What is Algorithm | Introduction to Algorithms

What is an Algorithm? Algorithm Basics

*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".*

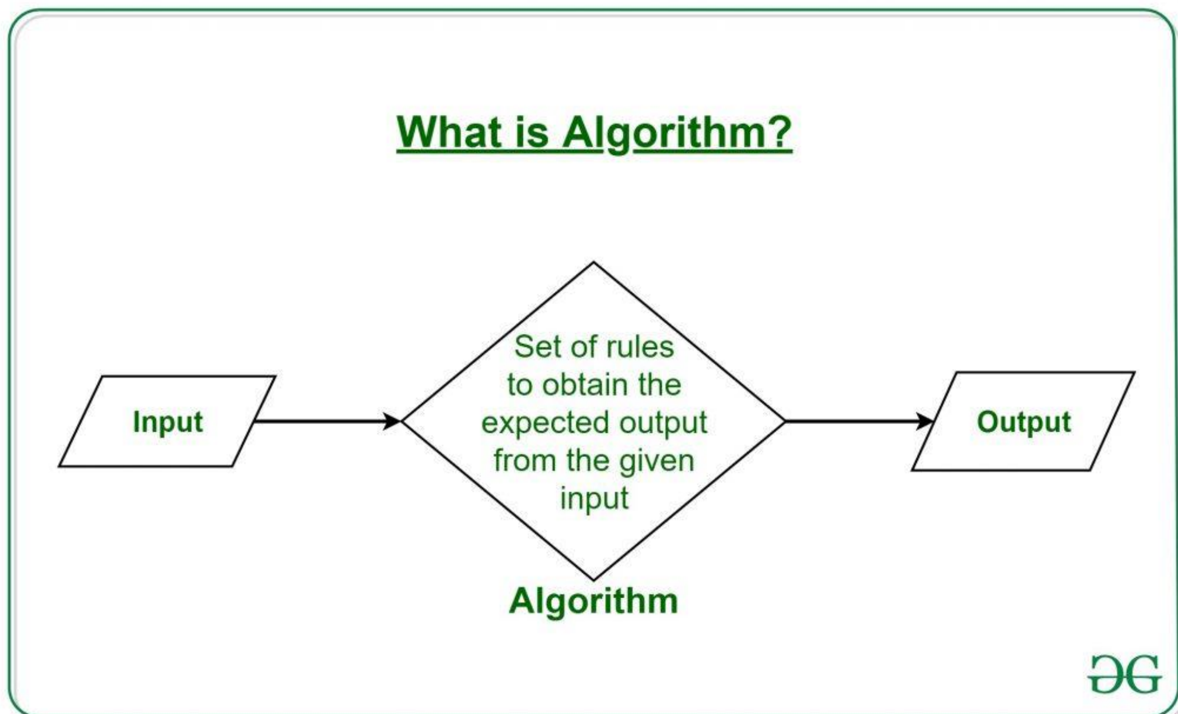
Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.

Use of the Algorithms:-

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

Computer Science: Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
Mathematics: Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
Operations Research: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
Artificial Intelligence: Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
Data Science: Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.
These are just a few examples of the many applications of algorithms. The use of algorithms is continually expanding as new technologies and fields emerge, making it a vital component of modern society.

Algorithms can be simple and complex depending on what you want to achieve.



It can be understood by taking the example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and executes them one by one, in the given sequence. The result thus obtained is the new dish is cooked perfectly. Every time you use your phone, computer, laptop, or calculator you are using Algorithms. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

What is the need for algorithms:

