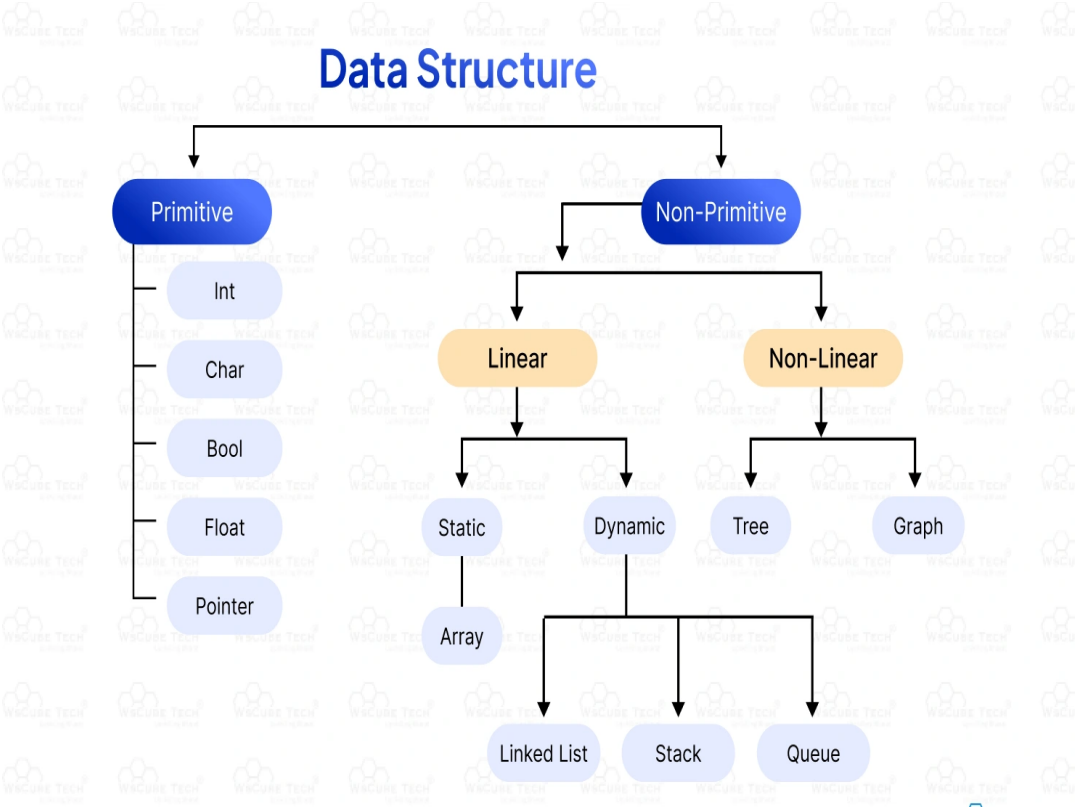
What are data structures ?

- A data structure is a storage which is used to store and organize the data on the computer.
- Data structures are the fundamental building blocks of any computer programming.
- Classification of data structures:
    - Primitive data structures: Primitive data structures are the basic fundamental data types which are there in a programming language and are used to represent the simple data values.

  Eg. int, float, char, boolean, string

    - Non-primitive data structures: Non-primitive data structures are made using primitive data structures and store data multiple values.

  Eg. array, stack, queue, linked list, trees, graphs

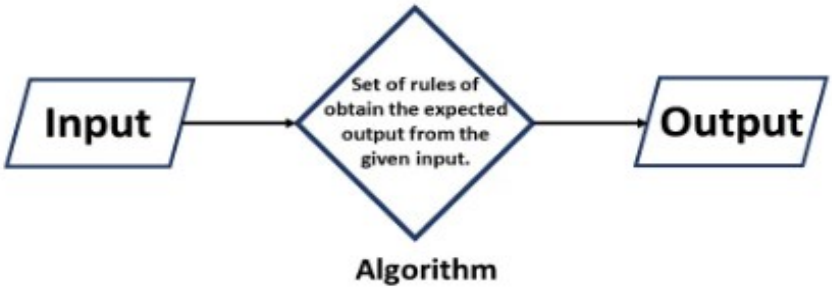


Basic data structure operations

- Insertion: Insertion means to add an element in the given data structure.
- Deletion: Deletion means to delete an element in the given data structure.
- Traversing: Traversing a data structure means to visit the element stored in it.
- Searching: Searching means to find a particular element in the given data-structure.

