Unit-1

Data Structures (CIC-209)
Dr. Nivedita Palia

UNIT – I

Overview of data structure, Basics of Algorithm Analysis including Running Time Calculations, Abstract Data

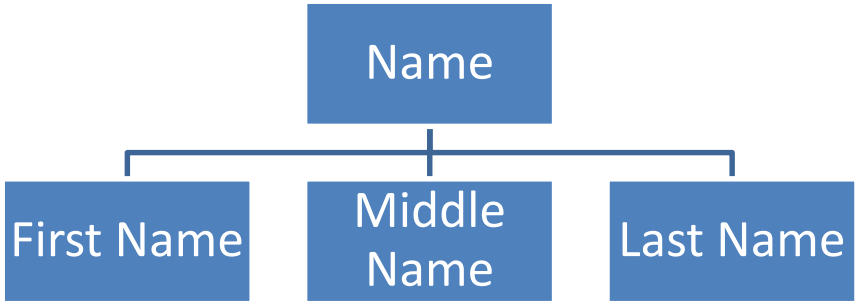
Types, Arrays, Arrays and Pointers, Multidimensional Array, String processing, General Lists and List ADT, List

manipulations, Single, double and circular lists. Stacks and Stack ADT, Stack Manipulation, Prefix, infix and

postfix expressions, recursion. Queues and Queue ADT, Queue manipulation.

Overview of Data Structures

Data : values or Set of values.
Data item : Single unit of values. It may be *group item* or *elementary item*. For example



Data Type: Collections of values and set of operations on those value

Data Structure

It is the logical or mathematical model of particular organization of data in such a way that we can perform operations on these data in an **effective way.** Depending on your requirement choose right data structure. For eg: name is stored as string where as employee id as number.

Need of Data Structures

Data structure provides a way of organizing, managing, and storing data efficiently. With the help of data structure, the data items can be traversed easily. Data structure provides

Efficiency: It make program in terms of time and space complexity.

Operation on Data Structures

- **Creation** : Declaration and initialization of the data structures and reserved memory location for data elements.
- **Traversal**: accessing or visiting each data item exactly once.
- **Searching**: finding the data item within the data structure which satisfies searching condition
- **Insertion**: adding a new data element within the data structure
- **Deletion**: removing a new data element from the data structure
- **Sorting**: arranging the data in some logical order
- **Merging**: combining the data elements of two data structures
- **Updating**: Changes data values of the data structure

Types of Data Structures in C

Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).

In C, arrays are declared using the following syntax:

```
type name[size];
```

For example,

```
int marks[10];
```

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Linked Lists

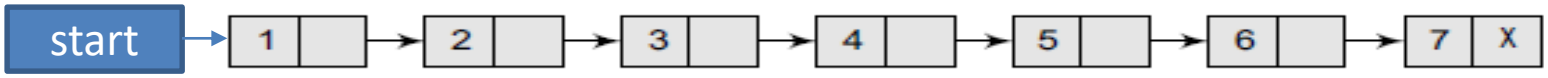
A linked list is a data structure in which elements (called nodes) form a sequential list. Every node in the list points to the next node in the list.

Therefore, in a linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list.

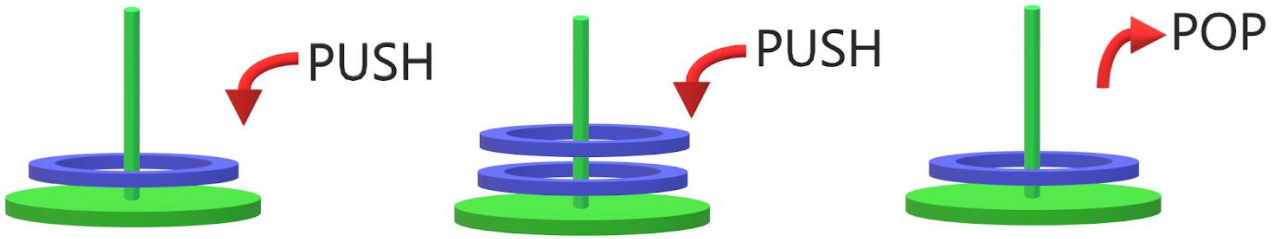
The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

Start is pointer variable which stores the address of first node in the list.