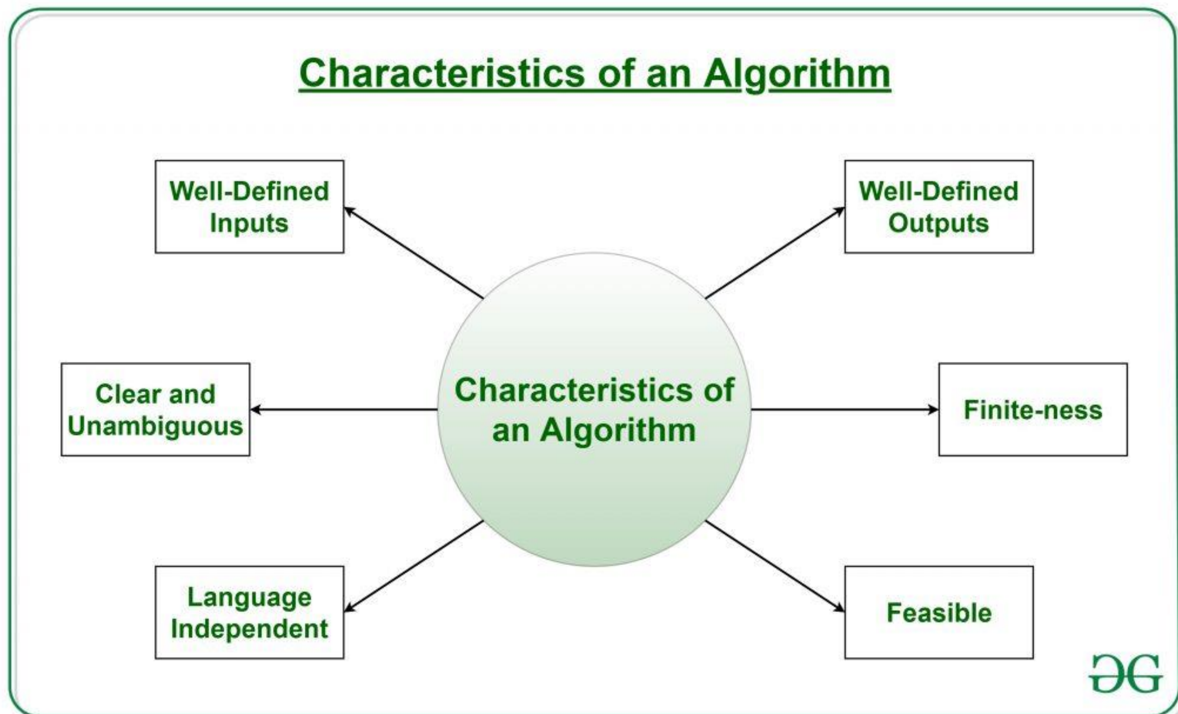
1.Algorithms are necessary for solving complex problems efficiently and effectively.

2.They help to automate processes and make them more reliable, faster, and easier to perform.

3.Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.

4.They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

What are the Characteristics of an Algorithm?



As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithms. In order for some instructions to be an algorithm, it must have the following characteristics:

- **Clear and Unambiguous**: The algorithm should be clear and 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.
- **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.

Properties 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.

Types of Algorithms:

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

1. Brute Force Algorithm: It is the simplest approach for 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 basically 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 and 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 together 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 of the next part is built based on the immediate benefit of the next part. The one solution giving 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.
To learn more about the types of algorithms refer to the article about "**Types of Algorithms**".

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms**(imp)**.

How to Design an Algorithm?

In order to write an algorithm, the following things are needed as a pre-requisite:

The **problem** that is to be solved by this algorithm i.e. clear problem definition.
The **constraints** of the problem must be considered while solving the problem.
The **input** to be taken to solve the problem.
The **output** to be expected when the problem is solved.

The **solution** to this problem, is within the given constraints.

Linear Search vs Binary Search

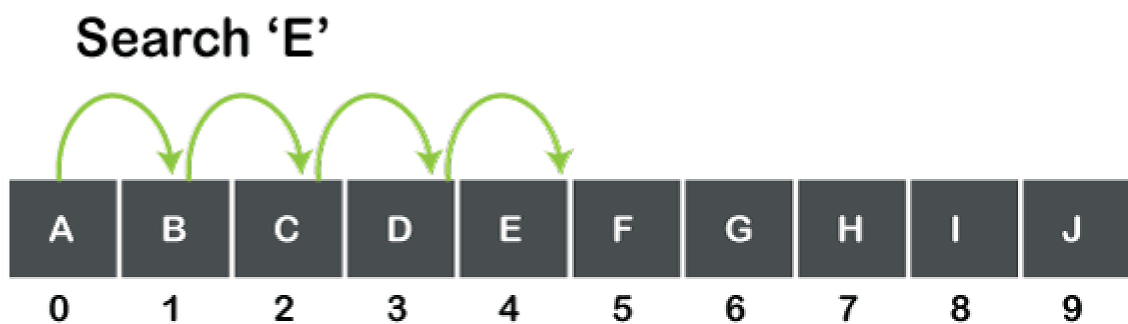
Before understanding the differences between the linear and binary search, we should first know the linear search and binary search separately.

What is a linear search?

A linear search is also known as a sequential search that simply scans each element at a time. Suppose we want to search an element in an array or list; we simply calculate its length and do not jump at any item.

Let's consider a simple example.

Suppose we have an array of 10 elements as shown in the below figure:



The above figure shows an array of character type having 10 values. If we want to search 'E', then the searching begins from the 0th element and scans each element until the element, i.e., 'E' is not found. We cannot directly jump from the 0th element to the 4th element, i.e., each element is scanned one by one till the element is not found.

