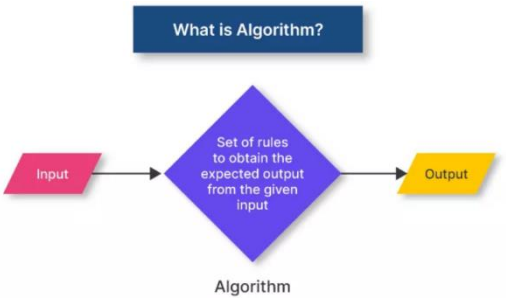
Asymptotic Analysis

- As we know that data structure is a way of organizing the data efficiently and that **efficiency is measured either in terms of time or space.**



- asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.
- the **ideal data structure** is a structure **that occupies the least possible time to perform all its operation and the memory space.**
- Our **focus** would be on **finding the time complexity rather than space complexity,** and by finding the time complexity, we can decide which data structure is the best for an algorithm.

Comparing time complexity of algorithms:

- The time complexity can be compared based on the operations performed in an algorithm.
- Suppose we have an **array of 100 elements,** and we want **to insert a new element at the beginning** of the array. This becomes a **very tedious task** as we first need to **shift the elements towards the right,** and we will **add new element at the starting of the array**.
- Suppose we consider **the linked list as a data structure to add the element at the beginning.** The linked list **contains two parts,** i.e., **data and address of the next node.** We simply add the **address of the first node in the new node,** and **head pointer will now point to the newly added node.** Therefore, we conclude that **adding the data at the beginning of the linked list is faster than the arrays.** In this way, we can compare the data structures and select the best possible data structure for performing the operations.

How to find the Time Complexity or running time for performing the operations?

- The measuring of the actual running time is not practical at all.(Not every system takes same time for same task)
- The running time to perform any operation depends on the size of the input.
- Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose **each element takes one unit of time. There are five elements, so five units of time would be taken.**
- Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It concludes that time complexity depends upon the input size.
- Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)?

We can compare the data structures by comparing their f(n) values.

We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

How to Find f(n)?

To determine f(n), follow these steps:

1. Analyze the Algorithm or Data Structure:
    - Break down the steps of the algorithm or operations performed by the data structure for a given input size.
    - Count the number of iterations, recursive calls, or operations.

For example:

- A loop running n times has f(n)=n
- A nested loop where each loop runs n times has $f(n)=n^2$

2. Use Asymptotic Notation
    - Focus on the **dominant term** in f(n) as n grows, ignoring constants and lower-order terms.
    - Express f(n) in terms of **O-notation (Big-O), Θ Theta -notation,** or **Ω Omega-notation,** depending on the context.

Example:

- $f(n) = 5n^2 + 3n + 2$ simplifies to $O(n^2)$

3. Asymptotic Notations:

Big O Notation (O):

- Represents the **worst-case time complexity**.
- Provides an **upper bound** on the growth rate of the function.

Omega Notation (Ω):

- Represents the **best-case time complexity**.
- Provides a **lower bound** on the growth rate of the function.

Theta Notation (θ):

- Represents the **average-case time complexity**.
- Provides **tight bounds,** meaning the function is bounded both above and below.