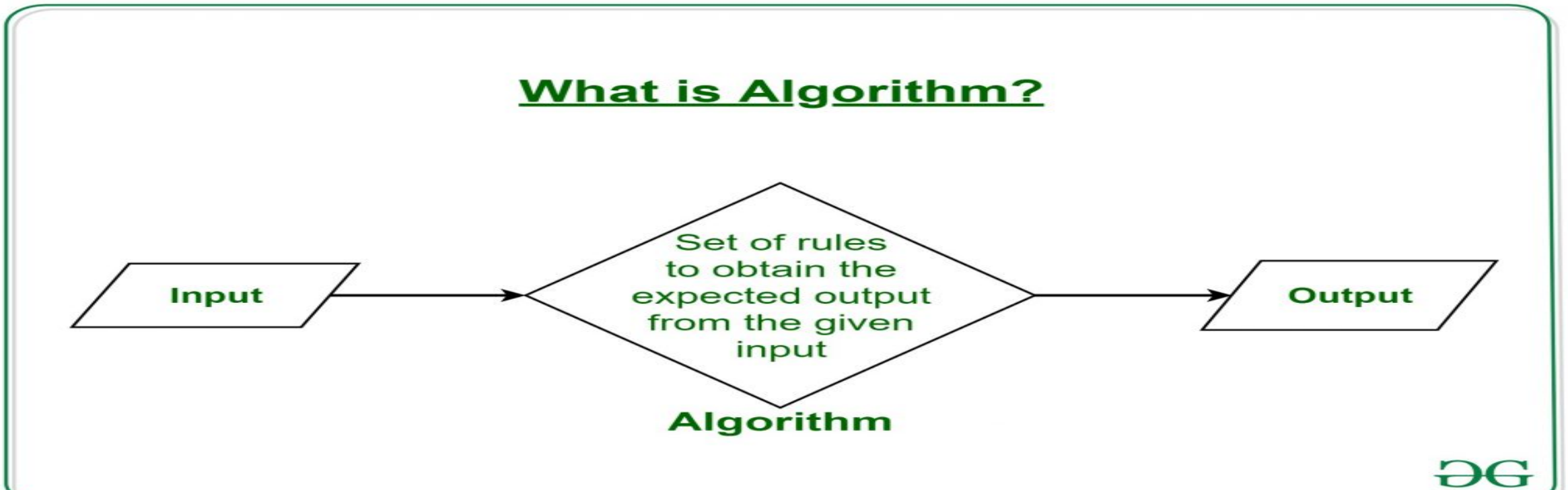
UNIT – I

Definition of Algorithm

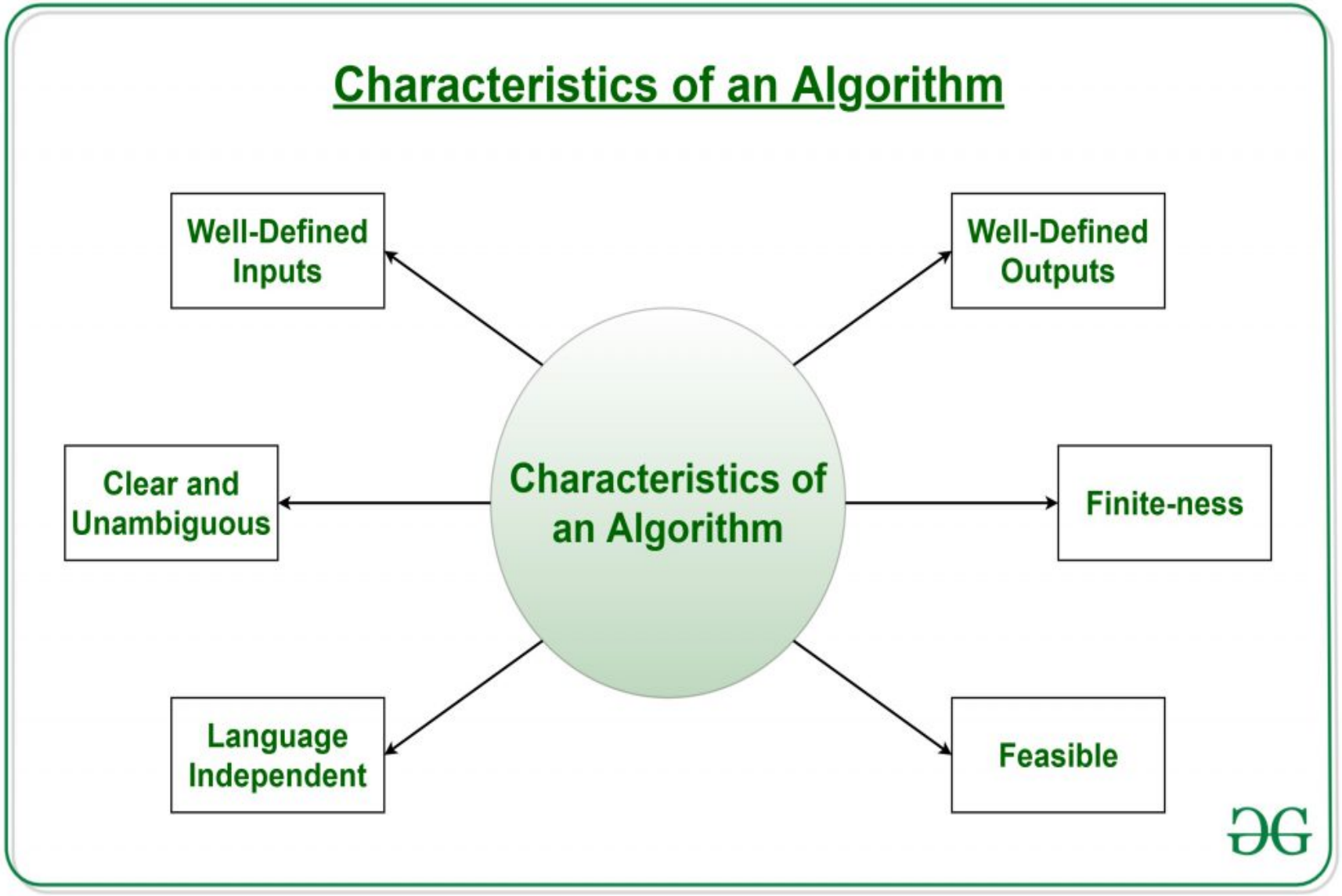
The word **Algorithm** means ” *A set of finite rules or instructions to be followed in calculations or other problem-solving operations* ”

Or

” *A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations*”.



What are the Characteristics of an Algorithm?



- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Notions of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Fundamental of Algorithms To write an algorithm, the following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** is to be expected when the problem is solved.
5. The **solution** to this problem is within the given constraints.

The main aim of designing an algorithm is to provide a optimal solution for a problem. Not all problems must have similar type of solutions; an optimal solution for one problem may not be optimal for another. Therefore, we must adopt various strategies to provide feasible solutions for all types of problems.

Types of Algorithms:

There are several types of algorithms available. Some important algorithms are:

1. Brute Force Algorithm:

It is the simplest approach to a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm:

A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm:

The backtracking algorithm builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm:

Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm:

Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm:

Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm:

This algorithm breaks a problem into sub-problems, solves a single sub-problem, and merges the solutions to get the final solution. It consists of the following three steps:

- Divide
- Solve
- Combine

8. Greedy Algorithm:

In this type of algorithm, the solution is built part by part. The solution for the next part is built based on the immediate benefit of the next part. The one solution that gives the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm:

This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm:

In the randomized algorithm, we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

The Role of Algorithms in Computing :

Algorithms play a crucial role in computing by providing a set of instructions for a computer to perform a specific task. They are used to solve problems and carry out tasks in computer systems, such as sorting data, searching for information, image processing, and much more.

An algorithm defines the steps necessary to produce the desired outcome, and the computer follows the instructions to complete the task efficiently and accurately.

The development of efficient algorithms is a central area of computer science and has significant impacts in various fields, from cryptography and finance to machine learning and robotics.

