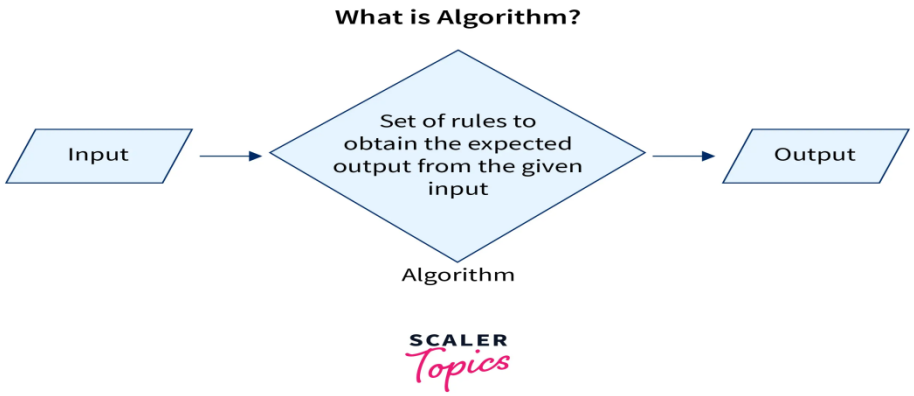
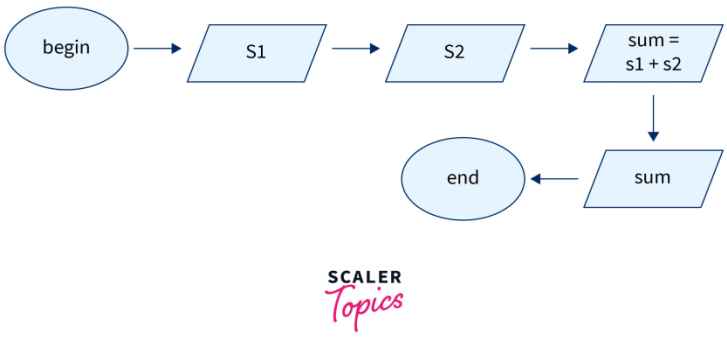
an **algorithm** is defined as the **set of well-defined instructions**, that helps us to solve any particular problem.



For example, the **algorithm to add two numbers** will be :

- Take two numbers as input
- Perform arithmetic addition using ('+') operator
- Return the result



There are certain parameters that we should note while writing any algorithm.

The parameters for good algorithms are stated below :

- The input and output of the algorithm must be accurate.
- The steps in the algorithm should be clear and with no ambiguity
- It should be effective, to solve a problem.

- The algorithm should not be written in any programming language. That means, it should be possible for the users to code the algorithm into any of the programming languages of their choice.

What is the Analysis of the Algorithm ?

Analysis of algorithms is the process of finding the computational complexity of any algorithm. By computational complexity, we are referring to the amount of time taken, space, and any other resources needed to execute(run) the algorithm.

The goal of algorithm analysis is to compare different algorithms that are used to solve the same problem. This is done to evaluate which method solves a certain problem more quickly, efficiently, and memory-wise.

Usually, the time complexity is determined by the size of the algorithm's input to the number of steps taken to execute it. Also, the space complexity is mostly determined by the amount of storage needed by the algorithm to execute.

If an algorithm runs in a reasonable amount of time or space on any machine, where that time and space are measured as a function of the size of the input (usually), then the **algorithm** is considered to be **efficient**. In simple terms, if the resource consumption or the computation cost of an algorithm grows slowly with the growth of the size of the input, then the algorithm might be considered as **efficient**.

Asymptotic Analysis: Big-O Notation and More

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

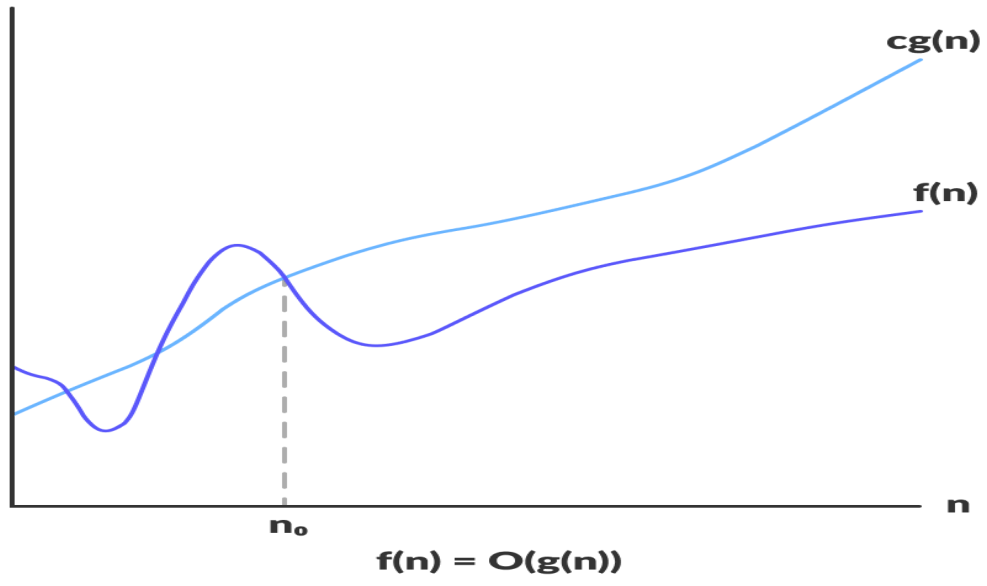
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