Stacks

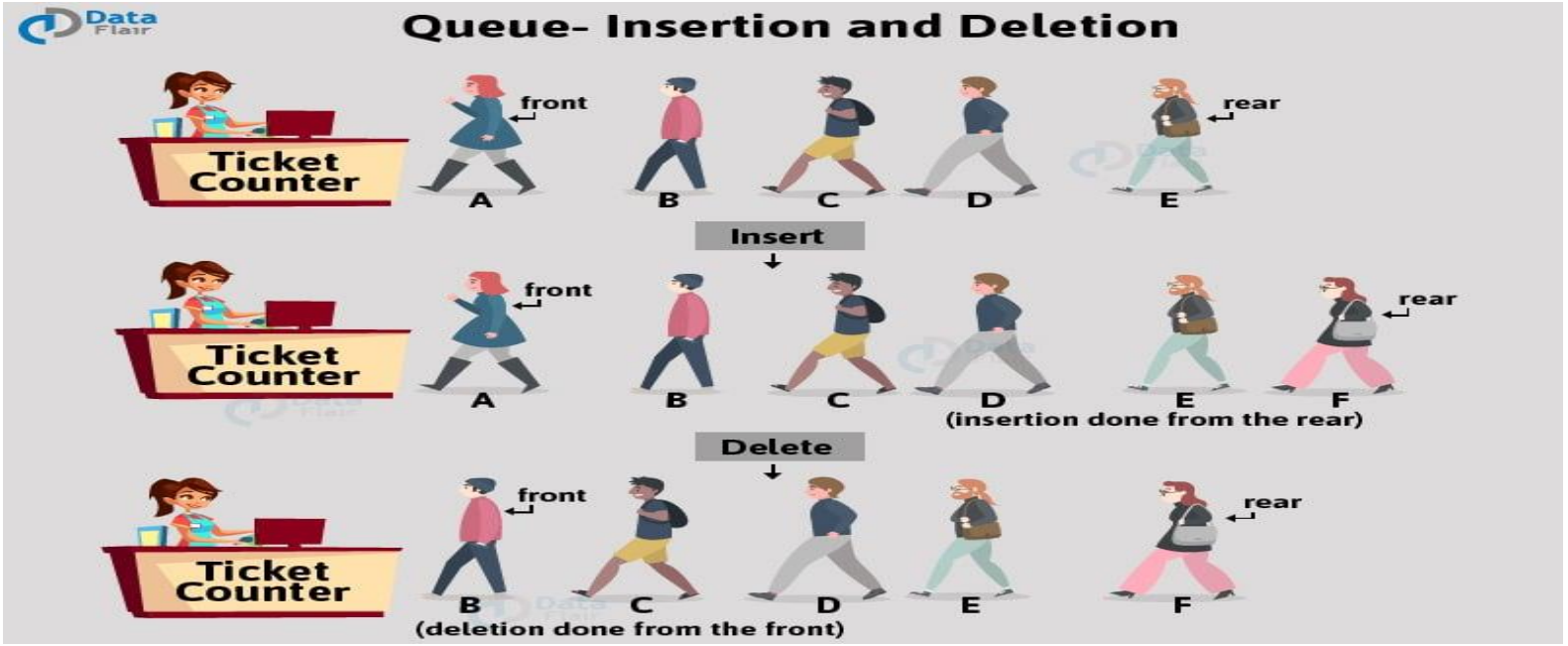
A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.



