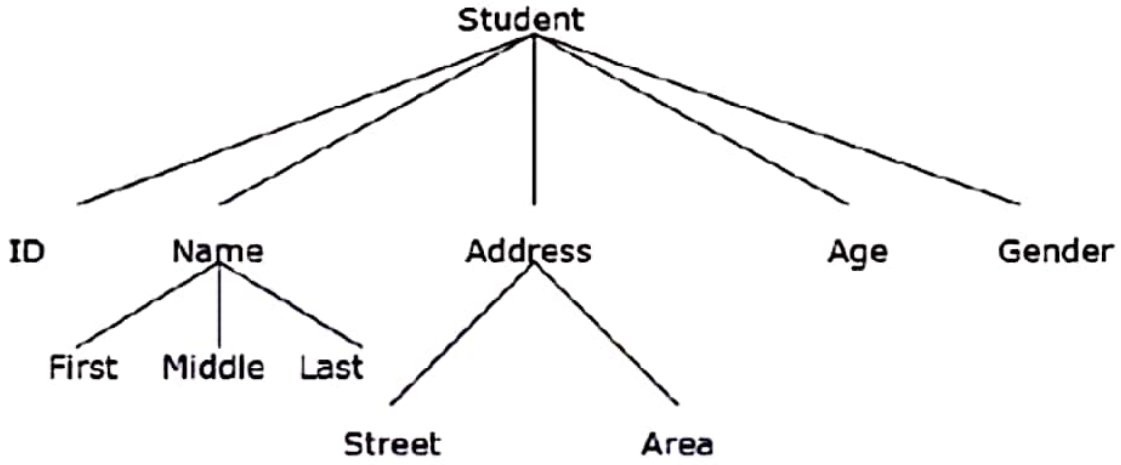
MODULE – I

Data:

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items those that are not are called elementary items. For example, a student's name may be divided into three sub items - [first name, middle name and last name) but the ID of a student would normally be treated as a single item.



In the above example (ID, Age, Gender, First, Middie, Last, Street Area) are elementary data items, whereas (Name, Address) are group data items.

Abstract Data Type (ADT)

Abstract data types or ADTS are a mathematical specification of a set of data and the set of operations that can be performed on the data. Data Structure.

Data Structure-

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

Introduction to Data Structure :-

→ In computer Science a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

→ A data structure is an arrangement of data in a computer's memory or even disk storage.

→ An example of several common data structures are arrays, linked lists, queues, stacks, binary trees and hash tables.

→ Different kinds of data structures are suited to different kinds of applications and some are highly specialized to specific tasks.

→ For example B-Trees are particularly well-suited for implementation of databases while compiler implementations usually use hash tables to look up identifiers.

→ Data Structures are used in almost every program or software system.

→ Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services.