Complexity of Linear search

As linear search scans each element one by one until the element is not found. If the number of elements increases, the number of elements to be scanned is also increased. We can say that the ***time taken to search the elements is proportional to the number of elements***. Therefore, the worst-case complexity is O(n)

What is a Binary search?

A binary search is a search in which the middle element is calculated to check whether it is smaller or larger than the element which is to be searched. The main

advantage of using binary search is that it does not scan each element in the list. Instead of scanning each element, it performs the searching to the half of the list. So, the binary search takes less time to search an element as compared to a linear search.

The one ***pre-requisite of binary search*** is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

There are three cases used in the binary search:

Case 1: data<a[mid] then left = mid+1.

Case 2: data>a[mid] then right=mid-1

Case 3: data = a[mid] // element is found

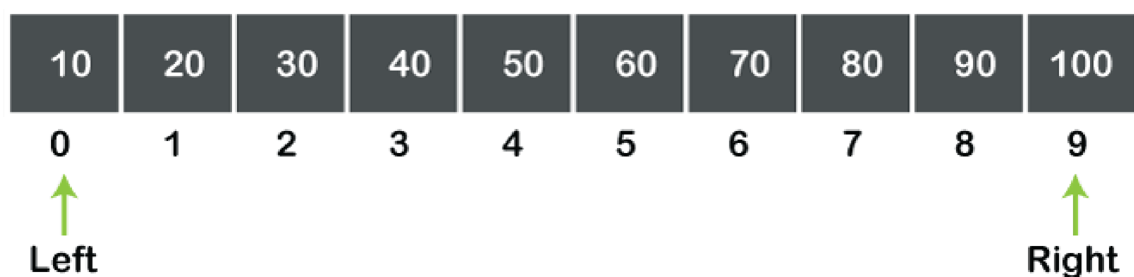
In the above case, '**a**' is the name of the array, **mid** is the index of the element calculated recursively, **data** is the element that is to be searched, **left** denotes the left element of the array and **right** denotes the element that occur on the right side of the array.

Let's understand the working of binary search through an example.

Suppose we have an array of 10 size which is indexed from 0 to 9 as shown in the below figure:

We want to search for 70 element from the above array.

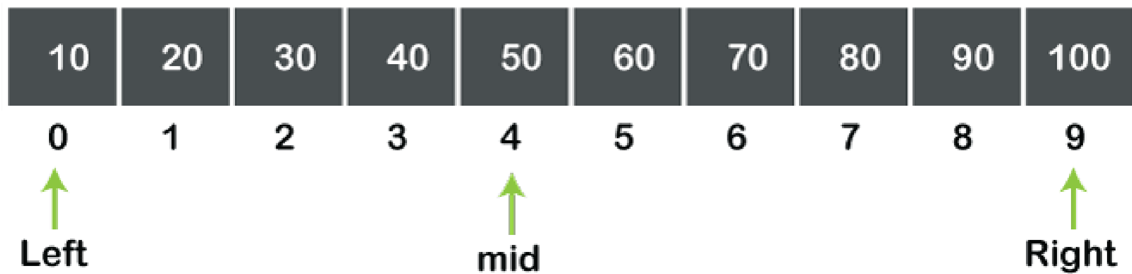
Step 1: First, we calculate the middle element of an array. We consider two variables, i.e., left and right. Initially, left =0 and right=9 as shown in the below figure:



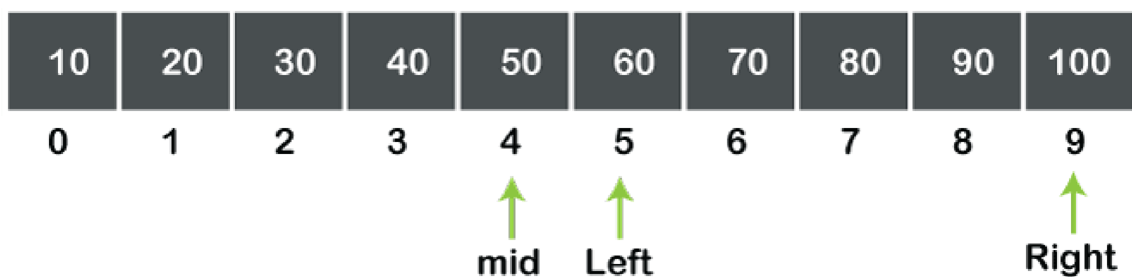
The middle element value can be calculated as:

$$mid = \frac{left + right}{2}$$

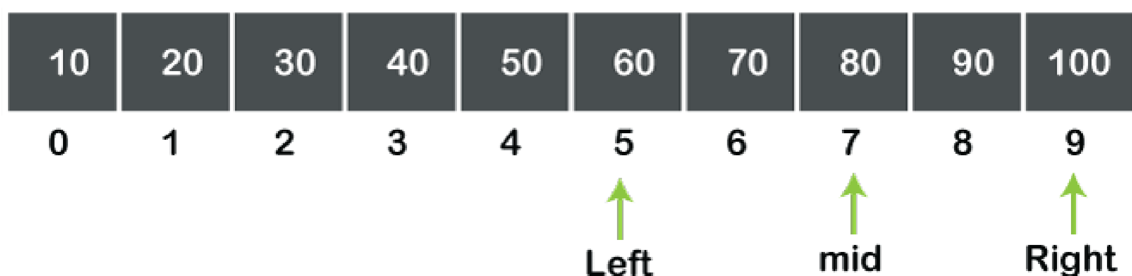
Therefore, mid = 4 and a[mid] = 50. The element to be searched is 70, so a[mid] is not equal to data. The case 2 is satisfied, i.e., data>a[mid].



Step 2: As data>a[mid], so the value of left is incremented by mid+1, i.e., left=mid+1. The value of mid is 4, so the value of left becomes 5. Now, we have got a subarray as shown in the below figure:

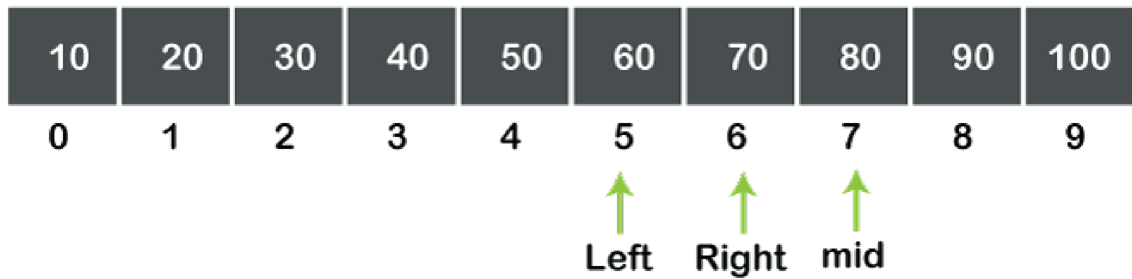


Now again, the mid-value is calculated by using the above formula, and the value of mid becomes 7. Now, the mid can be represented as:



In the above figure, we can observe that a[mid]>data, so again, the value of mid will be calculated in the next step.

Step 3: As a[mid]>data, the value of right is decremented by mid-1. The value of mid is 7, so the value of right becomes 6. The array can be represented as:



The value of mid will be calculated again. The values of left and right are 5 and 6, respectively. Therefore, the value of mid is 5. Now the mid can be represented in an array as shown below:

In the above figure, we can observe that a[mid]<data.

Step 4: As a[mid]<data, the left value is incremented by mid+1. The value of mid is 5, so the value of left becomes 6.

Now the value of mid is calculated again by using the formula which we have already discussed. The values of left and right are 6 and 6 respectively, so the value of mid becomes 6 as shown in the below figure:

We can observe in the above figure that a[mid]=data. Therefore, the search is completed, and the element is found successfully.

Differences between Linear search and Binary search

The following are the differences between linear search and binary search:

Description

Linear search is a search that finds an element in the list by searching the element sequentially until the element is found in the list. On the other hand, a binary search is a search that finds the middle element in the list recursively until the middle element is matched with a searched element.

Working of both the searches

The linear search starts searching from the first element and scans one element at a time without jumping to the next element. On the other hand, binary search divides the array into half by calculating an array's middle element.

Implementation

The linear search can be implemented on any linear data structure such as vector, singly linked list, double linked list. In contrast, the binary search can be implemented on those data structures with two-way traversal, i.e., forward and backward traversal.

Complexity

The linear search is easy to use, or we can say that it is less complex as the elements for a linear search can be arranged in any order, whereas in a binary search, the elements must be arranged in a particular order.

Sorted elements

The elements for a linear search can be arranged in random order. It is not mandatory in linear search that the elements are arranged in a sorted order. On the other hand, in a binary search, the elements must be arranged in sorted order. It can be arranged either in an increasing or in decreasing order, and accordingly, the algorithm will be changed. As binary search uses a sorted array, it is necessary to insert the element at the proper place. In contrast, the linear search does not need a sorted array, so that the new element can be easily inserted at the end of the array.