What is algorithm ?

A set of finite rules or instructions to be followed in calculations or other problem-solving operations



Performance analysis of an algorithm

- Performance analysis of an algorithm means evaluating performance of algorithms and programs.
- Efficiency is measured in terms of time and space.
    - Time complexity: It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

- Space complexity: The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm.

- **Asymptotic analysis:** Asymptotic analysis describes the running time (or space) of an algorithm in terms of input size n, as n approaches infinity. It focuses on the growth rate of an algorithm's complexity.
- **Amortized analysis:** Amortized analysis gives the average running time per operation over a sequence of operations, even if some operations are expensive occasionally.

- **Asymptotic notations**

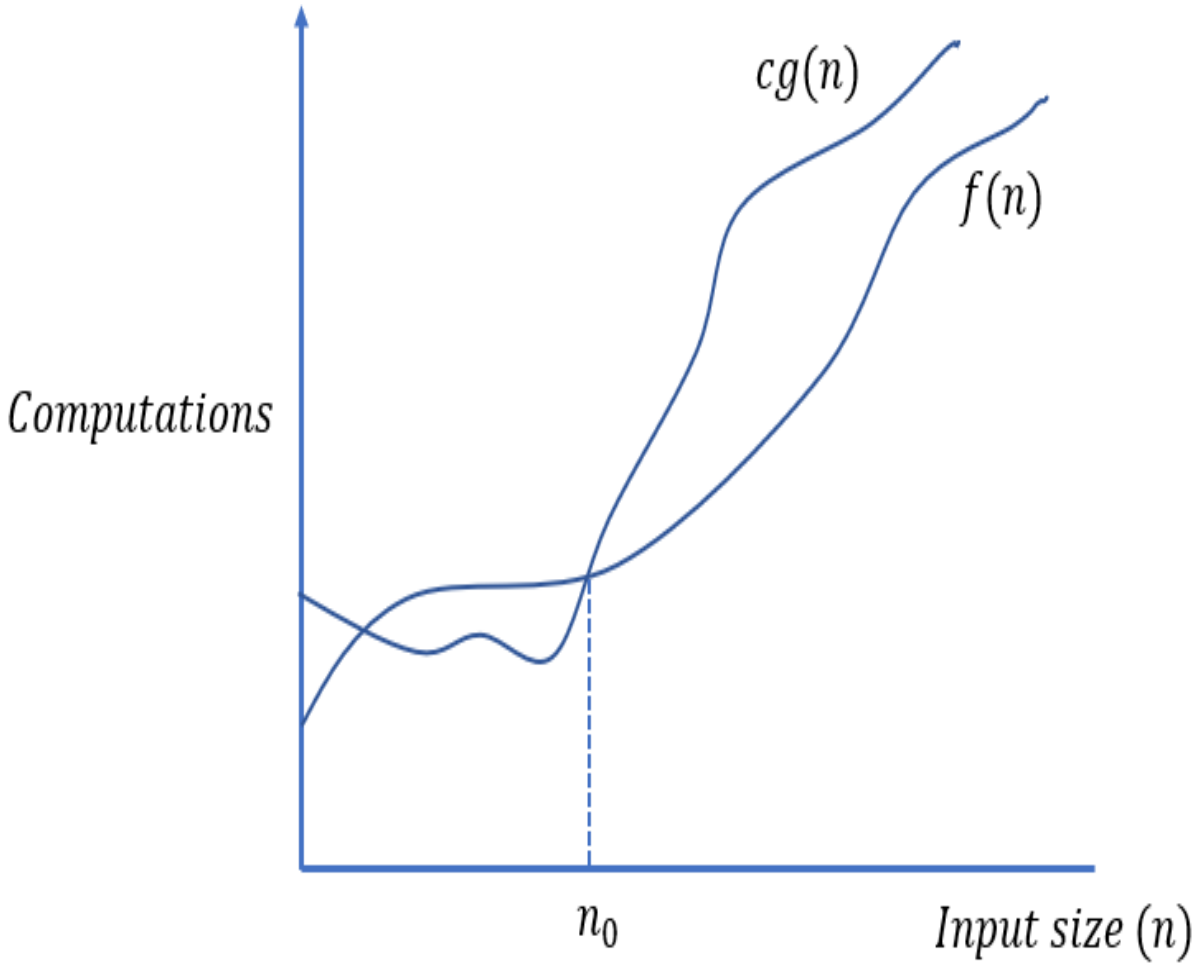
  Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.

