

## What is an Algorithm?

An algorithm is a well-defined, step-by-step procedure or set of instructions for solving a specific problem or performing a particular task. Algorithms serve as the foundational building blocks of computer programs and are used in various fields, including mathematics, computer science, engineering, and everyday problem-solving.

Algorithms are used in various applications, including sorting and searching data, performing mathematical calculations, solving complex problems, and controlling computer hardware and software. They are a fundamental concept in computer science and play a crucial role in the development of software and systems across many domains. The study and analysis of algorithms are essential for designing efficient and effective solutions to real-world problems.

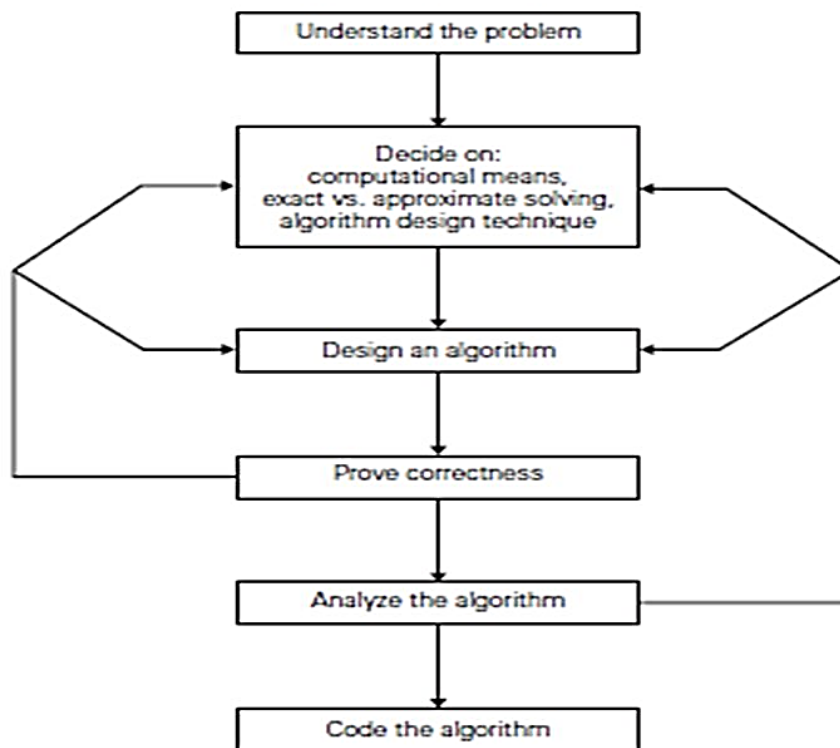
### Key characteristics of algorithms include:

- **Input:** Algorithms take some input or data as their starting point. This input can be in various forms, such as numbers, text, images, or other data structures.
- **Processing:** Algorithms consist of a finite sequence of steps, each of which performs a specific operation on the input data. These steps are executed in a specific order to achieve the desired outcome.
- **Output:** After processing the input data according to the defined steps, algorithms produce an output or result. This output could be a solution to a problem, a transformed dataset, or some other meaningful information.
- **Determinism:** Algorithms are deterministic, meaning that given the same input and conditions, they will always produce the same output. This predictability is a fundamental property of algorithms.
- **Termination:** Algorithms are designed to terminate or finish after a finite number of steps. They should not run indefinitely.
- **Correctness:** An algorithm is considered correct if it produces the desired output for all valid inputs and follows the specified rules and constraints.
- **Efficiency:** Efficient algorithms are designed to perform their tasks with reasonable resource utilization, such as time and memory. Optimization is often a concern when designing algorithms.

### Fundamentals of algorithms problem solving

1. **Understand the Problem:** Before you start coding, make sure you thoroughly understand the problem. Read and re-read the problem statement to clarify any ambiguities.
2. **Break Down the Problem:** If the problem is complex, break it down into smaller, more manageable subproblems. This simplifies the overall task and helps you focus on solving individual parts.
3. **Gather Requirements:** Identify the requirements and constraints of the problem. What are the input parameters, expected outputs, and any limitations on time or space?
4. **Choose the Right Data Structures:** Select appropriate data structures to represent the problem's data efficiently. This choice can significantly impact the algorithm's performance.
5. **Algorithm Design:** Choose an algorithmic approach that fits the problem's nature:
  - a. Brute Force: Try all possible solutions.
  - b. Greedy Algorithms: Make locally optimal choices at each step.
  - c. Divide and Conquer: Break the problem into smaller subproblems.
  - d. Dynamic Programming: Solve subproblems and store results to avoid redundant calculations.