Scanned with CamScanner

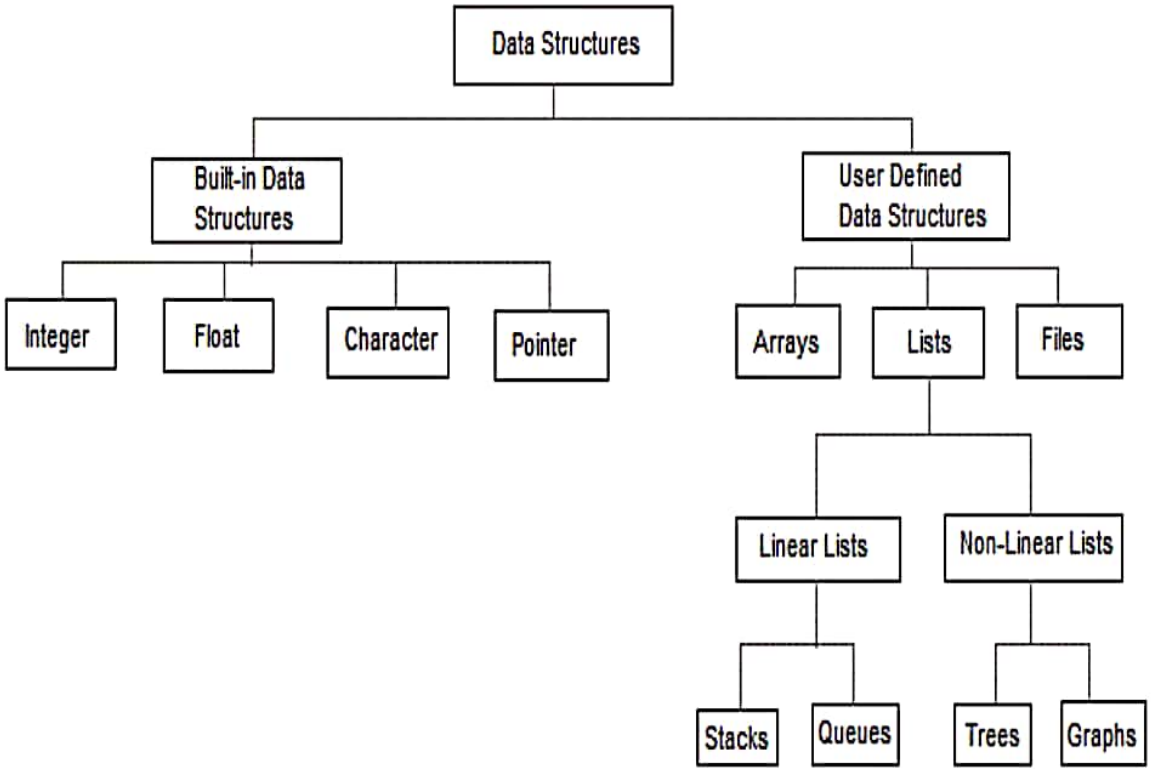
Basic types of Data Structures

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



The data structures can also be classified on the basis of the following characteristics:

Characterstic	Description
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: **Array**
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: **Tree**, **Graph**
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: **Array**
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: **Structures**
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: **Array**
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: **Linked List created using pointers**

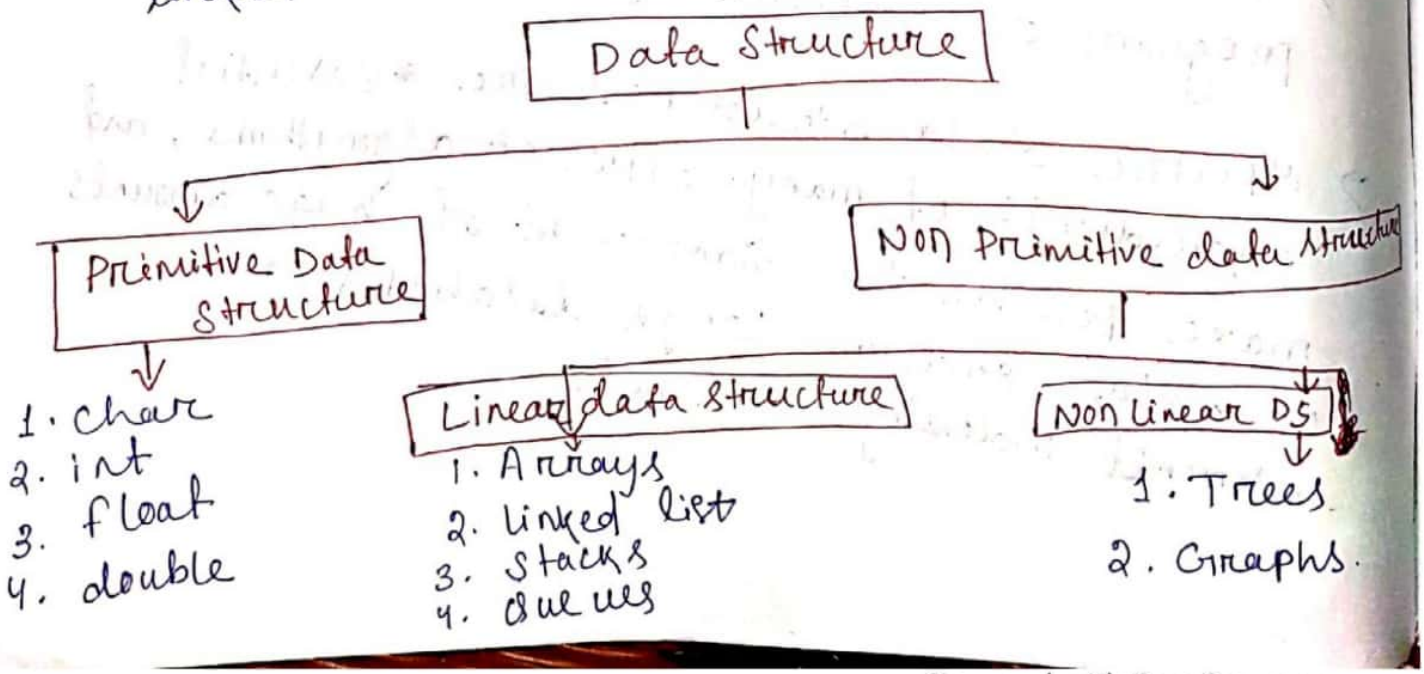
→ Some formal design methods and programming languages emphasize data structures, rather than algorithms as the key organizing factor in software design.