Algorithms are widely used in various industrial areas to improve efficiency, accuracy, and decision-making. Some of the key applications include:

1.**Manufacturing:** Algorithms are used to optimize production processes and supply chain management, reducing waste and increasing efficiency.

2.Finance: Algorithms are used to analyze financial data and make predictions, enabling traders and investors to make informed decisions.

3.Healthcare: Algorithms are used to process and analyze medical images, assist in diagnosing diseases, and optimize treatment plans.4Retail: Algorithms are used for customer relationship management, personalized product recommendations, and pricing optimization.

4.Transportation: Algorithms are used to optimize routes for delivery and transportation, reducing fuel consumption and increasing delivery speed.

5.Energy: Algorithms are used to optimize energy generation, distribution, and consumption, reducing waste and increasing efficiency.

6.Security: Algorithms are used to detect and prevent security threats, such as hacking, fraud, and cyber-attacks.

Analysis algorithms :

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.

The term **"analysis of algorithms"** was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.

Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

Time complexity :

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it take for each statement to complete.

Space Complexity :

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

Fundamentals of the Analysis of Algorithm Efficiency :

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.

In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space.

Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Asymptotic notation and Basic Efficiency Classes:

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we **evaluate the performance of an algorithm in terms of input size** (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size.

Asymptotic notation is a way to describe the running time or space complexity of an algorithm based on the input size.

It is commonly used in complexity analysis to describe how an algorithm performs as the size of the input grows.

There are mainly three asymptotic notations:

- *Big-O Notation (O-notation)*
- *Omega Notation (Ω-notation)*
- *Theta Notation (Θ-notation)*

Big Oh, O: Asymptotic Upper Bound

Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.

This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

.It is the most widely used notation for Asymptotic analysis.
.It specifies the upper bound of a function.
.The maximum time required by an algorithm or the worst-case time complexity.
.It returns the highest possible output value(big-O) for a given input.
.Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. I**t returns the highest possible output value (big-O)for a given input.**

The execution time serves as an upper bound on the algorithm’s time complexity. It is represented as shown below:

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

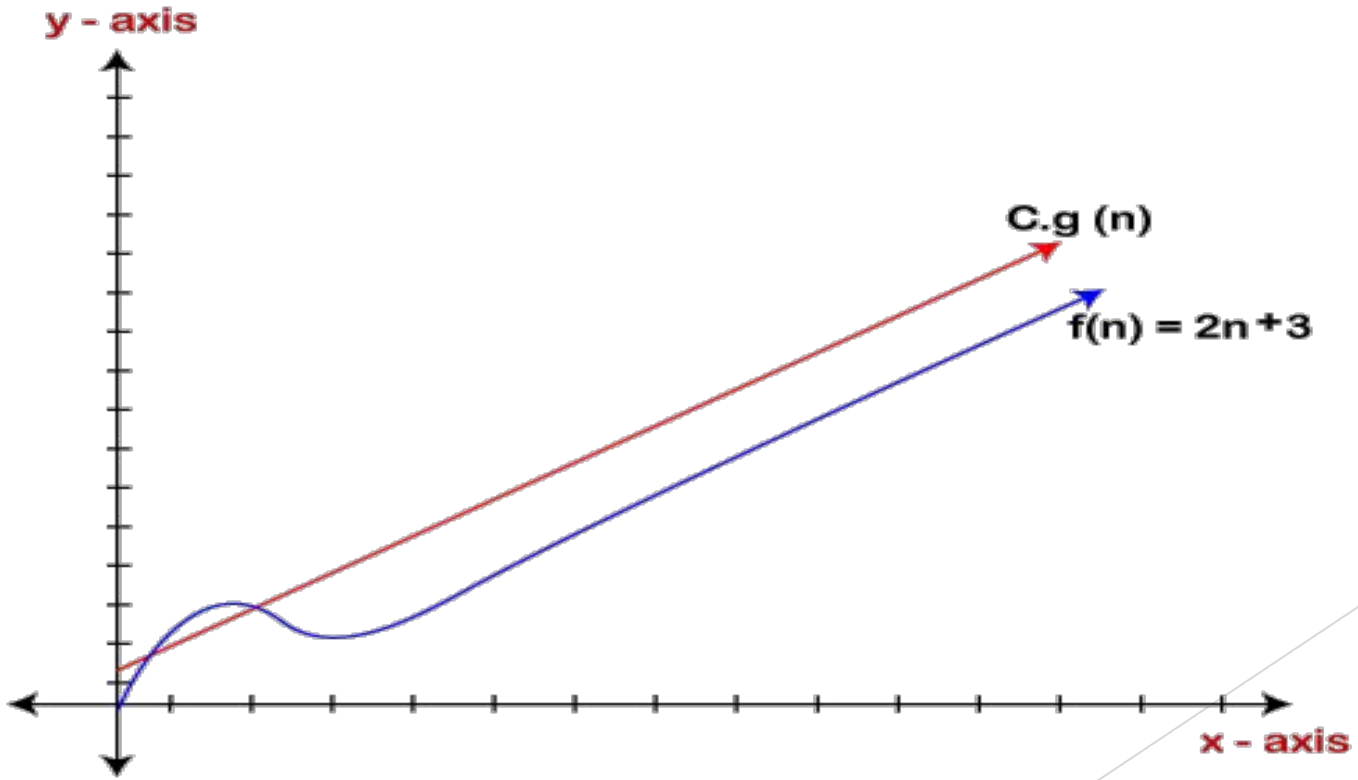
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity.