- e. **Backtracking:** Systematically explore possible solutions and backtrack when necessary.
6. **Pseudocode:** Write pseudocode or a high-level outline of your algorithm's logic before diving into coding. Pseudocode helps you plan your approach.
7. **Plan for Edge Cases:** Consider edge cases and corner cases. Think about how your algorithm should handle unexpected or extreme inputs.
8. **Code Incrementally:** Write code incrementally, starting with the simplest parts and gradually adding complexity. This allows you to test and debug more easily.
9. **Test Thoroughly:** Test your code with various test cases, including both typical and edge cases, to ensure it produces the correct output.
10. **Debugging:** Debug any errors or unexpected behavior in your code. Use debugging tools, print statements, or debuggers as needed.
11. **Analyze Time and Space Complexity:** Analyze the time complexity (how the runtime scales with input size) and space complexity (memory usage) of your algorithm. Big O notation is commonly used for this purpose.



### Fundamentals of analysis of algorithm and efficiency

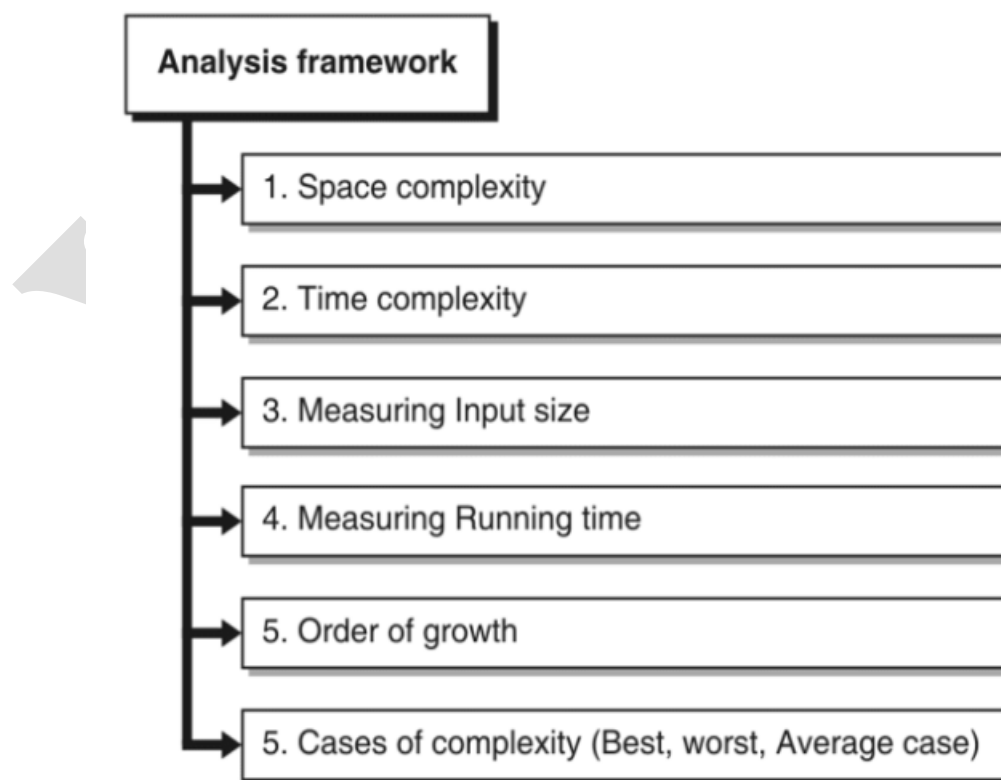
Algorithm analysis involves assessing the behavior of an algorithm in terms of time and space requirements as a function of input size. Some of key factors involves

1. **Time Complexity:** Time complexity measures the amount of time an algorithm takes to complete its task as a function of the input size. It is usually expressed using Big O notation, such as  $O(n)$ ,  $O(\log n)$ ,  $O(n^2)$ , etc.
2. **Space Complexity:** Space complexity measures the amount of memory (or space) an algorithm uses as a function of the input size. Like time complexity, it is expressed using Big O notation.
3. **Best, Worst, and Average Case:** Algorithms may exhibit different performance characteristics depending on the nature of the input data. Analysis considers the best-case, worst-case, and average-case scenarios.