Types of Data structures :-

→ Data structures are generally categorized into two classes : primitive and non primitive datastructure.

→ Primitive data structures are the fundamental data types which are supported by a programming language.
Some basic data types are :- Char, int, float, and double.

→ Non Primitive data structures are those data structures which are created using primitive data structures.
Examples of such data structures includes linked list, stacks, trees and graphs.



Scanned with CamScanner

→ Non primitive data structures can further be classified into two categories:
Linear and
Non linear data structures.

→ A data structure that maintains a linear relationship between its elements, it is called linear data structure.

→ For example an array holds the linear relationship between its elements, it is linear data structure.

→ In case of non-linear data structure, they maintain hierarchical relationship between their elements. Consider a tree structure. You can't define linear relationship between the elements. It is an example of non-linear data structure.

Linear Data Structure :-

In linear data structure the elements are stored in sequential ~~order~~ order. The linear data structures are :-

* Array :-

Array is a collection of data of same data type stored in consecutive memory location and is referred by common name.

Scanned with CamScanner

Linked List :-

Linked list is a collection of data of same data type but the data items need not be stored in consecutive memory locations.

Stack :-

→ A stack is a last-in-first-out linear data structure in which insertion and deletion takes place at only one end called the top of the stack.

Queue :-

→ A queue is a first-in-First-out linear data structure in which insertions takes place one end called the rear and the deletions takes place at one end called the Front.

Non-linear Data-Structure :-

→ Elements are stored based on the hierarchical relationship among the data.
The following are some of the non linear data structures

Trees :-

Trees are used to represent data that has some hierarchical relationship among the data elements.

Scanned with CamScanner

Graph:-

Graph is used to represent data that has relationship between pair of elements not necessarily hierarchical in nature.

Operations on Data Structure:-

The different operations that can be performed on the various data structures are:-

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

Problems can be solved using Data Structure:-

The following ~~prob~~ computer problems can be solved using Data Structure.

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-warshall
- Shortest path by Dijkstra
- Project Scheduling.

Scanned with CamScanner

5th Sep. 2020

Array:-

Array is a container which can hold a fix number of ~~fixed~~ items and these items should be of the same type.

Element:- Each item stored in an array is called an element.

Index:- Each location of an element in an array has a numerical index, which is used to identify the element.

Base address:- The starting address of the array.

Array Representation:-

→ Arrays can be declared in various ways in different languages.

Operations on array:-

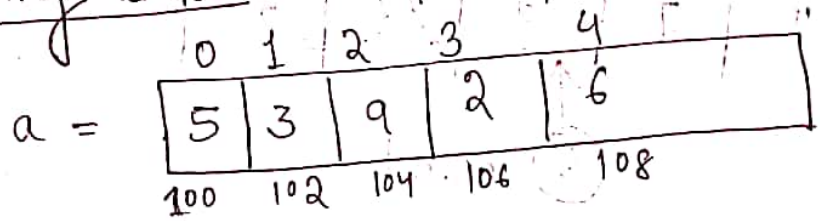
1. Traversing:- Means to visit all the elements of the array in an operation is called traversing.
2. Insertion:- Means to put values into an array.
3. Deletion:- Remove to delete a value from an array
4. Sorting:- Re arrangement of values in an array in a specific order (Ascending/Descending) is called Sorting.

Scanned with CamScanner

5. Searching :-

The process of finding the location of a particular element in an array is called searching. There are two popular searching techniques Linear Search & Binary Search.