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



Big-O gives the upper bound of a function

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

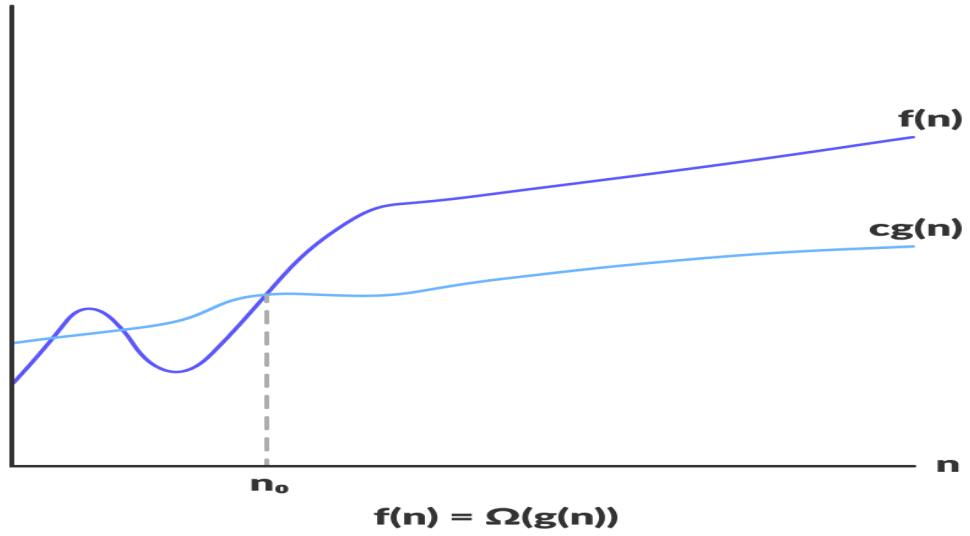
The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive constant `c` such that it lies between `0` and `cg(n)`, for sufficiently large `n`.

For any value of `n`, the running time of an algorithm does not cross the time provided by `O(g(n))`.

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-notation)

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



Omega gives the lower bound of a function

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

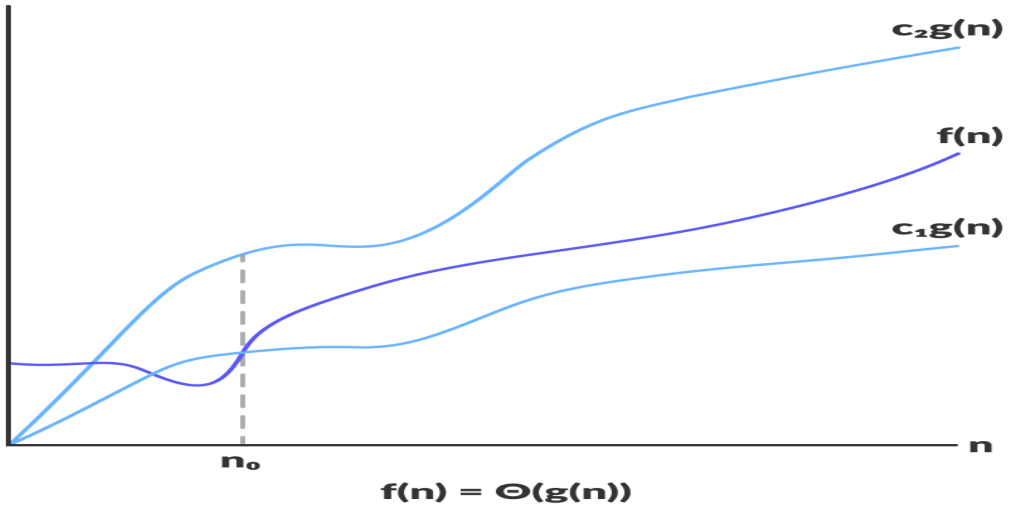
The above expression can be described as a function `f(n)` belongs to the set `Ω(g(n))` if there exists a positive constant `c` such that it lies above `cg(n)`, for sufficiently large `n`.

For any value of `n`, the minimum time required by the algorithm is given by Omega `Ω(g(n))`

.

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 bounds the function within constants factors

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

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

The above expression can be described as a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

If a function $f(n)$ lies anywhere in between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n0$, then $f(n)$ is said to be asymptotically tight bound.

Types of Algorithm Analysis

Till now we must already be aware that, the running time of an algorithm increases with the increase in the size of the input. However, if the running time is constant then it will remain constant, no matter what the input is.

Even if the size of the input is the same sometimes, the running time of the algorithm still may vary. Hence, we perform the best, average, and worst-case analyses of the algorithms, covering all the possible cases where the algorithm may behave abruptly high or low.

