Agenda

1. What is an Algorithm?
2. Data Structure and Types
3. Why Learn Data Structures and Algorithms?
4. Asymptotic Analysis

What is an Algorithm?

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces desired output. For example,

An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of Good Algorithms

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Algorithm Examples

Algorithm 1: Add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Algorithm 2: Find the largest number among three numbers

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a > b
           If a > c
              Display a is the largest number.
           Else
              Display c is the largest number.
        Else
           If b > c
              Display b is the largest number.
           Else
              Display c is the greatest number.
Step 5: Stop
```

Algorithm 3: Find Root of the quadratic equatin ax^2 + bx + c = 0

```
Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
         D ← b^2-4ac
Step 4: If D ≥ 0
              r1 ← (-b+√D)/2a
              r2 ← (-b-√D)/2a
              Display r1 and r2 as roots.
        Else
              Calculate real part and imaginary part
              rp ← -b/2a
              ip ← √(-D)/2a
              Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop
```

Algorithm 4: Find the factorial of a number

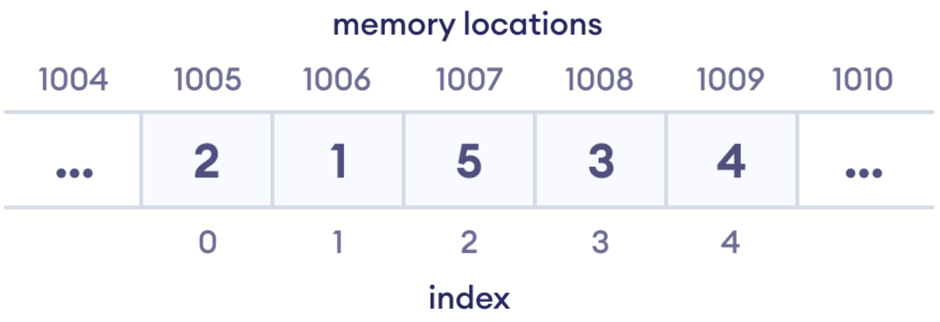
```
Step 1: Start
Step 2: Declare variables n, factorial and i.
Step 3: Initialize variables
          factorial ← 1
          i ← 1
Step 4: Read value of n
Step 5: Repeat the steps until i = n
     5.1: factorial ← factorial*i
     5.2: i ← i+1
Step 6: Display factorial
Step 7: Stop
```

Data Structure and Types

What are Data Structures?