Traversing Example :-




```
int main ( )
{
 int a[5];
 int i;
 printf ("Enter Element")
 for (i=0; i<=4 ; i++)
 {
 scanf ("%d", & a[i]);
 }
 printf ("The elements are");
 for (i=0; i<=4 ; i++)
 {
 printf ("%d," a[i]);
```

Output :-

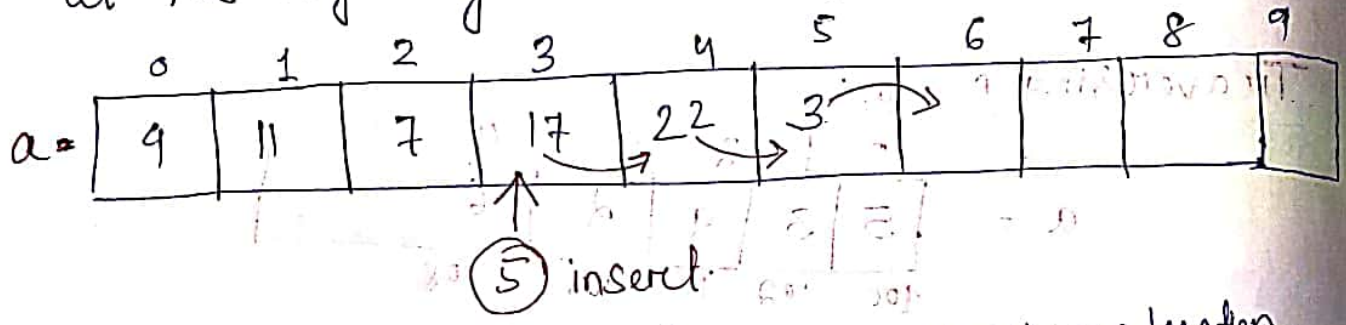
a[0] = 5
a[1] = 3
a[2] = 9
a[3] = 2
a[4] = 6

Scanned with CamScanner

Insertion in array:-

→ Insertion operation is to insert one or more data elements into an array.

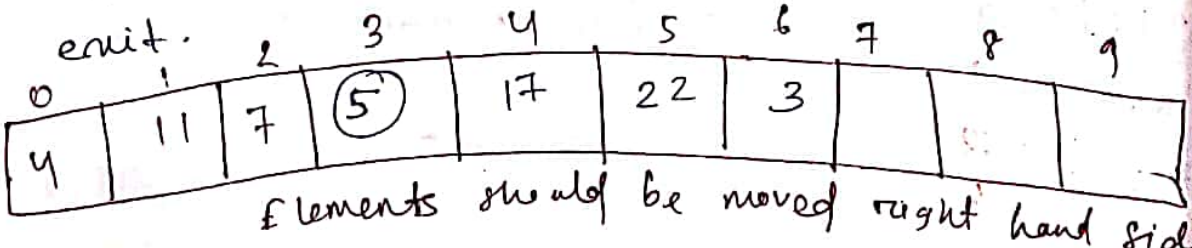
→ Based on requirement, a new element can be added at the beginning, end or any given index of array.



Here MaxLoc = 10
n = 6
loc = 4
item = 5

MaxLoc - Maximum location available in array.
n = no of elements available in array.
loc = The position where we have to insert.
item = which element we want to insert.

```
array insert (a, maxloc, n, loc, item)
if (max loc == n)
{printf ("overflow");
 exit
}
for (i=n ; i>= loc ; i--)
{
 a[i] = a[i-1]
 a[loc] = item
 n = n+1
 exit.
```



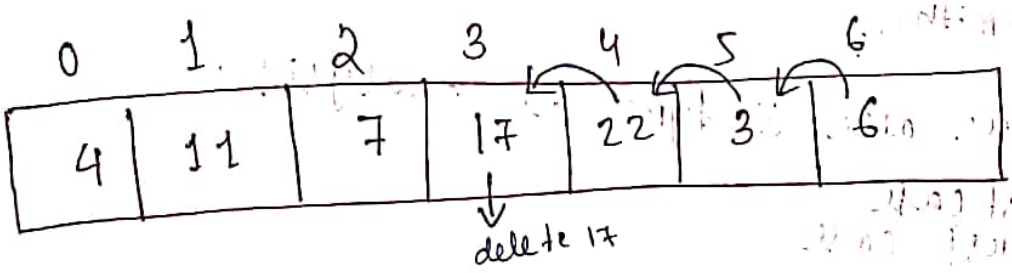
Elements should be moved right hand side.