Approach

The linear search uses an iterative approach to find the element, so it is also known as a sequential approach. In contrast, the binary search calculates the middle element of the array, so it uses the divide and conquer approach.

Data set

Linear search is not suitable for the large data set. If we want to search the element, which is the last element of the array, a linear search will start searching from the first element and goes on till the last element, so the time taken to search the element would be large. On the other hand, binary search is suitable for a large data set as it takes less time.

Speed

If the data set is large in linear search, then the computational cost would be high, and speed becomes slow. If the data set is large in binary search, then the computational cost would be less compared to a linear search, and speed becomes fast.

Dimensions

Linear search can be used on both single and multidimensional array, whereas the binary search can be implemented only on the one-dimensional array.

Efficiency

Linear search is less efficient when we consider the large data sets. Binary search is more efficient than the linear search in the case of large data sets.

Let's look at the differences in a tabular form.

Basis of comparison	Linear search	Binary search

Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.
Implementation	The linear search can be implemented on any linear data structure such as an array, linked list, etc.	The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferrable for the small-sized data sets.	It is preferrable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst- case scenario for finding the element is O(n).	In a binary search, the worst-case scenario for finding the element is $O(log_2n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element in the list is O(1).	In a binary search, the best-case scenario for finding the first element in the list is O(1).
Dimensional array	It can be implemented on both a single and multidimensional array.	It can be implemented only on a multidimensional array.

What Is 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.

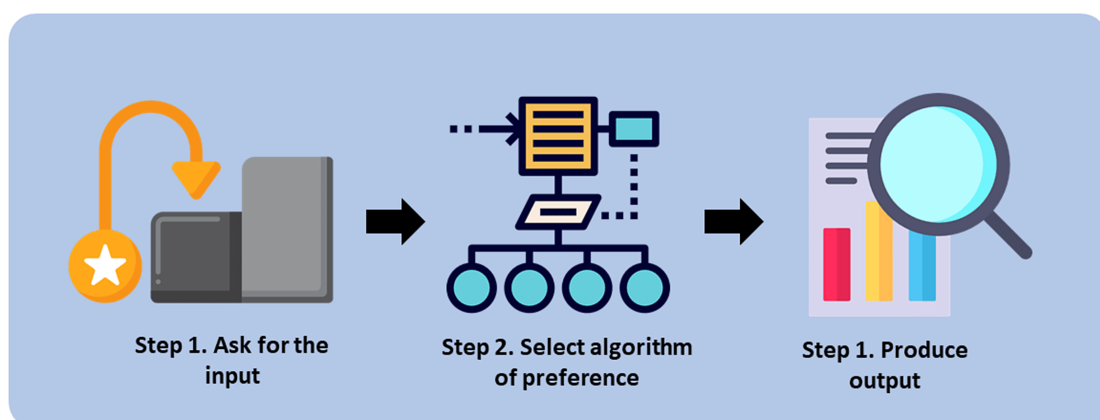
Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the processed data. It also aids in defining an algorithm's effectiveness and evaluating its performance.

Also Read: What Is An Algorithm?

What Is 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.

What Does It Take To Develop a Good Algorithm?



A good algorithm executes quickly and saves space in the process. You should find a happy medium of space and time (space and time complexity), but you can do with the average. Now, take a look at a simple algorithm for calculating the "mul" of two numbers.

Step 1: Start.

Step 2: Create two variables (a & b).

Step 3: Store integer values in ‘a’ and ‘b.’ -> Input

Step 4: Create a variable named 'mul'

Step 5: Store the mul of 'a' and 'b' in a variable named 'mul" -> Output

Step 6: End.

You will now see how significant space and time complexity is after understanding what they are.

How Significant Are Space and Time Complexity?