- **There are mainly three asymptotic notations:**
  - **Big-O Notation (O-notation)**
  - **Omega Notation (Ω-notation)**
- **Theta Notation (Θ-notation)**

Big-O notation (O-notation)

- Big-O notation represents the upper bound of the running time of an algorithm.
- The maximum time required by an algorithm or the worst-case time complexity. Therefore, it gives the worst-case complexity of an algorithm.
- It is the most widely used notation for Asymptotic analysis.
- Mathematical Representation of Big-O Notation:

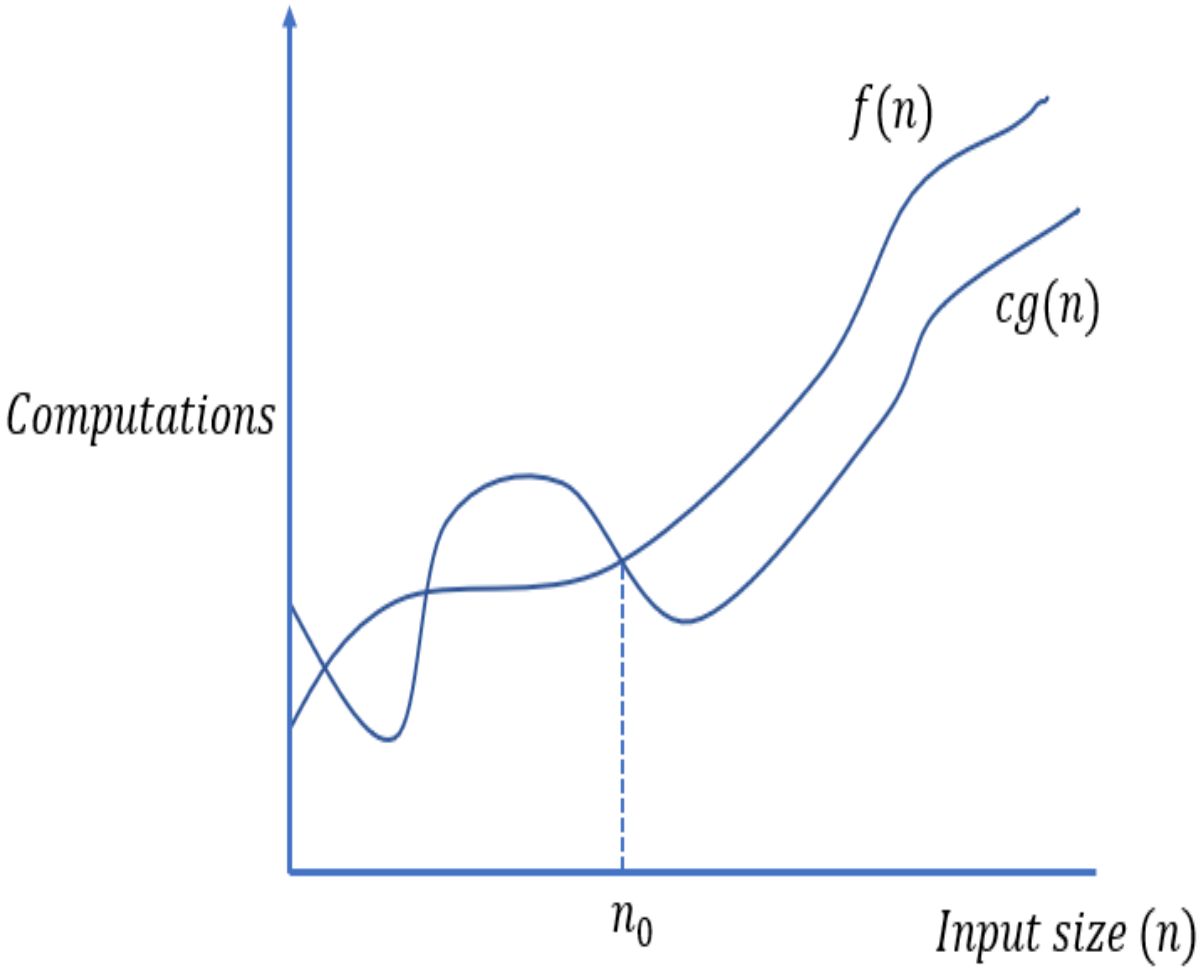
$O(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n0 \}$