Simple Representation of a Stack with PUSH and POP operations

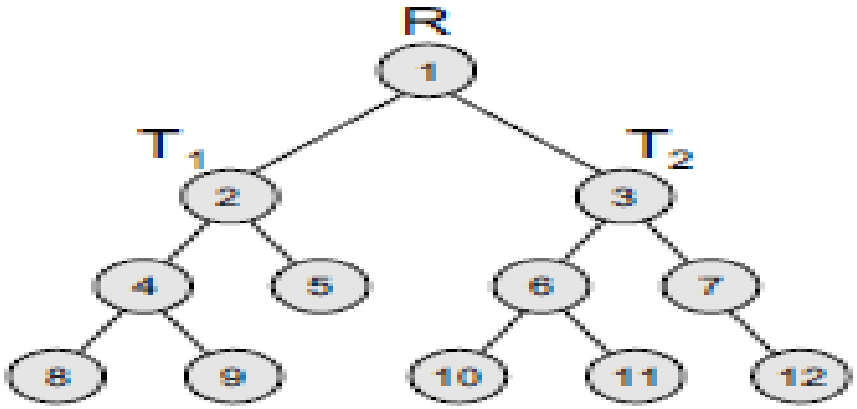
Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the `rear` and removed from the other end called the `front`. Like stacks, queues can be implemented by using either arrays or linked lists.



Trees

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

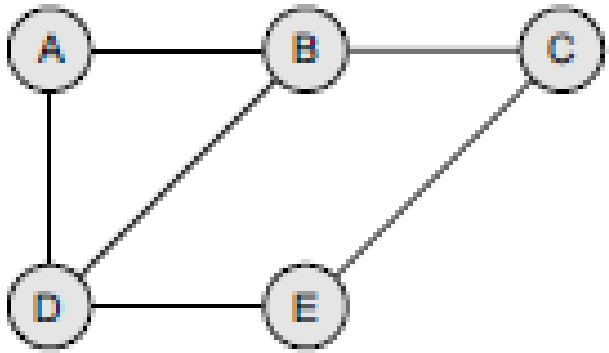


Graph

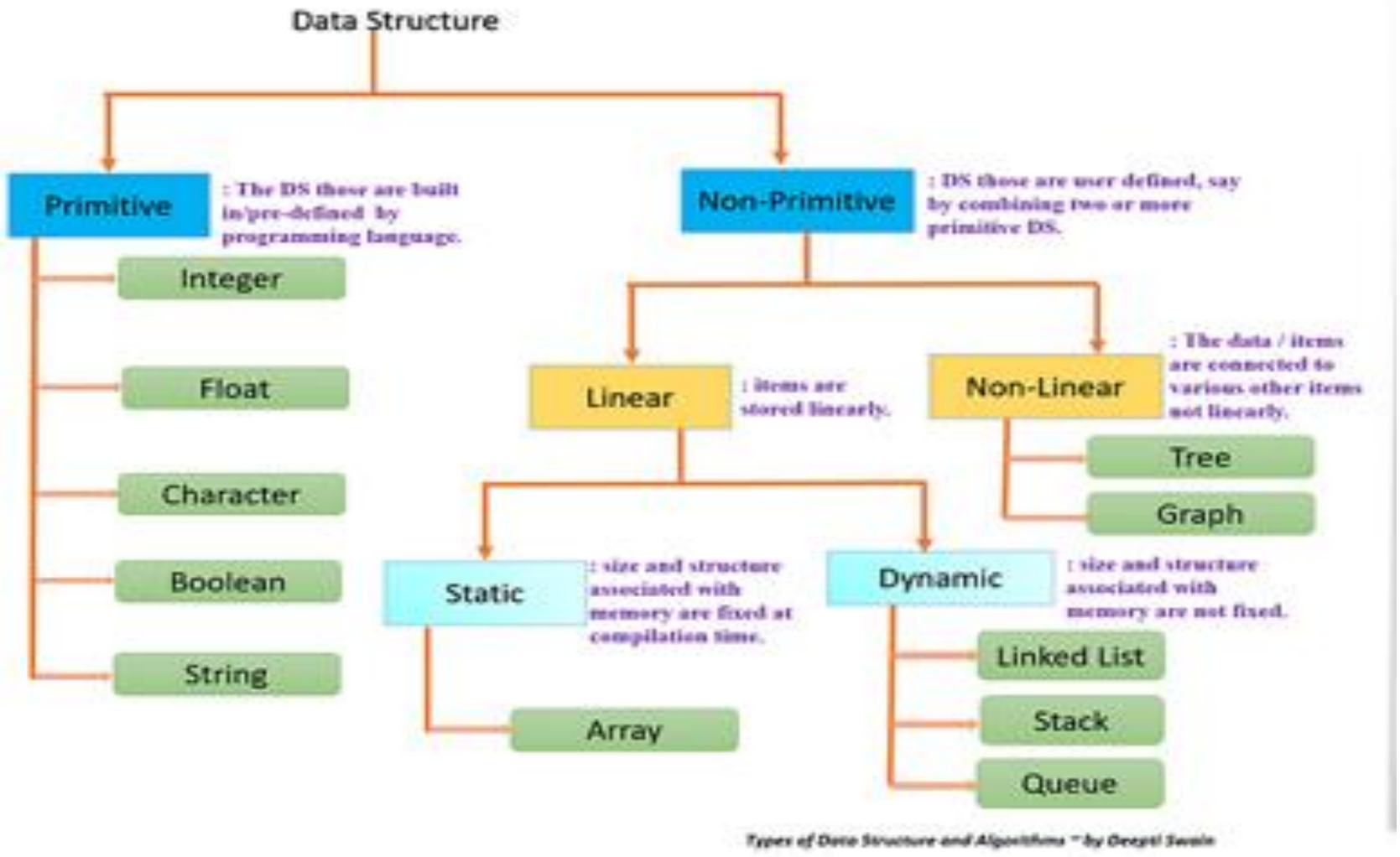
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines that connect any two nodes in the graph.

