(g(n))$

Notation	Definition	Explanation
Big O (O)	$f(n) \le C * g(n)$ for all $n \ge n_0$	Describes the upper bound of the algorithm's running time. Used most of the time.
Ω (Omega)	$f(n) \ge C * g(n)$ for all $n \ge n_0$	Describes the lower bound of the algorithm's running time . Used less
θ (Theta)	$C_1 * g(n) \le f(n) \le C_2 * g(n)$ for $n \ge n_0$	Describes both the upper and lower bounds of the algorithm's **running time**. Also used a lot more and preferred over Big O if we can find an exact bound.