Data structure is a storage that is used to store and organise data. It is a way of arranging data on a computer so that it can be accessed and update efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store da sequentially in the memory, then you can go for the Array data structure.



Note: Data structure and data types are slightly different. Data structure is the collection of data types arranged in a specific order.

Types of Data Structure

Basically, data structures are divided into two categories:

- Linear data structure
- Non-linear data structure

Let's learn about each type in detail.

Linear data structures

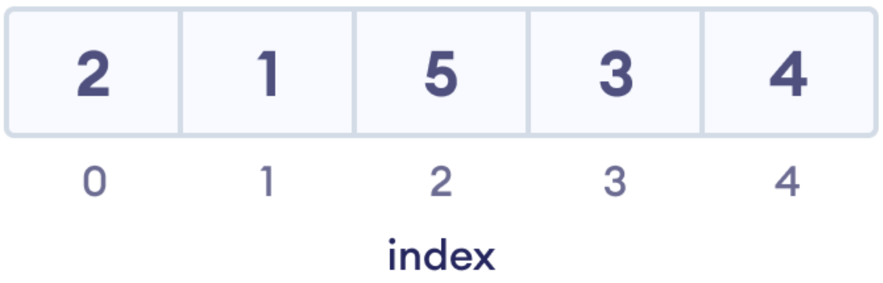
In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy implement.

However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.

Popular linear data structures are:

1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements th can be stored in the form of arrays is determined by the programming language.

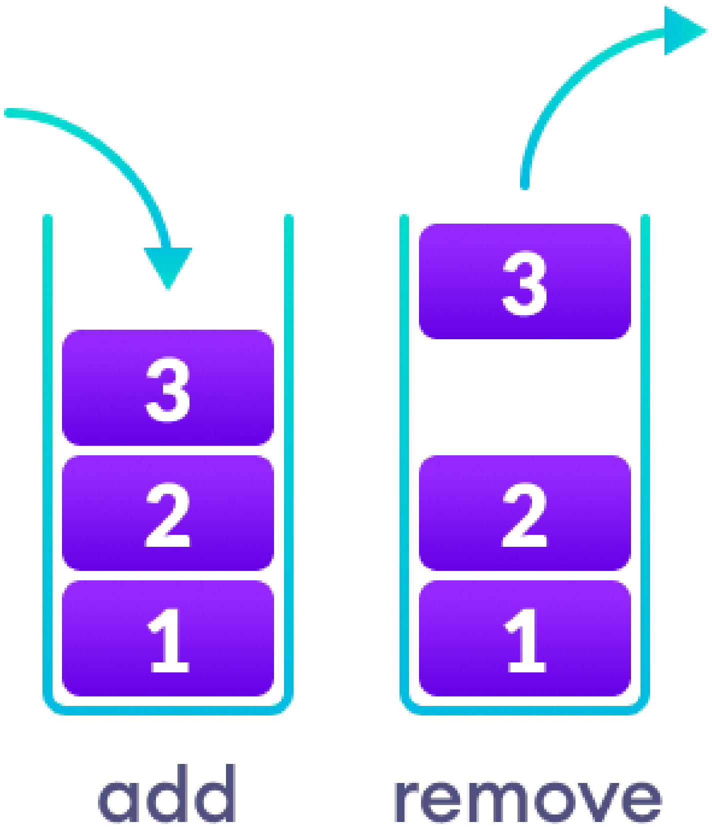


An array with each element represented by an index

2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.

It works just like a pile of plates where the last plate kept on the pile will be removed first.

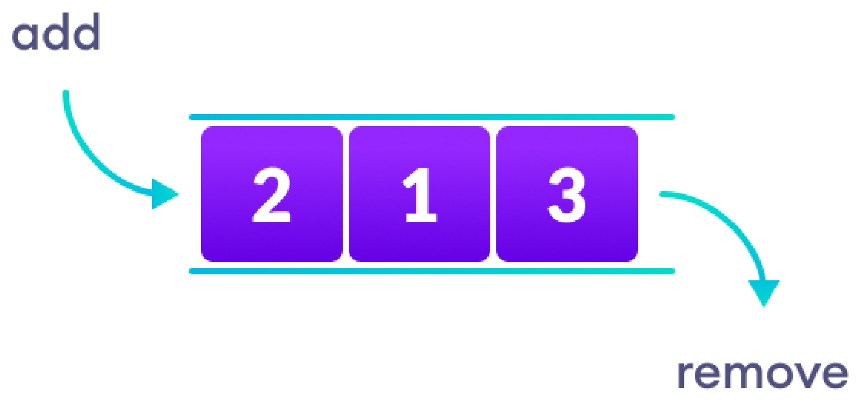


In a stack, operations can be perform only from one end (top here).

3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first.

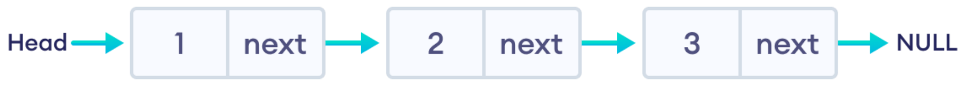
It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.



In a queue, addition and removal are performed from separate ends.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the ne node.



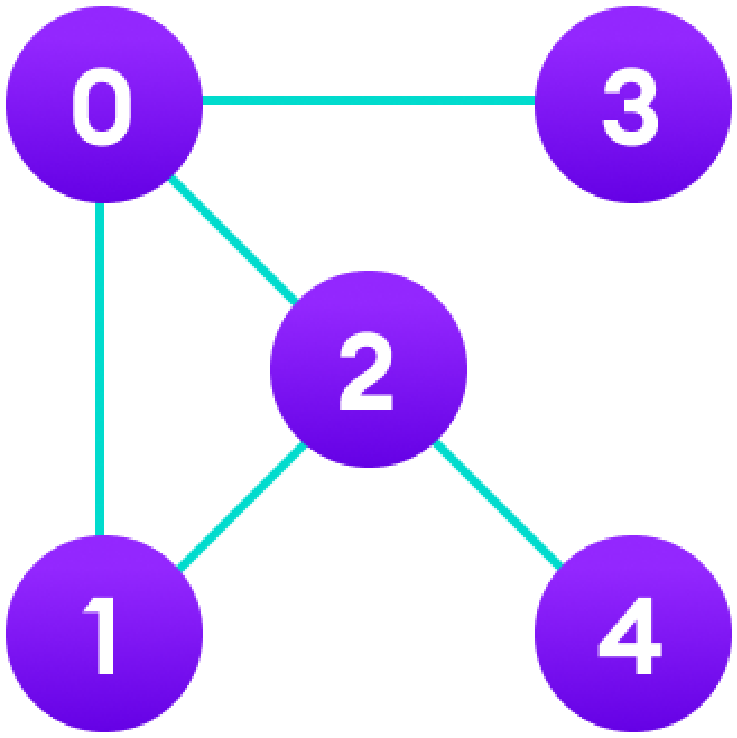
Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner whe one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.



Graph data structure example