4. **Asymptotic Analysis:** Asymptotic analysis focuses on how an algorithm behaves as the input size approaches infinity, rather than on exact runtime measurements.
5. **Comparative Analysis:** Comparative analysis involves comparing the performance of multiple algorithms for the same problem to determine which one is more efficient.
6. **Efficiency Goals:** Efficiency goals vary based on the problem and context. Common goals include minimizing time complexity, space complexity, or both.
7. **Trade-offs:** In some cases, there are trade-offs between time and space efficiency. Analysing these trade-offs helps in making informed algorithm design choices.
8. **Optimization Techniques:** Techniques like memorization, dynamic programming, and data structure selection are used to optimize algorithms for better efficiency.
9. **Amortized Analysis:** Amortized analysis considers the average time or space usage over a sequence of operations. It helps evaluate the efficiency of data structures like arrays and lists.
10. **Practical Considerations:** Practical efficiency considerations may involve real-world factors like hardware constraints, input distribution, and the need for real-time processing.
11. **Algorithmic Problem Solving:** Efficient problem solving involves choosing or designing algorithms that strike a balance between time and space efficiency while satisfying problem-specific constraints.

Efficiency analysis is a critical aspect of algorithm design and is essential for building scalable and performant software systems. It helps in making informed decisions about which algorithms to use for various tasks and understanding their behavior as input sizes grow.

#### Analysis Framework:



1. **Space complexity:** Problem-solving using a computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve a given problem is called the space complexity of the algorithm

The space complexity of the algorithm is controlled by two components:

- a. Fixed-size components: It includes the programming part whose memory requirement does not alter program execution. For example,
  - i. Instructions
  - ii. Variables
  - iii. Constants
  - iv. Arrays
- b. Variable size components: It includes the part of the program which whose size depends on the problem being solved. For example,
  - i. Size of loop
  - ii. Stack required to handle a recursive call
  - iii. Dynamic data structures like a linked list

To indicate the space complexity of the issue with input size  $n$ , we use the notation  $S(n)$ . The word  $n$  refers to the size of the input or the size of the problem.

2. **Time Complexity:** The valid algorithm executes in a finite period of time. The time complexity of the algorithm refers to how long it takes the algorithm to solve a given problem. In algorithm analysis, time complexity is very useful measure.

**Example: Addition of two scalar variables:**

```
Algorithm ADD_SCALAR(A, B)

// Description : Perform arithmetic addition of two numbers
// Input : Two scalar variables A and B
// Output : variable C, which holds the addition of A and B
C ← A + B
return C
```

The sum of two scalar numbers requires one addition operation. Thus, the time complexity of this algorithm is constant, so  $T(n) = O(1)$

The addition of two scalar numbers requires one extra memory location to hold the result. Thus, the space complexity of this algorithm is constant, hence  $S(n) = O(1)$ .

3. **Measuring Input Size:** The algorithm's execution time is mostly determined by the size of the input. For larger input, the algorithm runs longer. As a result, the algorithm's complexity is always assessed in terms of input size. The complexity is independent of hardware architecture, available memory, CPU speed, and so forth.

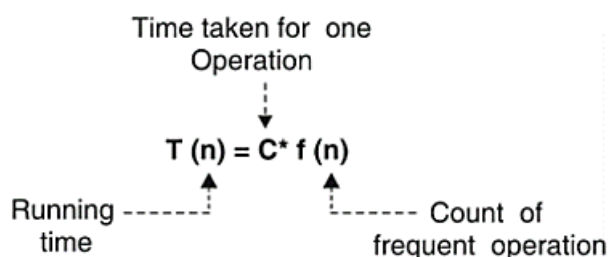
A specific algorithm requires prior knowledge of the size of the input to solve the problem. For example, in order to perform a binary search, we must first know the size of the array. Some applications, such as counting the number of nodes in a linked list, do not require it in advance.

4. **Measuring Running Time:** The running time of an algorithm is primarily determined by the amount of the input and, as a result, the number of frequent operations executed by the algorithm. The algorithm's running time is not measured in terms of real time consumed by the algorithm in seconds or minutes, but rather as a function of a number of primitive/frequent operations. Primitive operation is another major factor of efficiency analysis of algorithm.