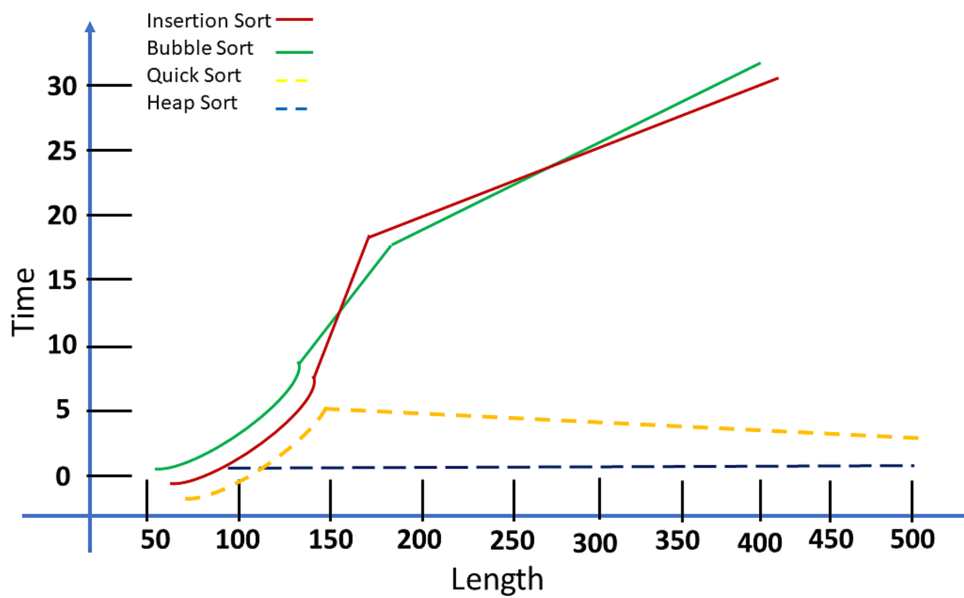
Significant in Terms of Time Complexity

The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

Here is an example.

Assume you have a set of numbers S= (10, 50, 20, 15, 30)

There are numerous algorithms for sorting the given numbers. However, not all of them are effective. To determine which is the most effective, you must perform computational analysis on each algorithm.



Here are some of the most critical findings from the graph:

- This test revealed the following sorting algorithms: Quicksort, Insertion sort, Bubble sort, and Heapsort.
- Python is the programming language used to complete the task, and the input size ranges from 50 to 500 characters.
- The results were as follows: "Heap Sort algorithms performed well despite the length of the lists; on the other hand, you discovered that Insertion sort and Bubble sort algorithms performed far worse, significantly increasing computing time." See the graph above for the results.
- Before you can run an analysis on any algorithm, you must first determine its stability. Understanding your data is the most important aspect of conducting a successful analysis.

What Are Asymptotic Notations?

Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows. This is also referred to as an algorithm's growth rate. When the input size increases, does

the algorithm become incredibly slow? Is it able to maintain its fast run time as the input size grows? You can answer these questions thanks to Asymptotic Notation.

You can't compare two algorithms head to head. It is heavily influenced by the tools and hardware you use for comparisons, such as the operating system, CPU model, processor generation, and so on. Even if you calculate time and space complexity for two algorithms running on the same system, the subtle changes in the system environment may affect their time and space complexity.

As a result, you compare space and time complexity using asymptotic analysis. It compares two algorithms based on changes in their performance as the input size is increased or decreased.

Asymptotic notations are classified into three types:

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

Now, go over each of these notations one by one.

1. Big-Oh (O) Notation

Paul Bachmann invented the big-O notation in 1894. He inadvertently introduced this notation in his discussion of function approximation.

From the definition: O (g(n)) =

{

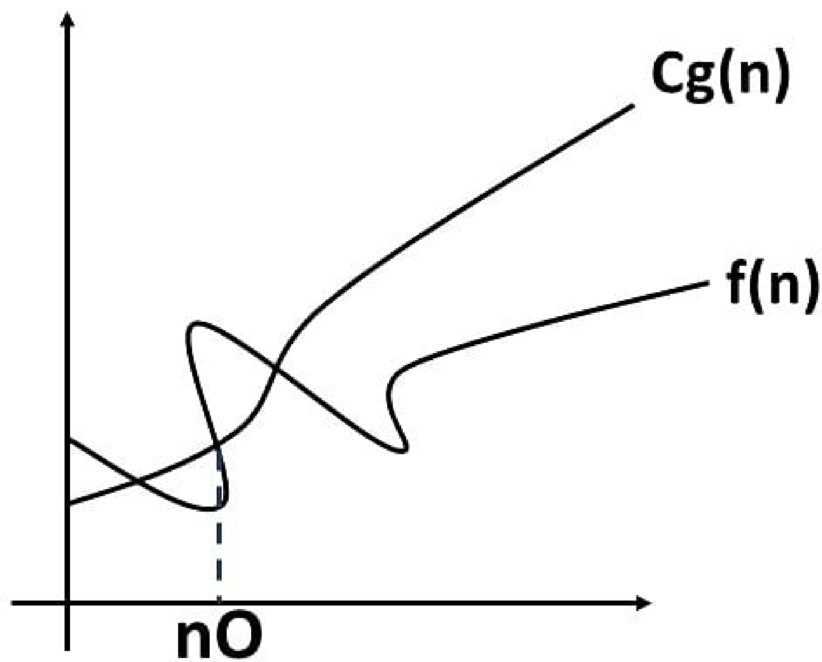
f(n) : there exist positive constant c and n0 such that 0 <= f(n) <= c*g(n)

For all n >= n0

}

'n' denotes the upper bound value. If a function is O(n), it is also O(n2) and O(n3).