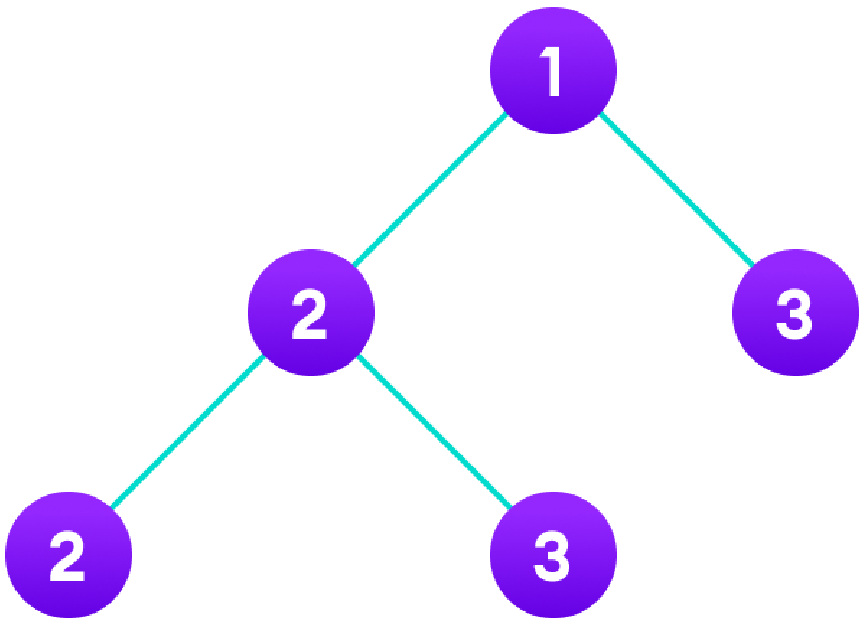
Popular Graph Based Data Structures:

- Spanning Tree and Minimum Spanning Tree
- Strongly Connected Components
- Adjacency Matrix
- Adjacency List

All these algorithms will be discussed in detail in later sessions.

2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.



Tree data structure example

Popular Tree based Data Structure

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree
- Red-Black Tree

All these algorithms will be discussed in detail in later sessions.

Linear Vs Non-linear Data Structures

Now that we know about linear and non-linear data structures, let's see the major differences between them.

Linear Data Structures	Non Linear Data Structures
The data items are arranged in sequential order, one after the other.	The data items are arranged in non-sequential order (hierarchical manner).
All the items are present on the single layer.	The data items are present at different layers.
It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.	It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.
The memory utilization is not efficient.	Different structures utilize memory in different efficient ways depending on the need.
The time complexity increase with the data size.	Time complexity remains the same.
Example: Arrays, Stack, Queue	Example: Tree, Graph, Map

Why Data Structure?

Knowledge about data structures help you understand the working of each data structure. And, based on that you can select the right data structures f your project.

This helps you write memory and time efficient code.

Informally, an algorithm is nothing but a mention of steps to solve a problem. They are essentially a solution.

For example, an algorithm to solve the problem of factorials might look something like this:

Problem: Find the factorial of n