Or

Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).



Classification of Data Structure



Primitive DS VS Non-Primitive DS

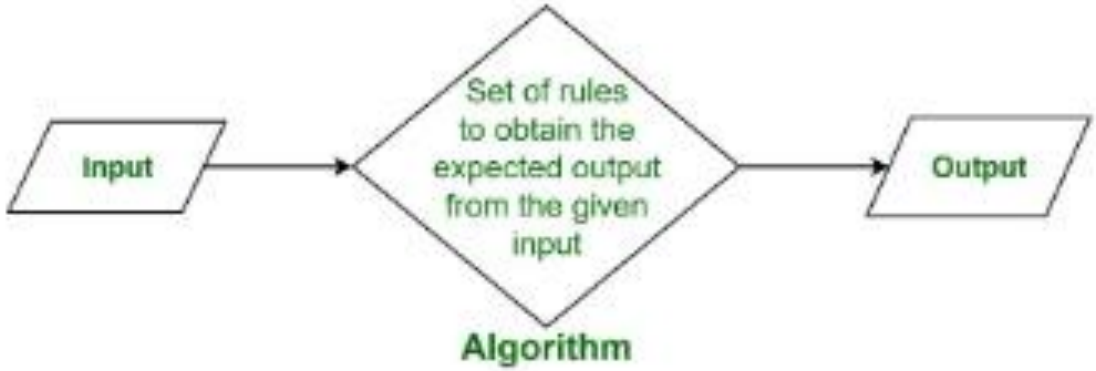
Primitive data structure	Non Primitive Data structure
Primitive data structure is the data structure that allows you to store only single data type values.	Non-Primitive data structure is a data structure that allows you to store multiple data type values.
integer, character, float, etc. are some examples of primitive data structures.	Array, Linked List, Stack, etc. are some examples of non-primitive data structures.
Primitive data structure always contains some value i.e. these data structures do not allow you to store NULL values.	You can store a NULL value in the non-primitive data structures.
The size of the primitive data structures is dependent on the type of the primitive data structure.	The size of the non-primitive data structure is not fixed.

Linear VS Non-Linear Data Structures

	Linear Data structure	Non-Linear Data structure
Basic	In this structure, the elements are arranged sequentially or linearly and attached to one another.	In this structure, the elements are arranged hierarchically or non-linear manner.
Types	Arrays, linked list, stack, queue are the types of a linear data structure.	Trees and graphs are the types of a non-linear data structure.
implementation	Due to the linear organization, they are easy to implement.	Due to the non-linear organization, they are difficult to implement.
Traversal	As linear data structure is a single level, so it requires a single run to traverse each data item.	The data items in a non-linear data structure cannot be accessed in a single run. It requires multiple runs to be traversed.
Arrangement	Each data item is attached to the previous and next items.	Each item is attached to many other items.
Levels	This data structure does not contain any hierarchy, and all the data elements are organized in a single level.	In this, the data elements are arranged in multiple levels.