It is the most widely used notation for Asymptotic analysis. It specifies the upper bound of a function, i.e., the maximum time required by an algorithm or the worst-case time complexity. In other words, it returns the highest possible output value (big-O) for a given input.

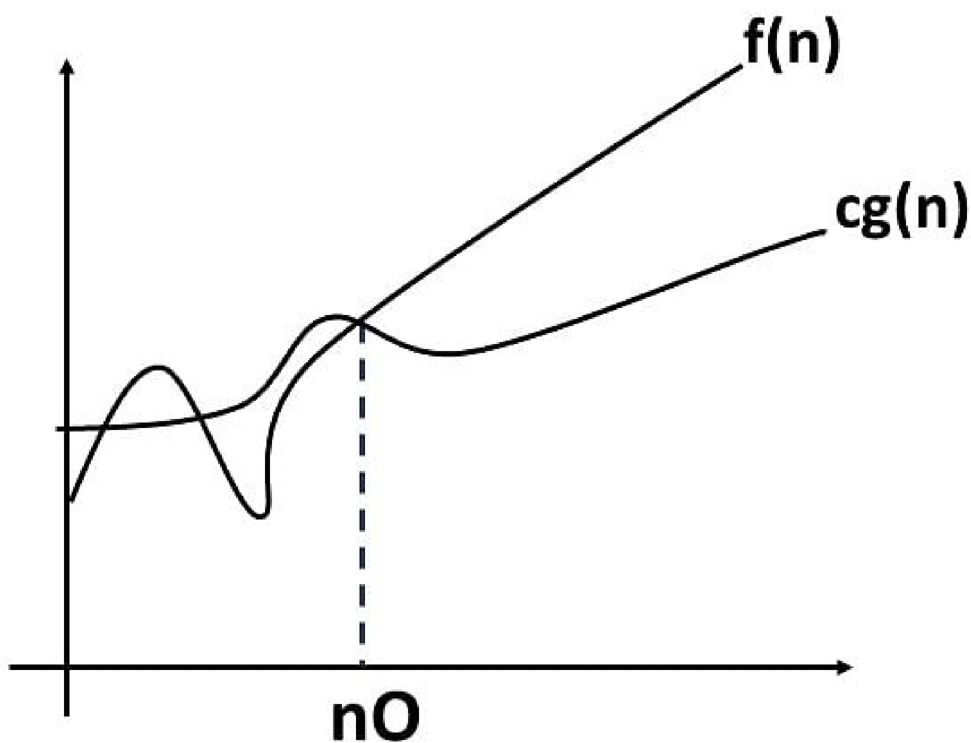


f(n) = O(g(n))

2. Big-Omega (Ω) notation

Big-Omega is an Asymptotic Notation for the best case or a floor growth rate for a given function. It gives you an asymptotic lower bound on the growth rate of an algorithm's runtime.

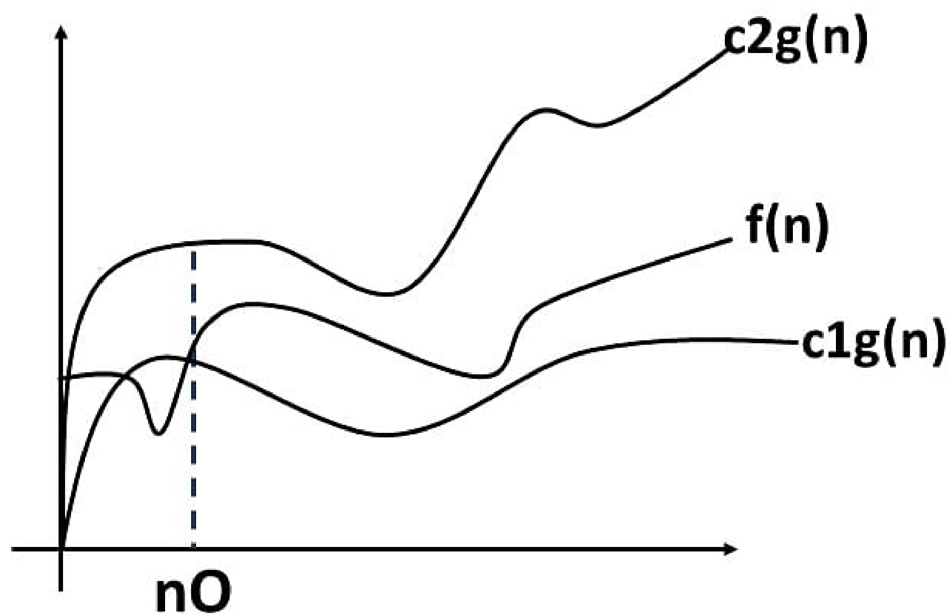
From the definition: The function f(n) is Ω (g(n)) if there exists a positive number c and N, such that f(n) >= cg(n) for all n >= N.



