Unit I : Introduction to data structures

Data:

Data is a set of values or facts and statistics collected for reference or the for the purpose of analysis.

Data object:

Is a container for data values for storage and easily retrievable. A data object is characterized by a set of attributes such as data type.
The attributes give information about number and type of values that the data object may contain and logical organization of these values.

A data object is a run-time instance of data structures

SET_OF_ALphabets= {*A*, *B*, …, *Z*, *a*, *b*, …, *z*}

I = {10,20,30,40, 50,60,70,80,90,100}

1) **Programmer defined data objects:**
Data objects that exist during program execution (variables, constants, arrays, and files.)
Programmer creates and manipulates these data objects through declarations and statements in the program.

2) System defined data objects
Generated automatically as needed during the execution of program

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.

Need of data structure

1. Data Structure provides a way to store and retrieve data to and from memory efficiently.
2. It provides ways to design and implement data structures.
3. Data structure gives the knowledge of
   How to organize data?
   How to control flow of data?
   How to design and implement efficient Data Structure
   How to reduce complexity of algorithm?
   How to increase efficiency of algorithm?

Data Structure

A data structure is a class of data that can be characterized by its organization and the operations that are defined on it.

Data Structure = Organized Data + Allowed Operations

A Data structure is a set of values along with the set of operations permitted on them.

Data structure (D)

Data structure is a triplet D=(d,F,A) where

D - set of range of values that data may have.

F- set of operations which can be applied to the elements of data

A - set of rules with which different operations belonging to F can be implemented

Abstract Data Type

It is an encapsulation of the data and the operations on this data and hide them from the user.

ADT is declaration of data, implementation of operations, and encapsulation of data and operations.

An abstract data type (**ADT**) is a logical description or a specification of components of the data and the operations that are allowed, which is independent of the implementation.

Types of ADT

Built-in types : boolean, integer, real, array

User-defined types: stack, queue, tree, list

Advantages of ADT

1. Encapsulation
2. Flexible
3. Reusable
4. Easy to understand

Integer ADT

Integers (whole numbers greater or equal to 0).

Methods:

`Integer add(Integer x, Integer y)`

Postcondition: returns the sum of `x` and `y`

`Integer Sub(Integer x, Integer y)`

Postcondition: returns the subtraction of `x` and `y`

`Integer multiply(Integer x,Integer y)`

Postcondition: returns `x` times `y`

`boolean equals(Integer x, Integer y)`

Postcondition: returns true of `x` equal to `y`

1. **Write an ADT for LIST**
2. **Write an ADT for employee data**
3. **Write an ADT for student data**

```
#include <iostream>
using namespace std;
 class Adder
{ public:
    Adder(int i = 0)
    { total = i; }
 void addNum(int number)
 { total += number; }
 int getTotal()
 { return total; };
 private:  int total;
 };
 int main( )
 { Adder a;
  a.addNum(10);  a.addNum(20);
 a.addNum(30);
 cout << "Total " << a.getTotal() <<endl;
 return 0; }
```

Output:

?

Classification of Data Structure

Primitive and Non-Primitive Data Structure

Linear and Non-Linear Data Structure

Homogenous and Non-Homogeneous Data Structure

Static and Dynamic Data Structure

Primitive and Non-Primitive Data Structure

The data structure that are atomic (indivisible) are called primitive.

Example: integer, real, Boolean and characters.

The data structure that are not atomic are called non-primitive or composite.

Example : records, array and string.

Linear and Non- Linear Data Structure

In a linear data structure, the data items are arranged in a linear sequence.

Example is array.

In a non-Linear data structure, the data items are not in a sequence.

Example is tree.

Homogeneous and Non- Homogenous Data Structure

In Homogeneous Structure, all the elements are of same type.
Example : arrays.