There are four types of analysis of algorithms. They are stated below :

1. Best case
2. Average case
3. Worst case
4. Amortized analysis

Let us look at each of them in detail.

1. Best Case Analysis of Algorithms

The best case analysis of algorithms gives us the **lower bound** on the **running time** of the algorithm for any given input. In simple words, it states that **any program will need at least** (greater than or equal to) that time to run. **For example**, suppose we have an algorithm that has the best case time complexity is O(N)O(N), then we can say that the program will take a minimum of O(N)O(N) time to run, it can never take sometime less than that.

The best case time or space complexity is often represented in terms of the **Big Omega (Ω)(Ω)** notation.

Example for Best Case Analysis :
Let us take the example of the linear search to calculate the best time complexity. Suppose, you have an array of numbers and you have to search for a number.

Find the code below for the above problem :

Code :

```
int linear_search(int arr, int l, int target) {
    int i;
    for (i = 0; i < l; i++) {
        if (arr[i] == target) {
            return arr[i]
        }
    }
    return -1
}
```

Now suppose, the number you are searching for is present at the very beginning index of the array. In that case, your algorithm will take O(1)O(1) time to find the number in the best case. So, the best case complexity for this algorithm becomes O(1)O(1), and you will get your output in a constant time.

How frequently do we use the best case analysis of any algorithm ?

The best case is rarely necessary for measuring the runtime of the algorithms in practice. An algorithm is never created using the best-case scenario.

2. Worst Case Analysis of Algorithms

The worst-case analysis of algorithms gives us the **upper bound** on the **running time** of the algorithm for any given input. In simple words, it states that **any program will need maximum that time** (less than or equal to) to run. **For example**, suppose we have an algorithm that has the worst-case time complexity is O(N2)O(N2), then we can say that the program will take a maximum of O(N2)O(N2) time to run, for an input of size NN it can never take more time than that to run.

The worst-case time or space complexity is often represented in terms of the **Big Oh (O)(O)** notation.

Example for worst case analysis :
Let us take our previous example where we were performing the **linear search**. Suppose, this time the element we are trying to find is at the end of the array. So, we will have to traverse the whole array before finding the element. Hence, we can say that the worst case for this algorithm is O(N)O(N) itself. Because we have to go through at most NN elements before finding our target. So, this is how we calculate the worst case of the algorithms.

How frequently do we use the worst-case analysis of any algorithm ?

In actual life, we typically analyze an algorithm's worst-case scenario for most of the cases. The longest running time W(n)W(n) of an algorithm for any input of size n is referred to as worst-case time complexity.

3. Average Case Analysis of Algorithms

As the name suggests, the average case analysis of algorithms takes the sum of the running time on every possible input, and after that, it takes the average. So, in this case, the execution time of the algorithm acts as both the lower and upper bound. In simple terms, we can get an idea of the average running time of the algorithm through it.

Generally, the average case of the algorithms is as high as the worst-case running of it. Hence, it roughly gives us an estimation of the worst case itself.

The average case time or space complexity is often represented in terms of the **Big theta (Θ)(Θ)** notation.

Example for average case analysis :
In our previous example, suppose we need to find any element which is present in the mid of our array. So, for that, we need to traverse at least the half length of the array. In other words, it will take us O(n/2)O(n/2) time for us to traverse the half-length. The time complexity O(n/2)O(n/2) is as good as O(n)O(n). That is why we say that the average case in most of the cases depicts the worst case of an algorithm.

How frequently do we use the average case analysis of any algorithm ?

To be precise, it is usually **difficult** to analyze the average case running time of an algorithm. It is **simpler** to calculate the **worst case** instead. This is mainly because it might not be a very exact thing to declare any input as the **"average"** input for any problem. Therefore, prior knowledge of the distribution of the input cases is necessary for a useful examination of the average behavior of an algorithm, which is necessarily an impossible condition.

4. Amortized Analysis of Algorithms

The amortized analysis of the algorithms mainly deals with the overall cost of the operations. It does not mention the complexity of any one particular operation in the sequence. The total cost of the operations is the major focus of amortized analysis.

In times when only a few operations are slow but a majority of other operations are quicker, we perform an amortized analysis. Through this, we return the **average running time per operation** in the **worst case.**

Example for Amortized case analysis
A perfect example of understanding the amortized case analysis would be a **hash table**. You must have used or implemented the hash tables or dictionaries, in whichever language you code.

So, in a hash table, for searching, the time taken is generally O(1)O(1), or constant time. However, there are situations when it takes O(N)O(N) times because it has to execute that many operations to search for a value.

Similarly, when we try to insert some value in a hash table, for a majority of the cases it takes a time of O(1)O(1), but still, there are cases when suppose multiple collisions occur, when it takes a time of O(N)O(N), for the collisions resolution.

Other data structures we need amortized analysis are Disjoint Sets etc.

How frequently do we use the amortized case analysis of any algorithm ?

Whenever a single operation or very few operations runs slowly on occasion, but most of the operations run quickly and frequently, then the amortized case analysis is used.