3. Big-Theta (Θ) notation

Big theta defines a function's lower and upper bounds, i.e., it exists as both, most, and least boundaries for a given input value.

From the definition : $f(n)$ is $\Theta(g(n))$ if there exists positive numbers c1, c2 and N such that $c1g(n) \le f(n) \le c2g(n)$ for all $n \ge N$.



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

Best Case, Worst Case, and Average Case in Asymptotic Analysis

Best Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time. In this case, the execution time serves as a lower bound on the algorithm's time complexity.

Average Case: You add the running times for each possible input combination and take the average in the average case. Here, the execution time serves as both a lower and upper bound on the algorithm's time complexity.

Worst Case: It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible. In this case, the execution time serves as an upper bound on the algorithm's time complexity.

You will now see how to calculate space and time complexity after grasping the significance of space and time complexity.

Significant in Terms of Space Complexity

Space complexity refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution. Calculate the space occupied by variables in an algorithm/program to determine space complexity.

However, people frequently confuse Space-complexity with auxiliary space. Auxiliary space is simply extra or temporary space, and it is not the same as space complexity. To put it another way,

Auxiliary space + space use by input values = Space Complexity

The best algorithm/program should have a low level of space complexity. The less space required, the faster it executes.

Method for Calculating Space and Time Complexity

Methods for Calculating Time Complexity

To calculate time complexity, you must consider each line of code in the program. Consider the multiplication function as an example. Now, calculate the time complexity of the multiply function:

```
1.   mul <- 1
2.   i <- 1
3.   While i <= n do
4.       mul = mul * 1
5.       i = i + 1
6.   End while
```

Let T(n) be a function of the algorithm's time complexity. Lines 1 and 2 have a time complexity of O. (1). Line 3 represents a loop. As a result, you must repeat lines 4 and 5 (n -1) times. As a result, the time complexity of lines 4 and 5 is O. (n).

Finally, adding the time complexity of all the lines yields the overall time complexity of the multiple function fT(n) = O(n).

The iterative method gets its name because it calculates an iterative algorithm's time complexity by parsing it line by line and adding the complexity.

Aside from the iterative method, several other concepts are used in various cases. The recursive process, for example, is an excellent way to calculate time complexity for recurrent solutions that use recursive trees or substitutions. The master's theorem is another popular method for calculating time complexity.

Methods for Calculating Space Complexity

With an example, you will go over how to calculate space complexity in this section. Here is an example of computing the multiplication of array elements:

```
1.   int mul, i
2.   While i < = n do
3.       mul <- mul * array[i]
4.       i <- i + 1
5.   end while
6.   return mul
```

Let S(n) denote the algorithm's space complexity. In most systems, an integer occupies 4 bytes of memory. As a result, the number of allocated bytes would be the space complexity.

Line 1 allocates memory space for two integers, resulting in S(n) = 4 bytes multiplied by 2 = 8 bytes. Line 2 represents a loop. Lines 3 and 4 assign a value to an already existing variable. As a result, there is no need to set aside any space. The return statement in line 6 will allocate one more memory case. As a result, S(n)= 4 times 2 + 4 = 12 bytes.

Because the array is used in the algorithm to allocate n cases of integers, the final space complexity will be fS(n) = n + 12 = O (n).

As you progress through this tutorial, you will see some differences between space and time complexity.

Time Complexity vs. Space Complexity

You now understand space and time complexity fundamentals and how to calculate it for an algorithm or program. In this section, you will summarise all previous discussions and list the key differences in a table.

Time Complexity	Space Complexity
Calculates the time required	Estimates the space memory required
Time is counted for all statements	Memory space is counted for all variables, inputs, and outputs.
The size of the input data is the primary determinant.	Primarily determined by the auxiliary variable size
More crucial in terms of solution optimization	More essential in terms of solution optimization

Now that you have reached the end of the tutorial on space and time complexity, sum up what you've learned thus far.