It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

For example, Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is $O(n^2)$.
Note: $O(n^2)$ also covers linear time.

Big Omega, Ω: Asymptotic Lower Bound

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.

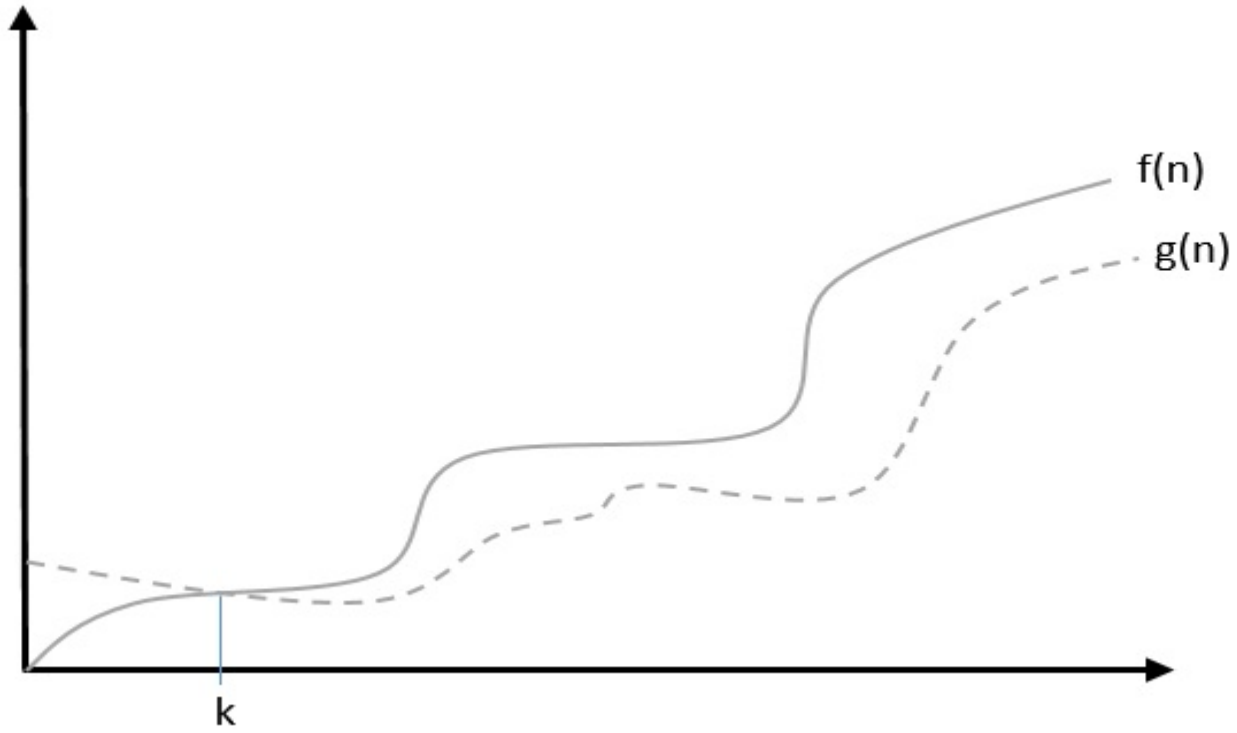
It measures the **best case time complexity** or the best amount of time an algorithm can possibly take to complete.