Problem	Primitive operation
Sorting	Comparison
Searching	Comparison
Matrix multiplication	Multiplication
Adding two arrays	Addition
Count number of nodes in a linked list	Addition
Insert element in the tree	Comparison
Find the greatest common divisor	Division

Running time is commonly indicated by  $T$  and is calculated by summing the frequent operations ( $n$ ).  $T(n)$  is the time it takes the algorithm to solve a problem of size  $n$ .

An algorithm's approximate running time is calculated as follows:



5. **Order of Growth:** The efficiency of the algorithm is expressed in term of input size  $n$ . The relationship between input size and performance of the algorithm is called an order of growth.

The order of increase reflects how rapidly the time required by the algorithm rises in relation to the size of the input.

The algorithm may run a number of steps in the sequence of  $\log n$ ,  $n$ ,  $n^2$ ,  $n^3$ , or anything else for input size  $n$ .

Efficiency class	Order of growth rate	Example
Constant	1	Delete the first node from a linked list, Remove the largest element from the max heap, Add two numbers
Logarithmic	$\log n$	Binary search Insert/delete element from binary search tree
Linear	$n$	Linear search, Insert node at the end of the linked list, Find minimum / maximum element from an array
$n \log n$	$n \log n$	Merge sort, Binary search, Quicksort, Heap Sort
Quadratic	$n^2$	Selection Sort, Bubble Sort, Input 2D array, Find maximum element form 2D matrix
Cubic	$n^3$	Matrix Multiplication
Exponential	$2^n$	Finding the power set of any set, Find the optimal solution for the Knapsack problem, Solve TSP using dynamic programming
Factorial	$n!$	Generating permutations of a given set, Solve TSP problem using brute force approach

**Efficiency classes are sorted as :**

$O(1) < O(\log^k n) < O(\log(\log n)) < O(\log n) < O((\log n)^k) < O(n) < O(n \log^k n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$ .

These are the most often used classes; however, there are numerous additional classes that reflect intermediate growth order. Algorithms with a running time at the top of the list are thought to be better. Algorithms with exponential or factorial running times are unsuitable for practical use.

6. **Cases of Complexity:** The algorithm's running time for the same data structures and the same data may differ.

With different permutations of input data, the number of steps necessary to solve the problem, and hence the running time, may change. If we want to sort two values and the input is  $\langle 1, 2 \rangle$ , we don't need to swap. However, if the input sequence is  $\langle 2, 1 \rangle$ , swapping is required. The organization of data is equally critical. On the basis of this assumption, we will look at three examples of complexity analysis.

Let  $T(n)$  denote the amount of time necessary to solve a problem of size  $n$ . We will look at all three examples of linear search complexity. The linear search method is explained below:

```

Algorithm LINEAR_SEARCH(A, Key)
// Description : Search element 'Key' in array A
// Input : Array A of size n, and the element to be searched i.e. Key
// Output : Status: Success / failure

flag ← 1
for i ← 1 to n do
  if A[i] == key then
    flag ← 1
    break
  end
end

if flag == 1 then
  print "Key is found on location i"
else
  print "Key is not present"
end
  
```

**Best case analysis:**

If an algorithm takes minimum time to solve the problem for a given set of input, it is called best case time complexity. Assume we wish to remove a node from a linked list of  $n$  nodes. First, we must locate the node to be removed from the list. If the node is at the very first place, only one comparison is necessary.

We need to update one link and then release the node. This procedure is independent of list size, which implies it may be executed in a fixed amount of time regardless of input size. As a result, the best-case running time for removing the first node from the list is constant,  $T(n) = O(1)$ .

**Worst-case analysis:**

If an algorithm takes maximum time to solve the problem for a given set of input, it is called worst-case time complexity. Worst case is the special case of interest for the mathematicians. If the

element to be removed is at the last position in linked list, we must compare each node value to a key element. In this situation, we must make  $n$  comparisons.

The time required for deletion and link updating is constant. As the size of the list grows, so does the number of comparisons. The number of comparisons is proportional to the size of the problem (i.e. list size). When an element is either at the end or not there at all, the algorithm does  $n$  comparisons.

As a result, the worst-case running time for this issue is linear, i.e.  $T(n) = O(n)$ .

### Average case analysis:

Input sequence which is neither best nor worst is called the average case. Average case analysis represents the general behavior of an algorithm

With a random series of input data, the average case occurs. In the preceding example, an average case occurs when the node to be removed is neither at the initial nor at the last place.