```
Initialize fact = 1
For every value v in range 1 to n:
    Multiply the fact by v
fact contains the factorial of n
```

Here, the algorithm is written in English. If it was written in a programming language, we would call it to **code** instead. Here is a code for finding tl factorial of a number in Javascript.

```
// program to find the factorial of a number

// take input from the user
const number = parseInt(prompt('Enter a positive integer: '));

// checking if number is negative
if (number < 0) {
    console.log('Error! Factorial for negative number does not exist.');
}

// if number is 0
else if (number === 0) {
    console.log(`The factorial of ${number} is 1.`);
}

// if number is positive
else {
    let fact = 1;
    for (i = 1; i <= number; i++) {
        fact *= i;
    }
    console.log(`The factorial of ${number} is ${fact}.`);
}
```

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are used to solve the problem using th data.

Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use eac one of them. It also teaches you the science of evaluating the efficiency of an algorithm. This enables you to choose the best of various choices.

Use of Data Structures and Algorithms to Make Your Code Scalable

Time is precious.

Suppose, Alice and Bob are trying to solve a simple problem of finding the sum of the first 1011 natural numbers. While Bob was writing the algorithr Alice implemented it proving that it is as simple as criticising Donald Trump.

Algorithm (by Bob)

```
Initialize sum = 0
for every natural number n in range 1 to 1011 (inclusive):
    add n to sum
sum is your answer
```

Code (by Alice)

```
let sum = 0;

// looping from i = 1 to number
// in each iteration, i is increased by 1
for (let i = 1; i <= 100000000000; i++) {
    sum += i;
}

console.log('The sum of natural numbers:', sum);
```

Alice and Bob are feeling euphoric of themselves that they could build something of their own in almost no time. Let's sneak into their workspace ar listen to their conversation.

```
Alice: Let's run this code and find out the sum.
Bob: I ran this code a few minutes back but it's still not showing the output. What's wrong with it?
```

Oops, something went wrong! A computer is the most deterministic machine. Going back and trying to run it again won't help. So let's analyse what wrong with this simple code.

Two of the most valuable resources for a computer program are **time** and **memory**.

The time taken by the computer to run code is:

```
Time to run code = number of instructions * time to execute each instruction
```

The number of instructions depends on the code you used, and the time taken to execute each code depends on your machine and compiler.

In this case, the total number of instructions executed (let's say x) are `x = 1 + (10^11 + 1) + (10^11) + 1` , which is `x = 2 * 10^11 + 3`

Let us assume that a computer can execute `y = 10^8` instructions in one second (it can vary subject to machine configuration). The time taken to ru above code is

```
Time taken to run code = x/y (greater than 16 minutes)
```

Is it possible to optimise the algorithm so that Alice and Bob do not have to wait for 16 minutes every time they run this code?

I am sure that you already guessed the right method. The sum of first N natural numbers is given by the formula:

```
Sum = N * (N + 1) / 2
```

Converting it into code will look something like this:

```
function sum(N) {
    return N * (N + 1) / 2;
}
```

This code executes in just one instruction and gets the task done no matter what the value is. Let it be greater than the total number of atoms in th universe. It will find the result in no time.

The time taken to solve the problem, in this case, is `1/y` (which is 10 nanoseconds).

Note: Computers take a few instructions (not 1) to compute multiplication and division. I have said 1 just for the sake of simplicity.

More on Scalability

Scalability is scale plus ability, which means the quality of an algorithm/system to handle the problem of larger size.

Consider the problem of setting up a classroom of 50 students. One of the simplest solutions is to book a room, get a blackboard, a few chalks, and th problem is solved.

But what if the size of the problem increases? What if the number of students increased to 200?

The solution still holds but it needs more resources. In this case, you will probably need a much larger room (probably a theater), a projector screen ar a digital pen.

What if the number of students increased to 1000?

The solution fails or uses a lot of resources when the size of the problem increases. This means, your solution wasn't scalable.

What is a scalable solution then?

Consider a site like Khanacademy, millions of students can see videos, read answers at the same time and no more resources are required. So, th solution can solve the problems of larger size under resource crunch.

If you see our first solution to find the sum of first N natural numbers, it wasn't scalable. It's because it required linear growth in time with the linear grow in the size of the problem. Such algorithms are also known as linearly scalable algorithms.

Our second solution was very scalable and didn't require the use of any more time to solve a problem of larger size. These are known as constant-tim algorithms.

Memory is expensive

Memory is not always available in abundance. While dealing with code/system which requires you to store or produce a lot of data, it is critical for yo algorithm to save the usage of memory wherever possible. For example: While storing data about people, you can save memory by storing only the date of birth, not their age. You can always calculate it on the fly using their date of birth and current date.

Example of an Algorithm's Efficiency

Here are some examples of what learning algorithms and data structures enable you to do:

Example 1: Age Group Problem

Problems like finding the people of a certain age group can easily be solved with a little modified version of the binary search algorithm (assuming th the data is sorted).

The naive algorithm which goes through all the persons one by one, and checks if it falls in the given age group is linearly scalable. Whereas, bina search claims itself to be a logarithmically scalable algorithm. This means that if the size of the problem is squared, the time taken to solve it is on doubled.

Suppose, it takes 1 second to find all the people at a certain age for a group of 1000. Then for a group of 1 million people,

- the binary search algorithm will take only 2 seconds to solve the problem
- the naive algorithm might take 1 million seconds, which is around 12 days

The same binary search algorithm is used to find the square root of a number.

Example 2: Rubik's Cube Problem

Imagine you are writing a program to find the solution of a Rubik's cube.

Cute looking puzzle has annoyingly 43,252,003,274,489,856,000 positions, and these are just positions! Imagine the number of paths one can take reach the wrong positions.

Fortunately, the way to solve this problem can be represented by the graph data structure. There is a graph algorithm known as Dijkstra algorithm which allows you to solve this problem in linear time. Yes, you heard it right. It means that it allows you to reach the solved position in minimum number of states.

Asymptotic Analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measure 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 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.

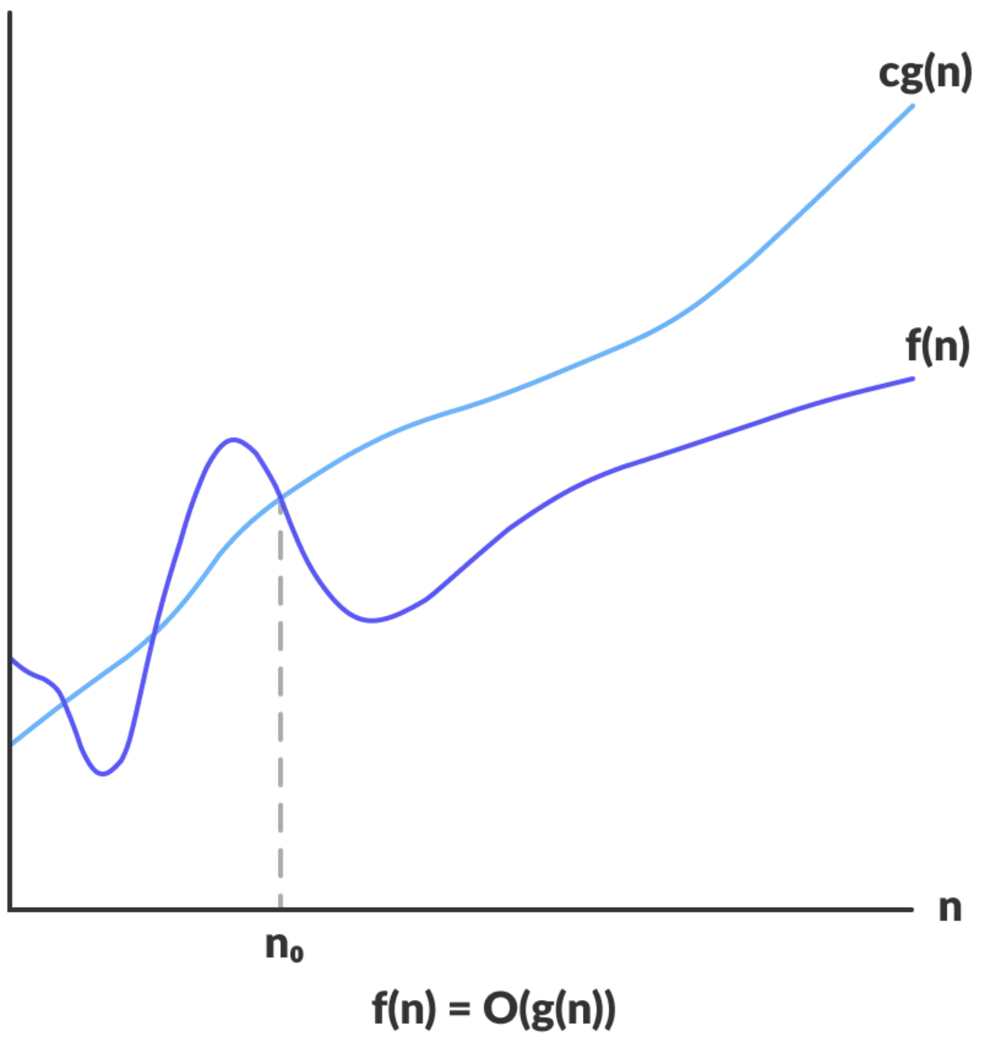
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

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 }
```