Scanned with CamScanner

Deletion :-

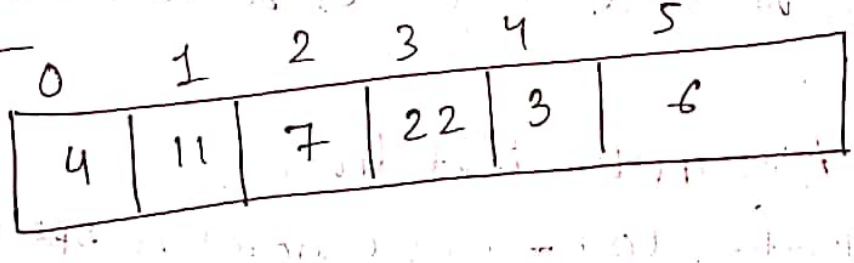
Deletion refers to moving an existing element from the array and re-organizing all elements of an array.

eg :-




```
array delete (a, max_loc, n, item, loc)
if (n==0)
{
print "underflow";
exit
}
item = a[loc];
for (i=(loc-1); i<=(n-2); i++)
a[i] = a[i+1]
n = n-1
exit
```

output



Scanned with CamScanner

Algorithm

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudo code** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be at least 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** Its the space required to store all the constants and variables(including temporary variables) value.
- **Environment Space:** Its the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible.

Asymptotic Notations

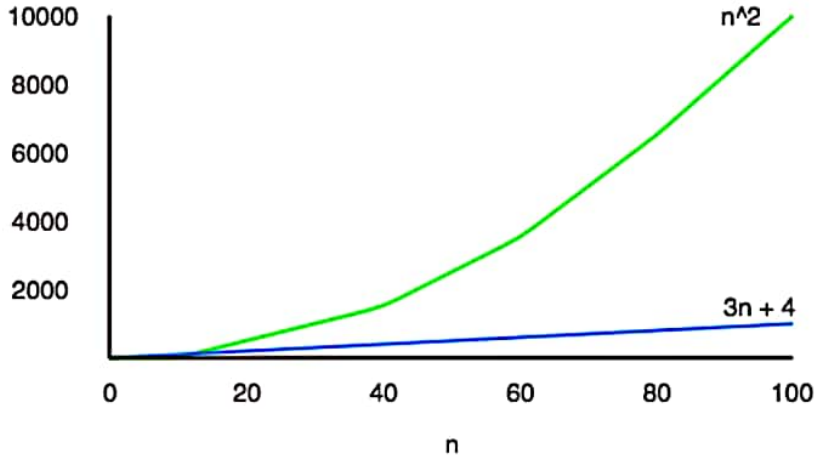
When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

Let us take an example,

if some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation.

For large values of n, the $3n + 4$ part will become insignificant compared to the n^2 part.



For $n = 1000$, n^2 will be 1000000 while $3n + 4$ will be 3004.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of $200n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 200. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

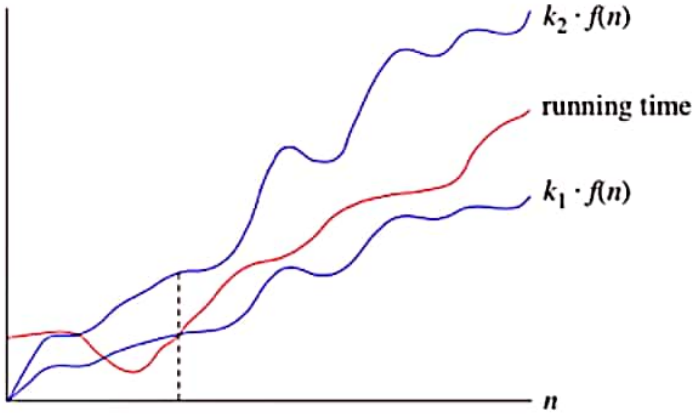
1. **Big Theta (Θ)**
2. **Big Oh(O)**
3. **Big Omega (Ω)**

Tight Bounds: Theta

When we say tight bounds, we mean that the time complexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the average time for any input n will remain in between, k1 * n^2 and k2 * n^2, where k1, k2 are two constants, thereby tightly binding the expression representing the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In **Worst case**, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of n, where n represents the number of total elements.