The program searches 50% of the list on average. It performs  $n/2$  comparisons roughly. Once again, it appears to be linear growth. As a result, the average-case running time of the algorithm is linear, i.e.  $T(n) = O(n/2) = O(n)$ .

Average case analysis is more than just taking the best and worst cases and averaging them. The average case is determined by the data distribution.

### Measuring input size of an algorithm:

Measuring the input size of an algorithm is a crucial step in analysing its efficiency. The input size helps you understand how the algorithm's performance scales as the problem size increases. The way you measure input size depends on the nature of the problem and the type of data the algorithm processes. Here are some common methods for measuring input size:

1. **Numeric Data:** If your algorithm operates on numeric data, such as sorting a list of numbers, the input size is typically the number of elements in the data structure. For example, if you're sorting an array of integers, the input size would be the length of the array.
2. **Strings:** When dealing with text data, such as searching for a substring in a text document, the input size is often measured by the length of the input string. For instance, if you're searching for a word in a text document, the input size is the number of characters in the document.
3. **Graphs:** In algorithms that work with graphs, like finding the shortest path in a graph, the input size is determined by the number of nodes and edges in the graph. You might express it as " $n$ " for the number of nodes and " $m$ " for the number of edges.
4. **Matrices:** For problems involving matrices or multi-dimensional arrays, the input size is usually defined by the dimensions of the matrix. For instance, if you're performing matrix multiplication, the input size is the number of rows and columns in the matrices.
5. **Database Queries:** In database-related algorithms, the input size might be the number of rows or records that need to be queried or processed.
6. **Images and Multimedia:** When working with images, audio, or other multimedia data, the input size can be more complex. It might involve factors like resolution, duration, or file size.
7. **Data Structures:** If your algorithm operates on data structures, like a binary tree or a linked list, the input size could be defined by the number of elements in the data structure.
8. **Combinations:** Some algorithms may require multiple parameters to define the input size. For example, in a scheduling algorithm, the input size might depend on the number of tasks, their durations, and resource constraints.

9. **Problem-Specific Metrics:** In certain cases, the input size may be defined by a problem-specific metric that best represents the problem's complexity. This may require domain-specific knowledge.

When expressing the time and space complexity of your algorithm using Big O notation, the input size is a critical factor. It helps you understand how the algorithm's efficiency scales with different problem sizes and allows for meaningful comparisons between different algorithms.

### UNITS FOR MEASURING RUNTIME OF AN ALGORITHM

When measuring the running time of an algorithm in the context of Design and Analysis of Algorithms (DAA), the most common unit used is "Big O Notation" ( $O(f(n))$ ). Big O notation represents the upper bound of an algorithm's time complexity in relation to the size of the input data. It provides a high-level understanding of how the algorithm's performance scales with input size. Here are some common examples of Big O notations used in DAA:

1.  **$O(1)$  - Constant Time:** The algorithm's running time does not depend on the input size. Examples include simple mathematical operations and array indexing.
2.  **$O(\log n)$  - Logarithmic Time:** The running time grows logarithmically with the input size. Examples include binary search and certain efficient algorithms on balanced trees.
3.  **$O(n)$  - Linear Time:** The running time is directly proportional to the input size. Examples include linear search and simple array traversals.
4.  **$O(n \log n)$  - Linearithmic Time:** Common in efficient sorting algorithms like Merge Sort and Heap Sort.
5.  **$O(n^2)$  - Quadratic Time:** The running time is proportional to the square of the input size. Examples include bubble sort and certain nested loop algorithms.
6.  **$O(n^k)$  - Polynomial Time:** The running time is a polynomial function of the input size. ( $k$ ) represents the degree of the polynomial. For example,  $O(n^3)$  for cubic time complexity.
7.  **$O(2^n)$  - Exponential Time:** The running time grows exponentially with the input size. Algorithms with recursive backtracking strategies often exhibit exponential time complexity.
8.  **$O(n!)$  - Factorial Time:** The running time grows factorially with the input size. Algorithms that generate permutations and combinations often have factorial time complexity.

Big O notation provides a standardized way to describe the efficiency of algorithms without getting into specific hardware or implementation details. It helps algorithm designers and analysts compare different algorithms and choose the most appropriate one for a specific problem based on their time complexity analysis.



## Pseudo-Code

Programmers are Often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are not computer programs but are more structured than usual prose. They also facilitate the high-level analysis of a data structure or algorithm; we call these descriptions pseudo-code.