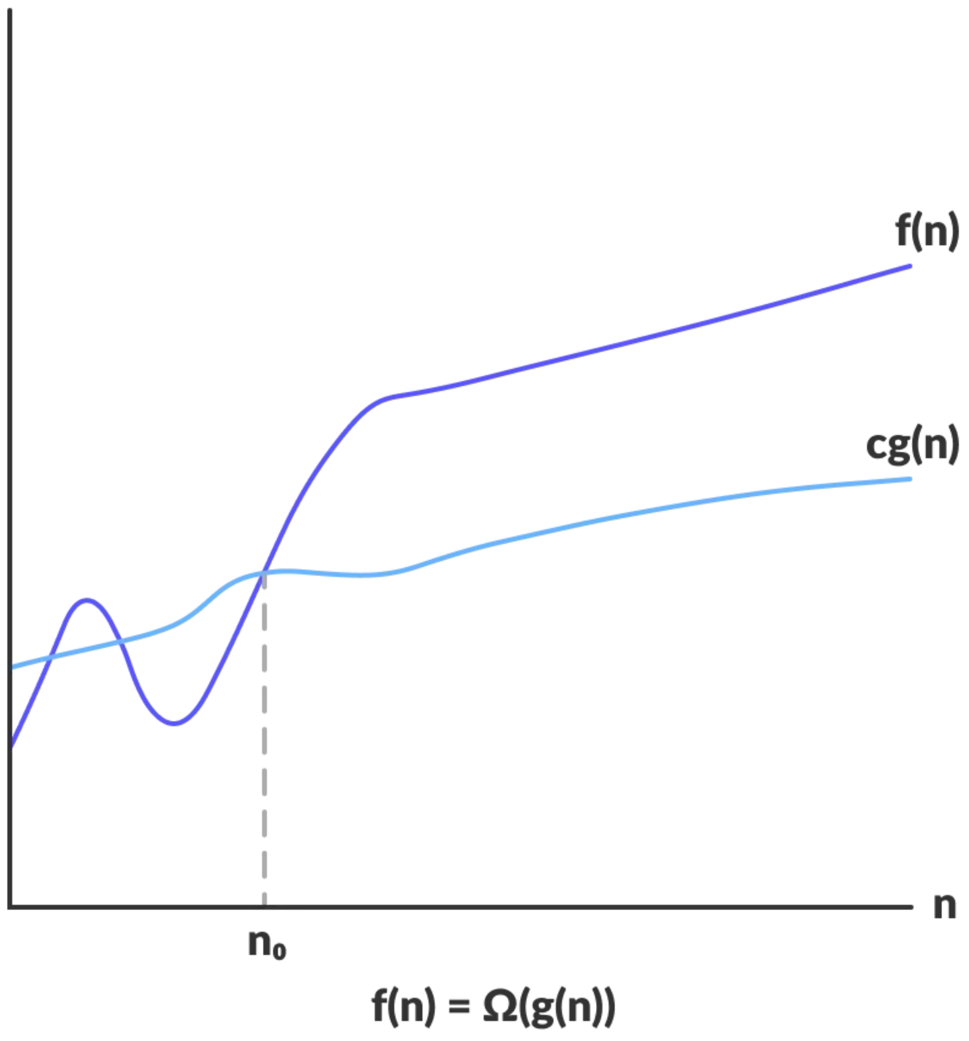
The above expression can be described as a function `f(n)` belongs to the set `O(g(n))` if there exists a positive constant `c` such that it lie between `0` and `cg(n)`, for sufficiently large `n`.