But it can happen that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to n, as this is the right way to represent the average time complexity, but when we use the big-O

notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it makes more sense.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take at least this much time to complete its execution. It can definitely take more time than this too.

Asymptotic Notation

→ Asymptotic notations are the expressions that are used to represent the complexity of an algorithm.

→ There are 3 types of analysis :-

Best case
worst case
Average case.

* Best case :- In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.

* worst case :-
In which we analyse the performance of an algorithm for the input, for which the algorithm takes ~~tog~~ long time or space.

* Average case :-
In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

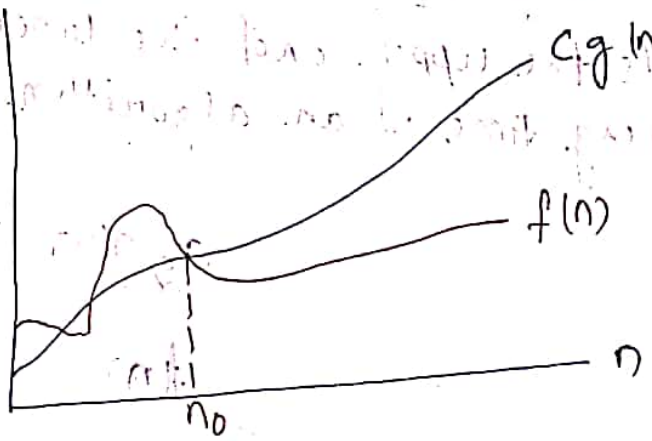
Types of Asymptotic Notation :-

1. Big-O Notation (O) – Big O notation specifically describes worst case scenario.
2. omega Notation (Ω) :- Omega (Ω) notation Specifically describes best case scenario.
3. Theta Notation (θ) :- This notation represents the average complexity of an algorithm

Scanned with CamScanner

Big-O Notation - (O)

→ It represents the upper bound of the running time of an algorithm.



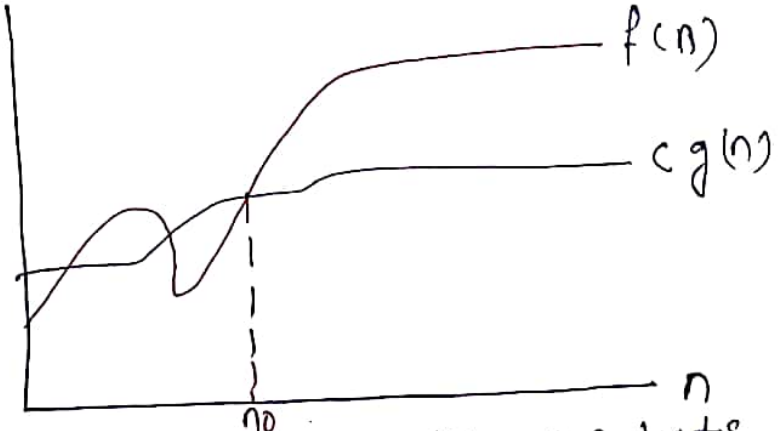
$f(n) = O(g(n))$

$O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0.
Such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0 \}$

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

Omega Notation (Ω)

→ Omega Notation represents the lower bound of the running time of an algorithm.