Omega notation (Ω-notation)

- Omega notation represents the **lower bound** of the running time of an algorithm.
- It is defined as the condition that allows an algorithm to complete statement execution in the **shortest amount of time**. Thus, it provides the best case complexity of an algorithm.
- Mathematical Representation of Omega notation :

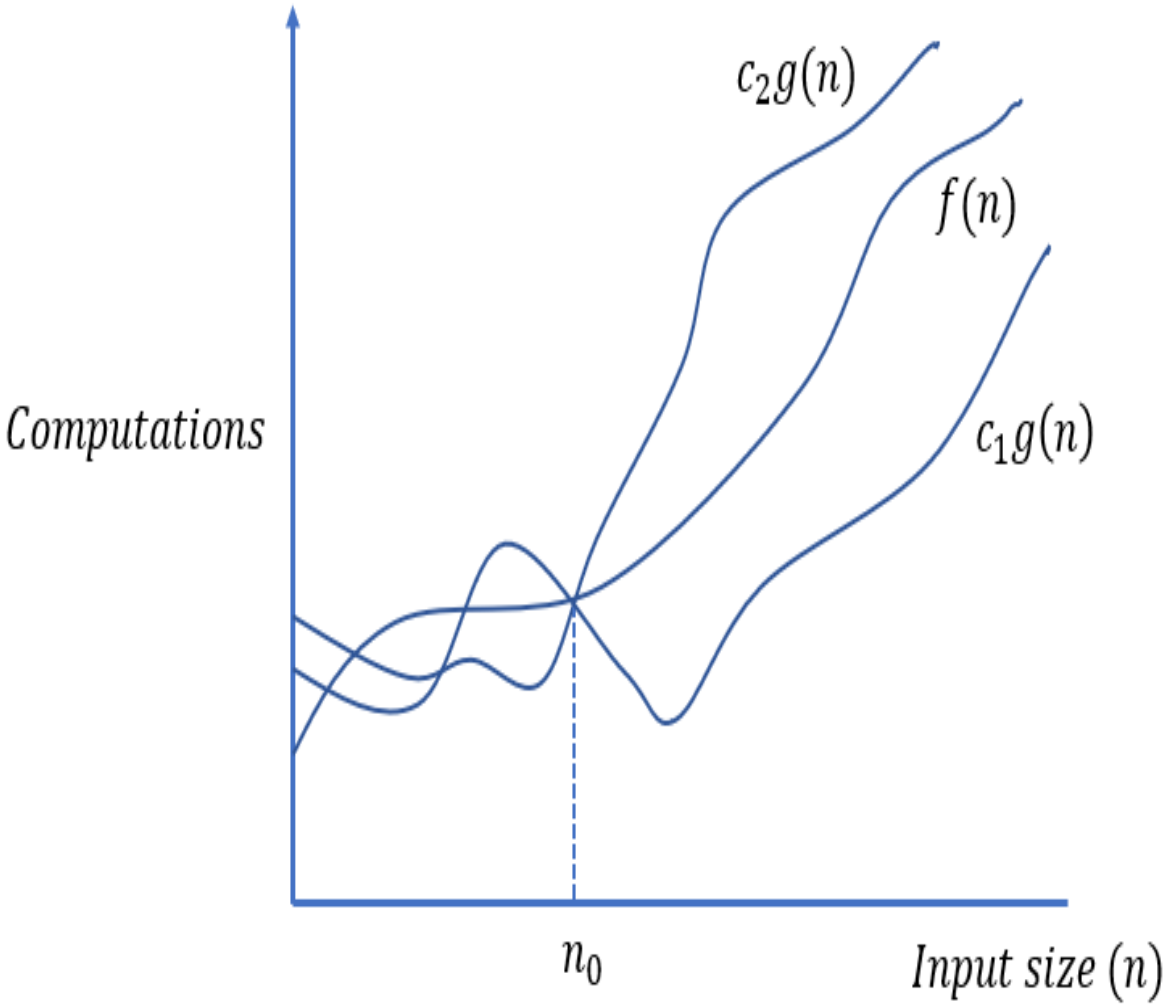
Ω(g(n)) = { f(n): there exist positive constants c and n0 such that **0 ≤ cg(n) ≤ f(n)** for all n ≥ n0 }