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

We say that $f(n)=\Omega(g(n))$ when there exists constant **c** that $f(n) \geqslant c.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer.

It means function ***g*** is a lower bound for function ***f*** ; after a certain value of ***n, f*** will never go below ***g***.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.



Example :

Let us consider a given function, f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n3 , f(n)≥4.g(n) for all the values of n>0

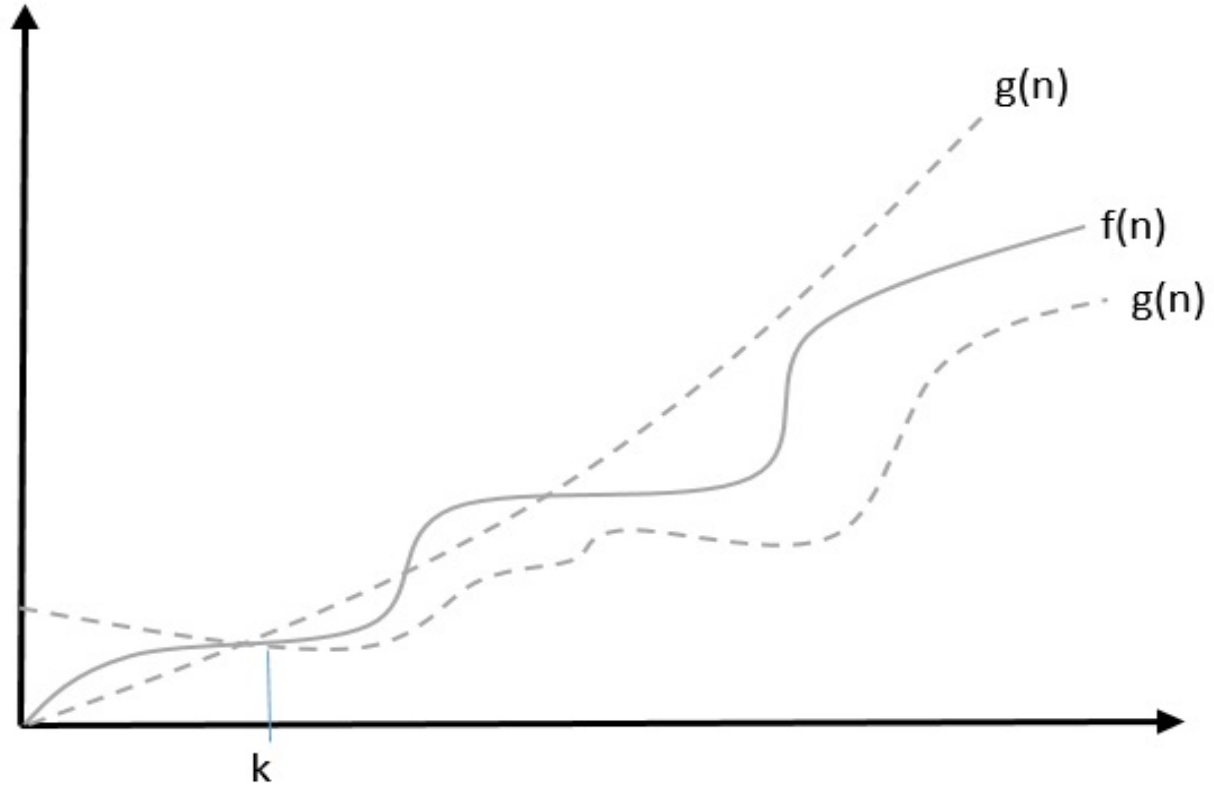
Hence, the complexity of ***f(n)*** can be represented as Ω(g(n)) i.e. Ω(n3)

Theta, θ: Asymptotic Tight Bound

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.