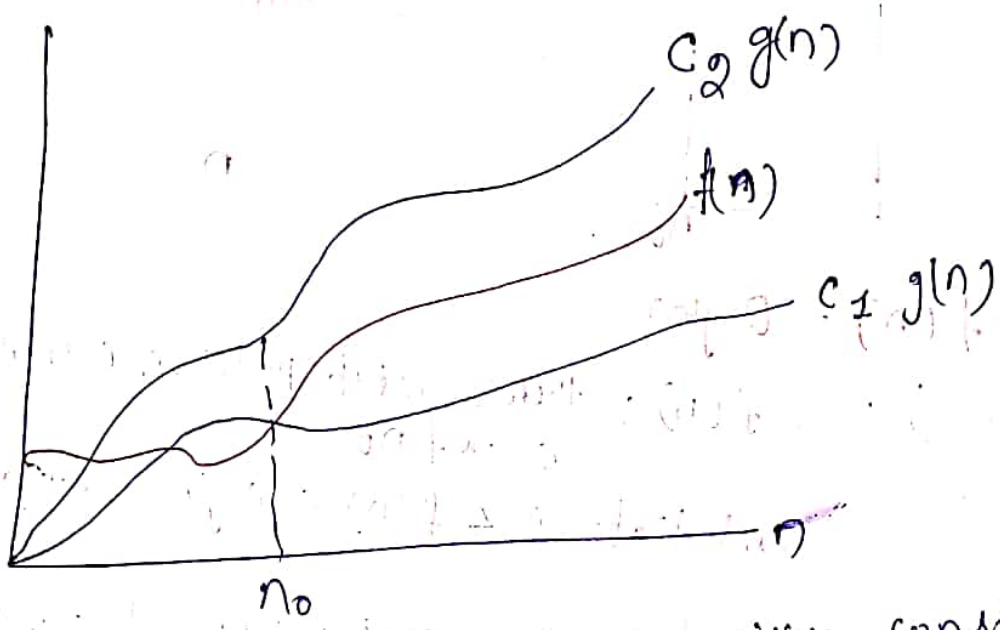
$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0
Such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0 \}$

Scanned with CamScanner

→ For any value of n, the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

Theta Notation :- (θ)

→ It represents the upper and the lower bound of the running time of an algorithm,



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

Scanned with CamScanner

List of some common asymptotic notations:-

Constant – $O(1)$
Logarithmic – $O(\log n)$
Linear – $O(n)$
nlogn – $O(n \log n)$
quadratic – $O(n^2)$
Cubic – $O(n^3)$
polynomial – $n^{O(1)}$
exponential – $2^{O(n)}$

Scanned with CamScanner

Searching Algorithms

There are two popular algorithms available:

1. **Linear Search**
2. **Binary Search**

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

Searching Algo - Linear Search

	0	1	2	3	4	5	6	7
a	15	5	20	35	2	42	67	17

n = 8

data = 42
data = 41

```
for (i=0; i<n; i++)
{
if (a[i] == data)
{
printf ("Element found at index: %d", i);
Break;
}
}

if (i == n)
{
printf ("Element not found");
}
```

Time complexity :-

Best case :- O(1)

worst case - O(n)

Avg case :- $\frac{\Sigma \text{ all cases}}{\text{no. of cases}} = \frac{1+2+3 \cdots n}{n}$

$= \frac{n(n+1)}{2} \cdot \frac{1}{n}$

$= \left(\frac{n+1}{2}\right)$

Scanned with CamScanner

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

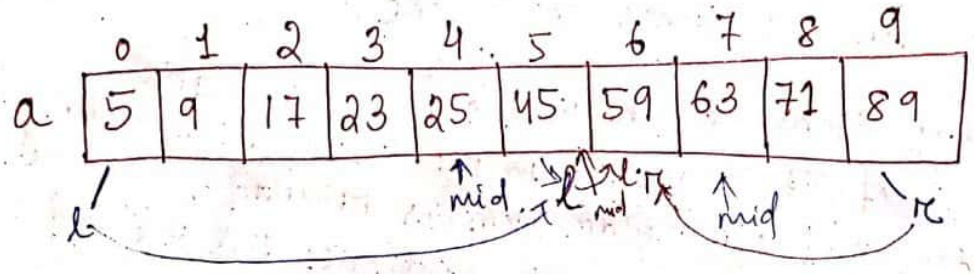
Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. It has a simple implementation.

Binary Search



l	r	mid = $\left\lfloor \frac{l+r}{2} \right\rfloor$ floor value.
0	9	9/2 = 4.5 = (4)
5	9	14/2 = (7)
5	6	11/2 = 5.5 = (5)
6	6	12/2 = (6)

case 1:- data = a[mid]
case 2:- data < a[mid]
case 3:- data > a[mid]

```
Binary search (a, n, data)
{
l = 0, r = n-1
while (l<r)
mid = (l+r)/2;
if (data == a[mid])
    return mid;
else if (data < a[mid])
    r = mid-1;
else
    l = mid+1;
}
return -1
```


Scanned with CamScanner