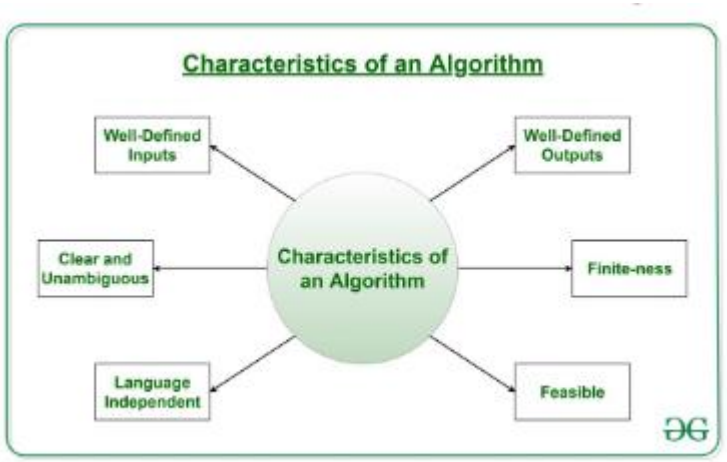
Algorithm

- An Algorithm is any well-defined computational procedure that takes some values, or set of values as input and produces some value, or set of values , as output.
- It is independent of the programming language.



Characteristics of Algorithm

Clear and Unambiguous: The algorithm should be 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.



Algorithm vs Pseudocode vs Program:

While algorithms are generally written in a natural language or plain English language, pseudocode is written in a format that is similar to the structure of a high-level programming language. Program on the other hand allows us to write a code in a particular programming language.

Example: Addition of two numbers

Algorithm	Pseudocode	Program

1. Start
2. Enter two numbers
3. Sum them and save result
4. Print sum
5. End
	1. Start
	2. Input a and b
	3. Sum=a+b
	4. Display Sum
	5. End
		int main()
		{
		int a, b, sum;
		scanf(“%d%d”,&a,&b);
		sum=a+b;
		printf(“%d”,sum);
		return (0);
		}

Types of Algorithm

Brute Force Algorithm: most basic and simplest type of algorithm. It is just like iterating every possibility available to solve that problem.

Recursive Algorithm :based on recursion. In recursion, a problem is solved by breaking it into sub-problems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

Factorial of a Number

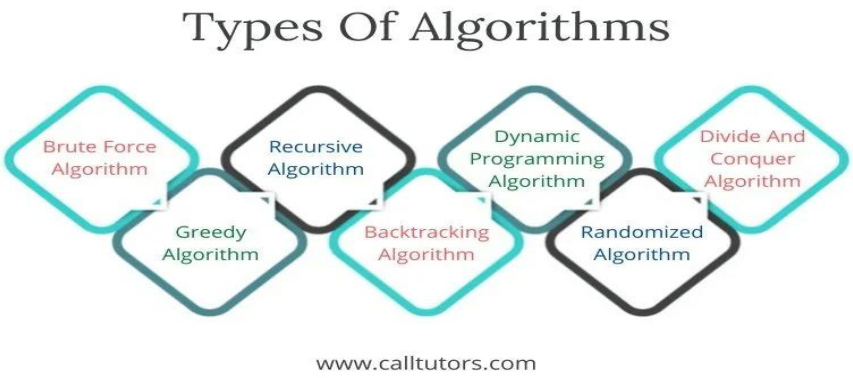
Divide and Conquer Algorithm: In this to solve the problem in two sections, the first section divides the problem into sub-problems of the same type. The second section is to solve the smaller problem independently and then add the combined result to produce the final answer to the problem. Binary Search

Dynamic Programming Algorithms: In Dynamic Programming, divide the complex problem into smaller overlapping sub-problems and store the result for future use. Floyd Warshall Algorithm

Greedy Algorithm: The decision to choose the next part is done on the basis that it gives an immediate benefit. It never considers the choices that had been taken previously. Dijkstra Shortest Path Algorithm

Backtracking Algorithm: . it is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time. Hamiltonian Cycle

Randomized Algorithm: In the randomized algorithm, we use a random number.it helps to decide the expected outcome. The decision to choose the random number so it gives the immediate benefit. Quicksort



Analysis of Algorithm

- Performance Analysis (machine independent)

  – space complexity: is the amount of memory it needs to run to completion.