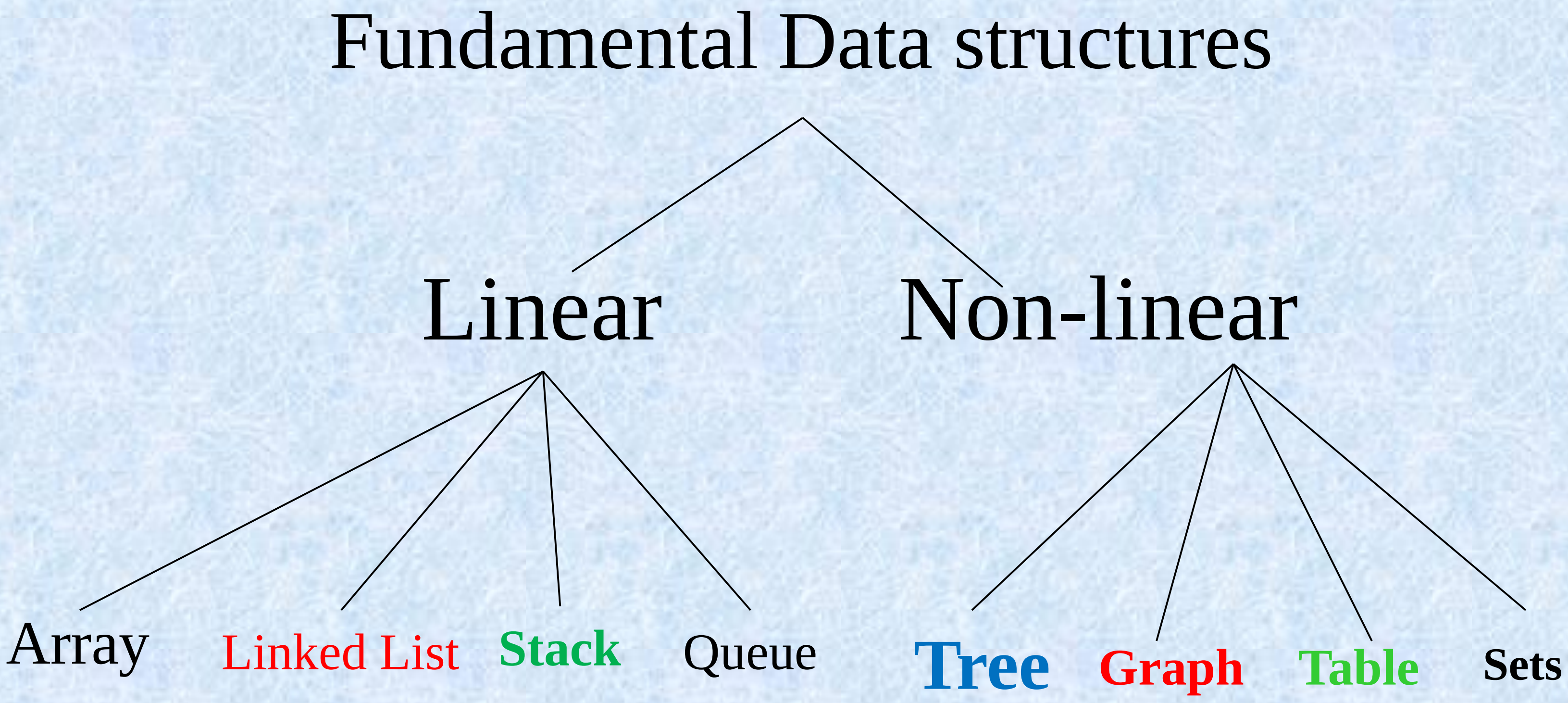
In Non-homogeneous structure, the elements may or may not be of same type.
Example : records.