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 analyse an algorithm as we are always interested in the worst-cas 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

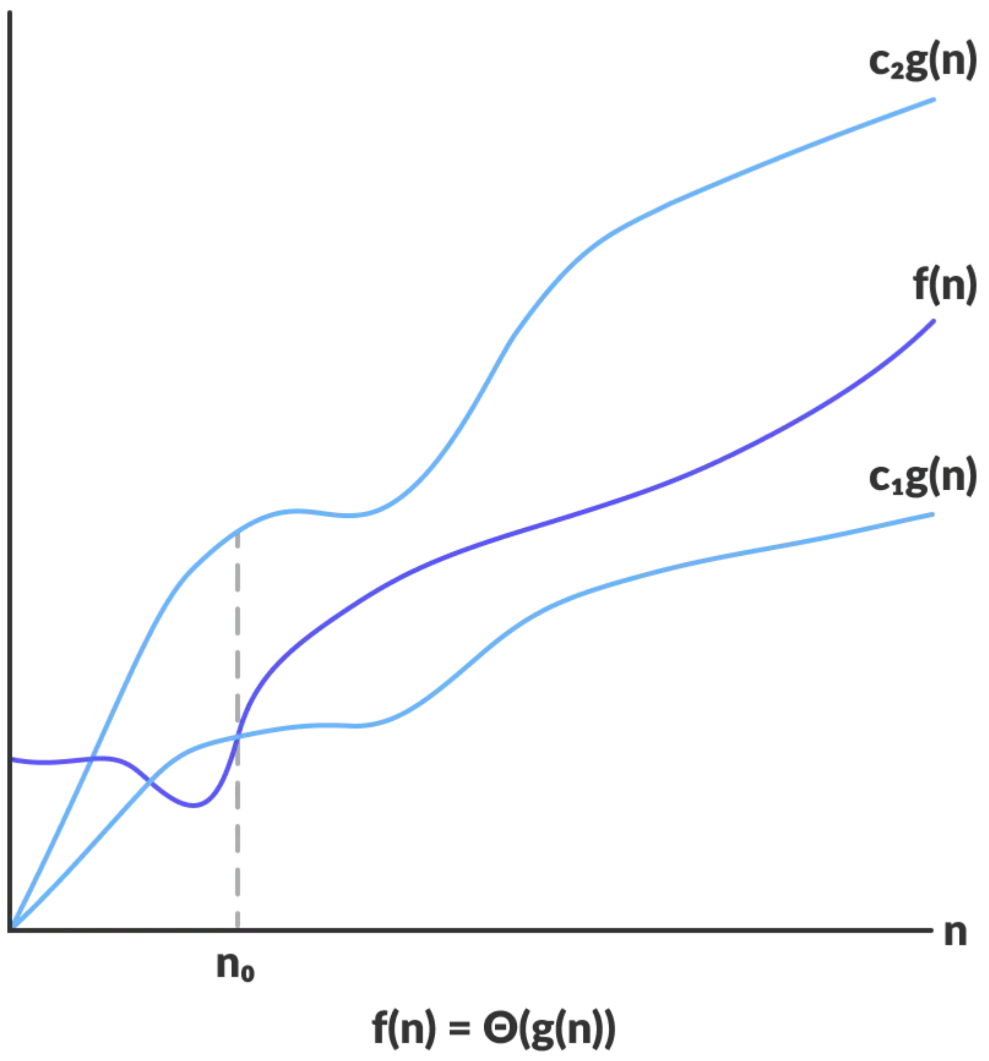
```
Ω(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }
```

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 lie 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 used for analysing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function `g(n)` , `Θ(g(n))` is given by the relation:

```
Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0
            such that 0 ≤ c1*g(n) ≤ f(n) ≤ c2*g(n) for all n ≥ n0 }
```

The above expression can be described as a function `f(n)` belongs to the set `Θ(g(n))` if there exist positive constants `c1` and `c2` such that can be sandwiched between `c1*g(n)` and `c2*g(n)` , for sufficiently large n.

If a function `f(n)` lies anywhere in between `c1*g(n)` and `c2*g(n)` for all `n ≥ n0` , then `f(n)` is said to be asymptotically tight bound.

Conclusion:

In this session, we have learnt about:

- Algorithms
- Data Structures and their types
- Asymptotic notations.

Thank You !