  – time complexity: is the amount of computer time it needs to run to completion.
- Performance Measurement (machine dependent)
- Performance analysis is also called as Priori Estimates
- Performance measurement is also called as Posteriori Testing
- The complexity of the algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)

Independent of the characteristics of the inputs and outputs

– instruction space

– space for simple variables, fixed-size structured variable, constants

- Variable Space Requirements ($S_P(I)$)

depend on the instance characteristic I

– number, size, values of inputs and outputs associated with I

– recursive stack space, formal parameters, return address

Algorithm 1: Simple arithmetic function

```
Algorithm abc(a, b, c)
{
return a + b + b * c + (a + b - c) / (a + b);
}
```

$S_{abc}(n)=3+0=3$ $S_{abc}(I) = 0$

Algorithm 2 : Iterative function for summing a list of numbers

```
Algorithm Isum(list, n)
{
sum = 0;
for (i = 0; i<n; i++)
sum += list [i];
return sum;
}
```

7

$S_{sum}(I) \geq (n+3)$

n for list and one for each sum, i, n

Time Complexity

- **Time complexity** is the computational complexity that describes the amount of time it takes to run an algorithm.
- Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm
- We Assume that each elementary operation takes a fixed amount of time to perform.
- Thus, the amount of time taken is the total number of elementary operations performed by the algorithm

Time Complexity

T(P)=C+Tp(I)

Compile Time (C) : independent of instance characteristics

Run Time /Execution Time: Tp

Example 1 (Time Complexity)

Statement	s/e	Frequency	Total steps
Algorithm sum(list, n)	0	0	0
{	0	0	0
s := 0;	1	1	1
i:=0;	1	1	1
for i := 1 to n do	1	n+1	n+1
s:= s+ list[i];	1	n	n
return s;	1	1	1
}	0	0	0
Total			2n+4

Example 2 (Time Complexity)

Statement	s/e	Frequency	Total steps

Algorithm add (a, b, c, m,n)
{
 	
for i:= 1 to m do
for j:= 1 to n do
c[i][j] := a[i][j] + b[i][j];
}
	0
	0
	
	1
	1
	1
	0
		0
		0
		
		m+1
		m • (n+1)
		m • n
		0
			0
			0
			
			m+1
			mn+m
			mn
			0

Total			2mn+2m+1

Complexity for the Linear Search

Linear Search algorithm

1. [Intialize] Set K:=1 , LOC:= 0
2. Repeat Step 3 & 4 while LOC:=0 & K<=N
3. If ITEM = DATA[K], then: Set LOC:=K
4. Set K:=K+1 [Increment counter]
   [End of Step 2 loop]
5. [LOC found?]
   If LOC=0, then:
   Write: ITEM not found
   Else
   Write: LOC is location of ITEM
   [End of If structure]
6. Exit

Complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data.

- The complexity of the search algorithm is given by the number C of comparison between item and Data[K]
- Best Case: When item found at first position.

$$C(n)=1$$

- Worst Case: When item is the last element in the array Data or it is not there at all.

$$C(n)=n$$

- Average Case : Item does appear in the Data and that it is equally likely to occur at any position. Accordingly , the number of the comparisons can be any of the number 1,2,3…,n and each number occurs with probability p=1/n. Then

$$\begin{aligned} C(n) &= 1\cdot\frac{1}{n}+2\cdot\frac{1}{n}+\cdots+n\cdot\frac{1}{n} \\ &= (1+2+\cdots+n)\cdot\frac{1}{n} \\ &= \frac{n(n+1)}{2}\cdot\frac{1}{n}=\frac{n+1}{2} \end{aligned}$$