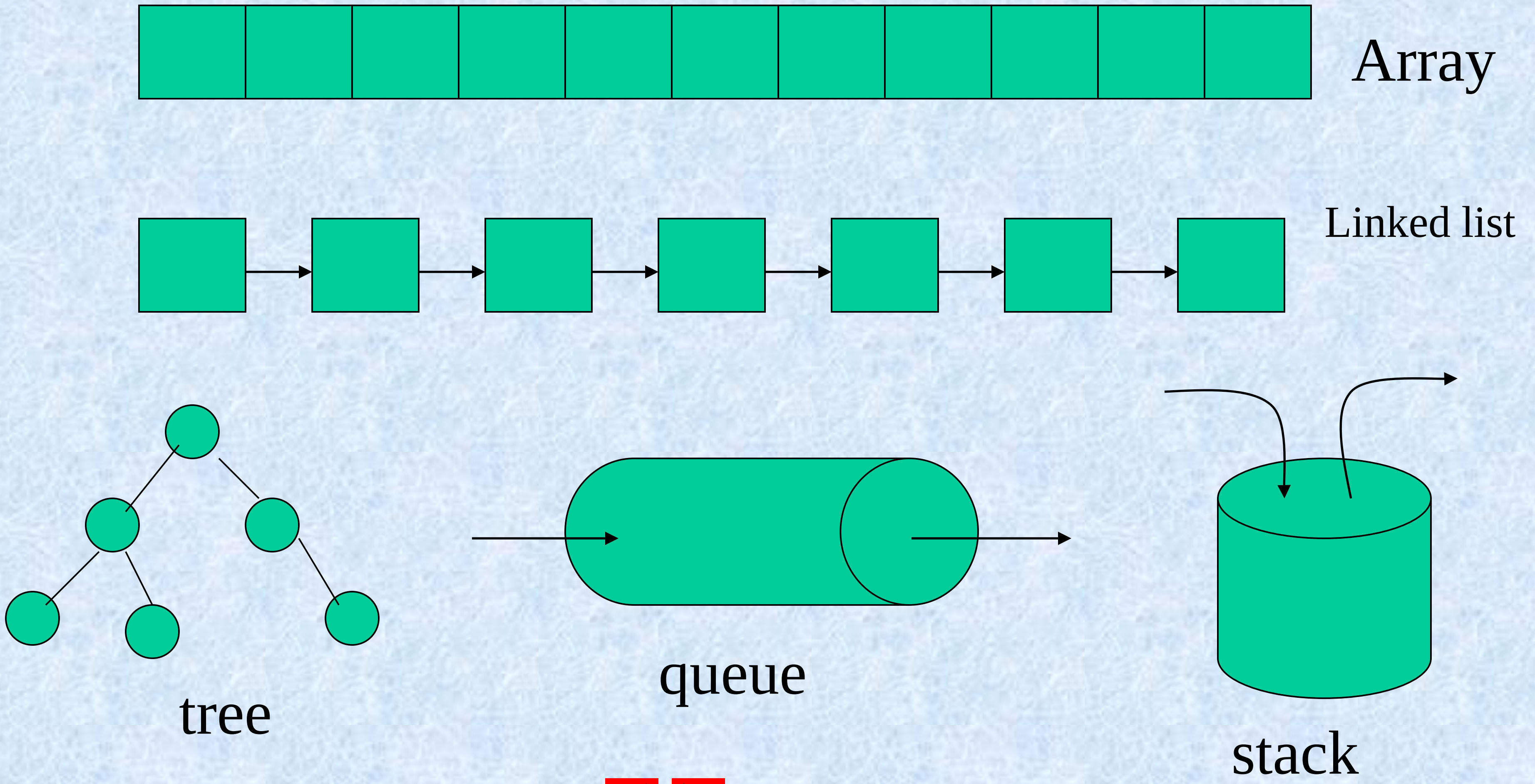
Static and Dynamic Data Structure

Static structures: whose size and structures, associated memory location are fixed at compile time.

Dynamic structures : which can expand or shrink as required during the program execution and there associated memory location change.

Classification of data structures




Array
Linked list
tree
queue
stack

Pseudo-code

A pseudo-code consists of natural language like statements that describes the steps of an algorithm.

Pseudo code: Addition of two numbers

```
Read num1 , num2
Addition =  num1 + num2
Display addition value
```

A data object is a ----------instance of data structures

Types of data objects are ------- and ------.

____________ Data structure include array, stack and linked list.

_________ Data structure include tree and graph.

Why need algorithm analysis?

- Just writing a syntax-error-free program is not enough. We need to know whether the algorithm is correct or not
- If the program is run on a large data set, the execution time becomes an issue.
- We want to know how the algorithm performs when the input size is large.
- The program may need lots of memory.
- We analyze the resources that the algorithm requires: memory, and computation time.

Analysis of the algorithm requires two main consideration i.e. performance Analysis:

Time complexity

Space complexity

Space complexity:

The space complexity of an algorithm is the amount of memory space required by algorithm during the course of its execution.

Instruction space

Data space

To calculate the space complexity, we must know the memory required to store different data type values

2 bytes to store Integer value,

4 bytes to store Floating Point value,

1 byte to store Character value,

8 bytes to store double value

Ex: SUM(A,B)

Step 1 : start

Step 2: C= A + B + 100

Step 3: stop

S(p)= 1+3 =4

Fixed amount of space for all input values then that space complexity is said to be Constant Space complexity.

```
Int sum(int S[ ], int n)
 { int sum = 0;
    int i;
    for(i = 0; i < n; i++)
       sum = sum + A[i];
    return sum;
 }
```

2n+8 bytes

Space complexity S(p) : S(p) of algorithm p is denoted as

S(p) = C+S(I) where C is fixed part and S(I) is variable part of I

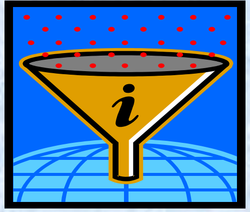
Time complexity :

The time complexity of an algorithm is a way to represent amount of time needed by the program to execute until its completion

1.Program is running on **Single/Multi** processor machine

2. **32 bit** machine or **64 bit** machine

3. Read and **Write** access speed of the machine.



Frequency Count:

- Total time taken by a statement in execution depend upon :
  - Amount of time single execution will take
  - Number of time given statement will be executed
- The product of these number will be the total time taken by the statement
- The number of time a statement will be executed is called as it's "Frequency Count"
- The frequency count may varies from data set to data set

```
x=x+1;
```
(a)

```
for(i=0;i<=n;i++)
    x=x+1;
```
(b)

```
For(j=0;j<=n;j++)
for(i=0;i<=n;i++)
    x=x+1;
```
(c)