### An Example of Pseudo-Code

The array-maximum problem is the simple problem of finding the maximum element in an array  $A$  storing  $n$  integers. To solve this problem, we can use an algorithm called `arrayMax`, which scans through the elements of  $A$  using a **for** loop.

The pseudo-code description of algorithm `arrayMax` is shown in Algorithm 1.2.

**Algorithm** `arrayMax`( $A, n$ ):

*Input:* An array  $A$  storing  $n \geq 1$  integers.

*Output:* The maximum element in  $A$ .

```

currentMax ← A[0]
for i ← 1 to n – 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax

```

Algorithm 1.2: Algorithm `arrayMax`.

Note that the pseudo-code is more compact than an equivalent actual software code fragment would be. In addition, the pseudo-code is easier to read and understand.

## What Is Pseudo-Code?

Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the pseudo-code language, however, because of its reliance on natural language at the same time, to help achieve clarity, pseudo-code mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as C, C++, and Java. These constructs include the following

- **Expressions:** We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign ( $\leftarrow$ ) as the assignment operator in assignment statements (equivalent to the  $=$  operator in C, C++, and Java), and we use the equal sign ( $=$ ) as the equality relation in Boolean expressions (equivalent to the  $==$  relation in C, C++, and Java).
- **Method declarations:** Algorithm name(param1, param2, ...) declares a new method “name” and its parameters.
- **Decision structures:** if condition then true-actions [else false-actions]. We use indentation to indicate what actions should be included in the true-actions and false-actions.
- **While-loops:** while condition do actions. We use indentation to indicate what actions should be included in the loop actions.
- **Repeat-loops:** repeat actions until condition. We use indentation to indicate what actions should be included in the loop actions.
- **For-loops:** for variable-increment-definition do actions. We use indentation to indicate what actions should be included among the loop actions.
- **Array indexing:**  $A[i]$  represents the  $i$ th cell in the array  $A$ . The cells of an  $n$ -celled array  $A$  are indexed from  $A[0]$  to  $A[n - 1]$  (consistent with C, C++, and Java).
- **Method calls:** object.method(args) (object is optional if it is understood).

- **Method returns:** return value. This operation returns the value specified to the method that called this one.

In conclusion, effective pseudo-code should be clear, precise, and easy to understand. It must focus on presenting the logical flow and core idea of the algorithm without getting lost in programming language syntax or unnecessary technical details. At the same time, it should include all essential steps to avoid ambiguity. Achieving this balance between clarity and completeness ensures that pseudo-code serves as a strong foundation for designing, analysing, and implementing algorithms.

### Asymptotic Notation

When analysing simple algorithms like arrayMax and recursiveMax, we often go into detailed step-by-step evaluation of their running time. However, performing such detailed analysis for complex algorithms would be cumbersome and impractical.

#### Key Idea

- Each step in a pseudo-code description or each statement in a high-level programming language corresponds to a small number of primitive operations.
- These primitive operations generally do not depend on the input size.
- Therefore, instead of counting every individual primitive operation exactly, we estimate the running time up to a constant factor.

#### Simplified Analysis

To make analysis manageable:

- We count the number of steps in the pseudo-code.
- We count the number of statements executed in the high-level implementation.
- We ignore constant factors because they do not significantly affect performance for large input sizes.

#### Purpose of Asymptotic Notation

Asymptotic notation provides a mathematical way to:

- Characterize the main factors that affect an algorithm's running time.
- Focus on how running time grows as input size increases.
- Avoid unnecessary details about exact counts of primitive operations.

Asymptotic notation helps us analyze algorithms efficiently by concentrating on growth rate rather than exact execution time. It allows us to compare algorithms based on scalability and performance for large inputs, making it a powerful tool in algorithm analysis.

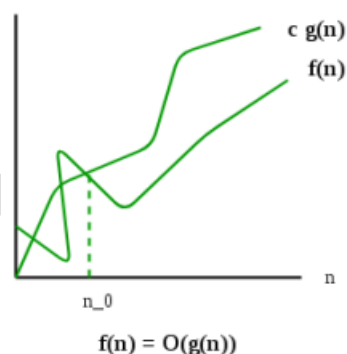
## The Big O Notation:

The Big O notation defines an upper bound of an algorithm; it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is  $O(n^2)$ . Note that  $O(n^2)$  also covers linear time. The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times, we easily find an upper bound by simply looking at the algorithm. For a given function  $g(n)$ , we denote by  $O(g(n))$  the set of functions.