We say that f(n)=θ(g(n) when there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that c1.g(n)⩽f(n)⩽c2.g(n) for all sufficiently large value of **n**.
Here **n** is a positive integer.
This means function **g** is a tight bound for function **f**.



Example :

Let us consider a given function, f(n)=4.n3+10.n2+5.n+1

Considering g(n)=n3, 4.g(n)⩽f(n)⩽5.g(n) for all the large values of **n**.

Hence, the complexity of ***f(n)*** can be represented as θ(g(n)), i.e. θ(n3).

Algorithms Design Techniques

What is an algorithm?

An Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input.

The algorithms can be classified in various ways. They are:

Implementation Method

Design Method

Design Approaches

Other Classifications

The classification of algorithms is important for several reasons:

Organization: Algorithms can be very complex and by classifying them, it becomes easier to organize, understand, and compare different algorithms.

Problem Solving: Different problems require different algorithms, and by having a classification, it can help identify the best algorithm for a particular problem.

Performance Comparison: By classifying algorithms, it is possible to compare their performance in terms of time and space complexity, making it easier to choose the best algorithm for a particular use case.

Reusability: By classifying algorithms, it becomes easier to re-use existing algorithms for similar problems, thereby reducing development time and improving efficiency.

Research: Classifying algorithms is essential for research and development in computer science, as it helps to identify new algorithms and improve existing ones.

Classification by Implementation Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

Recursion or Iteration: A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.

Example: The Tower of Hanoi is implemented in a recursive fashion while Stock Span problem is implemented iteratively.

Exact or Approximate: Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.

Example: For NP-Hard Problems, approximation algorithms are used. Sorting algorithms are the exact algorithms.

Serial or Parallel or Distributed Algorithms:

In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.

Classification by Design Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

Greedy Method: In the greedy method, at each step, a decision is made to choose the local optimum, without thinking about the future consequences.

Example: Fractional Knapsack, Activity Selection.

Divide and Conquer: The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.

Example: Merge sort, Quicksort.

Dynamic Programming: The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.

Example: 0-1 Knapsack, subset-sum problem.

Linear Programming: In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs.

Example: Maximum flow of Directed Graph

Reduction(Transform and Conquer): In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.

Example: Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called transform and conquer.

Backtracking:

This technique is very useful in solving combinatorial problems that have a single unique solution. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage.

This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution.

Example: N-queen problem, maize problem.

Branch and Bound:

This technique is very useful in solving combinatorial optimization problem that have multiple solutions and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution.

Example: Job sequencing, Travelling salesman problem.

Classification by Design Approaches : There are two approaches for designing an algorithm. these approaches include

Top-Down Approach :

Bottom-up approach

Top-Down Approach:

In the top-down approach, a large problem is divided into small sub-problem. and keep repeating the process of decomposing problems until the complex problem is solved.

Bottom-up approach:

The bottom-up approach is also known as the reverse of top-down approaches.

In approach different, part of a complex program is solved using a programming language and then this is combined into a complete program.

Top-Down Approach:

Breaking down a complex problem into smaller, more manageable sub-problems and solving each sub-problem individually.

Designing a system starting from the highest level of abstraction and moving towards the lower levels.

Bottom-Up Approach:

Building a system by starting with the individual components and gradually integrating them to form a larger system.

Solving sub-problems first and then using the solutions to build up to a solution of a larger problem.

Note: Both approaches have their own advantages and disadvantages and the choice between them often depends on the specific problem being solved.

Other Classifications: Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

Randomized Algorithms: Algorithms that make random choices for faster solutions are known as randomized algorithms.

Example: Randomized Quicksort Algorithm.

Classification by complexity: Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.

Example: Some algorithms take O(n), while some take exponential time.

Classification by Research Area: In CS each field has its own problems and needs efficient algorithms.

Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.

Branch and Bound Enumeration and Backtracking: These are mostly used in Artificial Intelligence.

END