$$C(n)=n/2$$

Asymptotic Notations

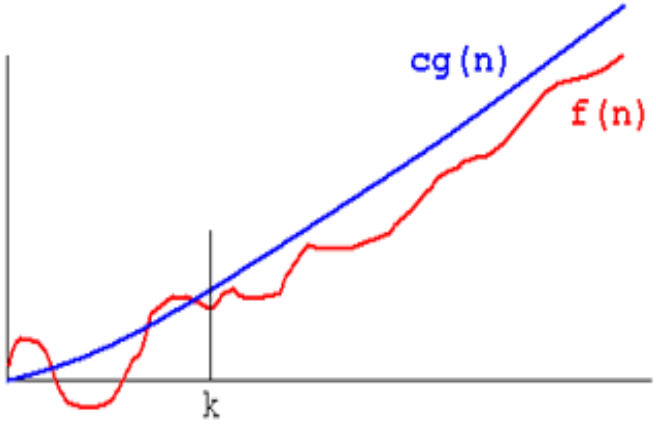
- Big Oh (O) Worst Case Analysis
- Big Omega (Ω) Best Case analysis
- Theta (Θ) Notation= Average Case Analysis

Big O Notation (worst case)

- Big O of a function gives us **'rate of growth'** of the step count function *f(n)*, in terms of a simple function *g(n)*, which is easy to compare.

$f(n) = O(g(n))$
iff there exist positive constants c and n_0 such that
$f(n) <= c*g(n)$ for all n, $n>=n_0$.

$f(n) <= c*g(n)$ for all n, $n>=n_0$.



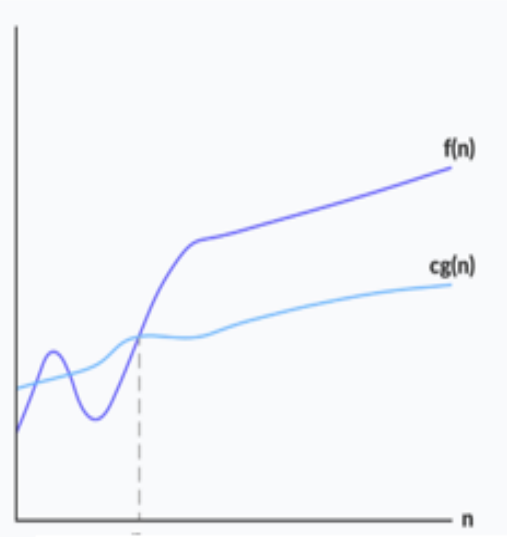
Example:2
Given $f(n) = 10n^2+4n+2$
Can we write $O(n^2)$??
If we can prove $f(n) <= c*g(n)$ then we can write O(g(n)) or $O(n^2)$
Yes, because $10n^2+4n+2 <= 11n^2$ for all **n >=5.**
Therefore the complexity of f(n) is O(n2)

Example:1 f(n) = 3n+2 is O(n)
because 3n+2 <= 4n for all
n >= 2. c = 4, n_0 = 2. Here g(n) = n.

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.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
- Omega gives the lower bound of a function

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



Example

Consider the following f(n) and g(n)...

$f(n) = 3n + 2$ **$g(n) = n$**

If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0 >= 1**

$f(n) >= C\ g(n)$

$\Rightarrow 3n + 2 >= C\ n$

Above condition is always TRUE for all values of **C = 1** and **n >= 1**.

By using Big - Omega notation we can represent the time complexity as follows...

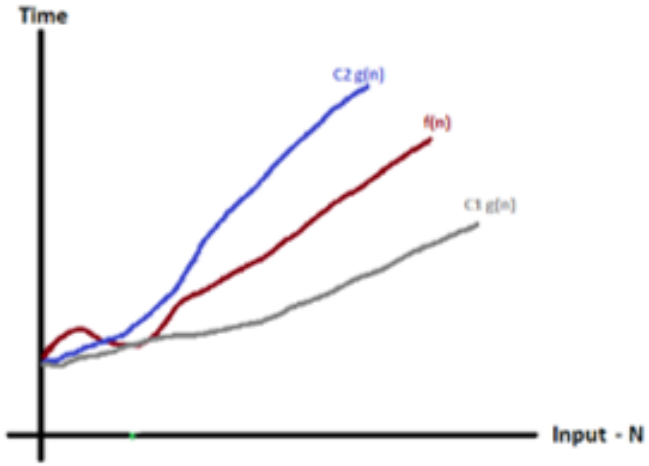
$3n + 2 = \Omega(n)$

Big - Theta Notation (Θ)

- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- Theta notation always indicates the average time required by an algorithm for all input values.
- Big - Theta notation describes the average case of an algorithm time complexity.

- Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.
- If $C_1\ g(n) <= f(n) <= C_2\ g(n)$ for all $n >= n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 >= 1$. Then we can represent f(n) as Θ(g(n)).
- $f(n) = \Theta(g(n))$

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



Example

Consider the following f(n) and g(n)...

$f(n) = 3n + 2$

$g(n) = n$

If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy

$C_1\ g(n) <= f(n) <= C_2\ g(n)$ for all values of $C_1 > 0$, $C_2 > 0$ and $n_0 >= 1$

$C_1\ g(n) <= f(n) <= C_2\ g(n)$

$\Rightarrow C_1\ n <= 3n + 2 <= C_2\ n$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n >= 2$.

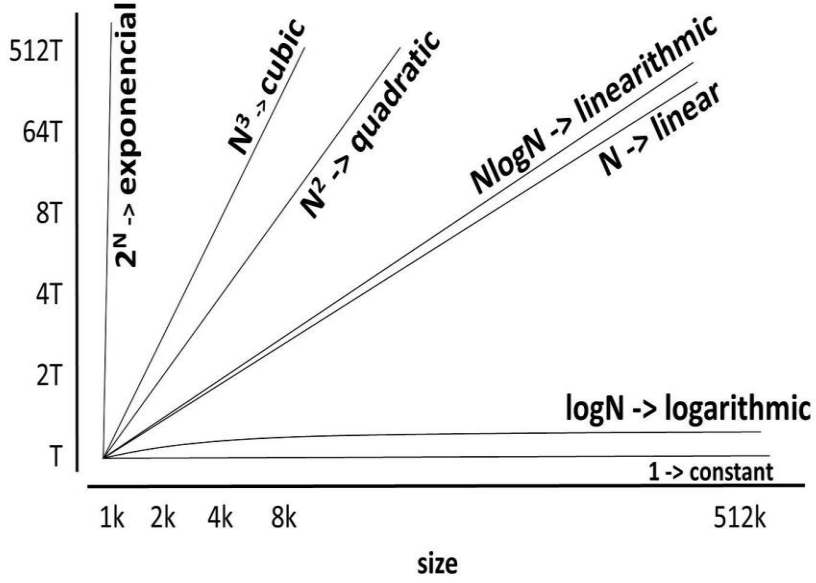
By using Big - Theta notation we can represent the time complexity as follows...

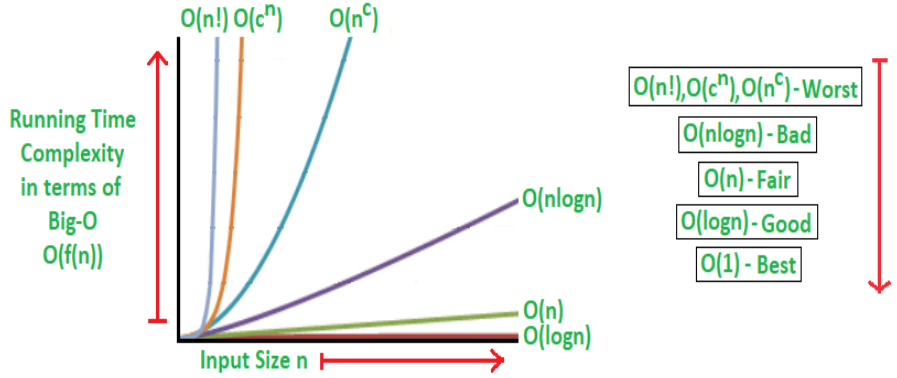
$3n + 2 = \Theta(n)$

VIPS
Vivekananda Institute of Professional Studies
योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

Rate of Growth

Order of Growth	Description
1	Constant
log n	Logarithmic
n	Linear
n log n	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential





g(n) / n	log n	n	n log n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

Abstract Data Type

- An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.
- In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details.
- The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.
- **For example,** a List is an abstract data type that is implemented using a dynamic array and structure. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue

END