### Interpretation:

- The growth of  $f(n)$  does not exceed  $g(n)$  asymptotically.
- It is commonly used to describe the **worst-case time complexity**.

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0, \text{ such that } 0 \leq f(n) \leq c \times g(n) \text{ for all } n \geq n_0 \}$



Example:

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

$$2n + 3 \leq 10n \quad \forall n \geq 1$$

$$\text{Here, } c=10, n_0=1, g(n)=n$$

$$\Rightarrow f(n) = O(n)$$

$$\text{Also, } 2n + 3 \leq 2n + 3n$$

$$2n + 3 \leq 5n \quad \forall n \geq 1$$

$$\text{And, } 2n + 3 \leq 2n^2 + 3n^2$$

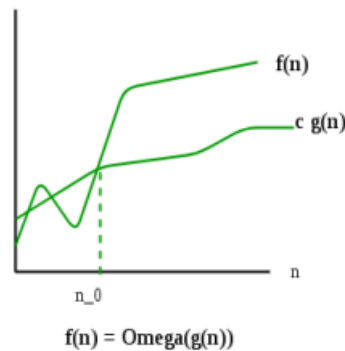
$$2n + 3 \leq 5n^2$$

$$\Rightarrow f(n) = O(n^2)$$

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n^n)$$

### The Omega ( $\Omega$ ) notation:

Just as Big O notation provides an asymptotic upper bound on a function,  $\Omega$  notation provides an asymptotic lower bound.  $\Omega$  notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previously, the best-case performance of an algorithm is generally not useful, the omega notation is the least used notation among all three. For a given function  $g(n)$ , we denote by  $\Omega(g(n))$  the set of functions.



$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}.$

Let us consider the same insertion sort example here. The time complexity of insertion sort can be written as  $\Omega(n)$ , but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

#### Example:

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

$$2n + 3 \geq n \quad \forall n \geq 1$$

$$\text{Here, } c=1, n_0=1, g(n)=n$$

$$\Rightarrow f(n) = \Omega(n)$$

$$\text{Also, } f(n) = \Omega(\log n)$$

$$f(n) = \Omega(\sqrt{n})$$

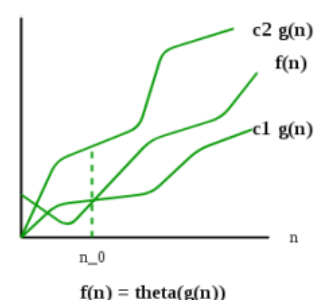
#### Interpretation:

- The algorithm will take **at least this much time** for large inputs.
- Often associated with **best-case complexity**.

### The Theta ( $\Theta$ ) notation:

The theta notation bounds a function from above and below, so it defines the exact asymptotic behaviour. A simple way to get theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$



Dropping lower order terms is always fine because there will always be a  $n_0$  after which  $\Theta(n^3)$  has higher values than  $\Theta(n^2)$  irrespective of the constants involved.

### Interpretation:

- The function grows **exactly at the same rate** as  $g(n)$  asymptotically.
- It provides the most accurate description of complexity.

For a given function  $g(n)$ , we denote  $\Theta(g(n))$  as the following set of functions.

$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \text{ for all } n \geq n_0\}$

The above definition means, if  $f(n)$  is theta of  $g(n)$ , then the value  $f(n)$  is always between  $c_1 \times g(n)$  and  $c_2 \times g(n)$  for large values of  $n (n \geq n_0)$ . The definition of theta also requires that  $f(n)$  must be non-negative for values of  $n$  greater than  $n_0$ .

### Example:

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

$$1 \cdot n \leq 2n + 3 \leq 5n \quad \forall n \geq 1$$

$$\text{Here, } c_1 = 1, c_2 = 5, n_0 = 1, g(n) = n$$

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

Feature	Big O	Big $\Omega$	Big $\Theta$
Type of Bound	Upper Bound	Lower Bound	Tight Bound
Represents	Worst Case	Best Case	Exact Asymptotic Growth
Inequality	( $f(n) \leq c \cdot g(n)$ )	( $f(n) \geq c \cdot g(n)$ )	( $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ )
Precision	Less Precise	Less Precise	Most Precise
Usage	Performance guarantee	Performance limitation	Exact characterization