Theta notation (Θ-notation):

- Theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.
- Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.
- Mathematical Representation of Theta notation:

Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that **0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n)** for all n ≥ n0}



Types of time complexities

- Constant Time (O(1)):

An algorithm is said to have constant time complexity if the execution time is constant, i.e., it does not depend on the size of the input.

Example: Accessing an element in an array, performing a simple arithmetic operation.

- Logarithmic Time (O(log n)):

Logarithmic time complexity occurs when the execution time of an algorithm increases logarithmically with the size of the input.

Example: Binary search in a sorted array.

- Linear Time (O(n)):

An algorithm has linear time complexity if its execution time is directly proportional to the size of the input data.

Example: Linear search in an unsorted array.

- **Quasilinear Time (O(n log n)):**

Quasilinear time complexity is a combination of linear and logarithmic time complexities.

Example: Many efficient sorting algorithms like Merge Sort and Heap Sort.

- **Quadratic Time ($O(n^2)$):**

Quadratic time complexity occurs when the execution time of an algorithm is proportional to the square of the size of the input.

Example: Bubble Sort, Insertion Sort.

- **Exponential Time ($O(2^n)$):**

Exponential time complexity arises when the execution time doubles with each additional input.

Example: Recursive algorithms without memorization that solve problems like the Tower of Hanoi.

- **Factorial Time (O(n!)):**

Factorial time complexity occurs when the execution time is a factorial of the size of the input.

Example: Algorithms that generate all permutations of a set.

Time complexity calculation

1. **Loops:** Number of iterations of a loop. (Same as the size of the loop).

Example:

```
for(i = 1; i <= n; i++) // loop will iterate for n number of times
    printf("ByteXL"); // constant time 'c'
```

Total Time = Constant c × n => cn => **O(n)**

2. Nested Loop: Analysis of nested loops will be from inside out. Total time complexity is the product of the size of all the loops.

Example:

```
for(i=1;i<=n;i++) // outer loop iterates for n times
{
    for(j=1;j<=n;j++) // inner loop iterates for n times
    {
        printf("ByteXL"); // constant time c
    }
}
```

Total Time = c × n × n => cn^2 => **O(n^2)**

3. Consecutive Statements: While analyzing the consecutive statements, multiple complexities may be found in a single program. The highest of all of them will be considered as the Time complexity of the entire program.

Example:

```
y = y + 1; // constant time
for(i = 0; i <= n; i++) // iterates n times
    k = k + 3; // constant time
for(j = 0; j <= n; j++) // outer loop iterates n times
{
    for(k = 2; k <= n; k++) // inner loop iterates n times
```

```
    {
        l = l + 1; // constant time
    }
}
```

Total Time = c + c1n + c2n^2 => **O(n^2)**

4. If -Else Statement:

Example:

```
if (1)
{
    k = k + 1; // constant part
}
else //(constant + constant) * n
{
    for(i = 0; i <= n; i++) // loop iterates n number of times
    {
        j = j * 2; // constant time
    }
}
```

Total Time = c0 + (c1 + c2) * n => **O(n)**

5. Logarithmic Complexities: An algorithm is O(logn) if it takes half the time of constant time.

Example:

```
for(i = 1; i <= n;)

i = i * 2;
```

Total Time = O(logn)