In fig. a, if we consider the statement is not contained within any loop then, how many times the statement will executed

Frequency count = 1

In fig. b, how many times the statement will executed

Frequency count = n

In fig. c, how many times the statement will executed

Frequency count =n^2 (n>=1)

- Now $1, n, n^2$ are said to be different & increasing order of magnitude just like 1, 10, 100 if we let n=10
- During the analysis of algorithm we shall be concerned with determine the order of magnitude of an algorithm
- This means that we will determine only those statements which may have greatest frequency count
- The following formula is useful in counting the steps executed by an algorithm
  - $1+2+\ldots\ldots\ldots\ldots\ldots+n = \frac{n(n+1)}{2}$

Analyzing Running Time

T(n), or the running time of a particular algorithm on input of size n, is taken to be the number of times the instructions in the algorithm are executed. Pseudo code algorithm illustrates the calculation of the mean (average) of a set of n numbers:

The computing time for this algorithm in terms on input size n is: $T(n) = 4n + 5$.

```
1. n = read input from user
2. sum = 0
3. i = 0
4. while i < n
5. number = read input from user
6. sum = sum + number
7. i = i + 1
8. mean = sum / n
```

Statement	Number of times executed
1	1
2	1
3	1
4	n+1
5	n
6	n
7	n
8	1

Fibonacci series

- $F_n=F_{n-1}+F_{n-2}$ $n>=2$

```
void fibonacci(int n)
{
int i,fn1,fn2,fn;
if(n<0)
{
printf(“error”);
return;
}
if(n==0)
{
printf(“%d”,0);
return;
}
if(n==1)
{
printf(“%d”,1);
return;
}
fn2=0;
 fn1=1;
for(i=0;i<=n;i++)
{
fn=fn2+fn1;
fn2=fn1;
fn1=fn;
printf(“%d”, fn);
}
}
```

Asymptotic Notation

Asymptotic Notation is used to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an **algorithm's growth rate.**

Ex.:

Algorithm 1 : $5n^2 + 2n + 1$
Algorithm 2 : $10n^2 + 8n + 3$

Big - Oh (O)

Big - Omega (Ω)

Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh Notation (O): used to define the **upper bound** of an algorithm.

Maximum time required by an algorithm for all input values.

Describes the **worst case** of an algorithm time complexity.

Let function **f(n)** the time complexity of an algorithm

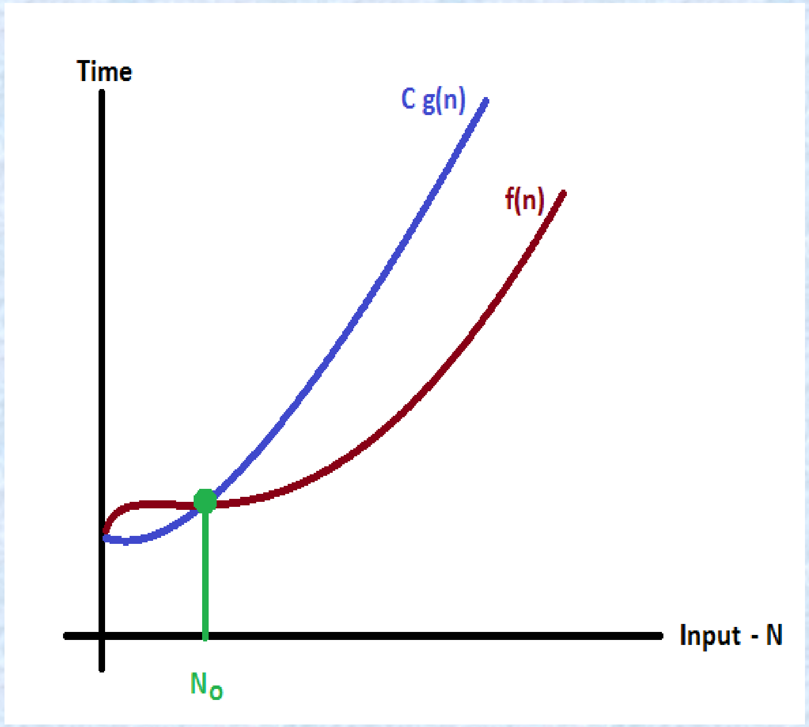
g(n) is the most significant term.

If $\mathbf{f(n) <= C\ g(n)}$ for all $\mathbf{n >= n_0}$, $\mathbf{C > 0}$ and $\mathbf{n_0 >= 1}$.

Then we can represent **f(n)** as **O(g(n))**.

$$\mathbf{f(n) = O(g(n))}$$

Big - Oh Notation (O)



f(n) = 3n + 2
g(n) = n
f(n) <= C x g(n) for all values of **C > 0** and **n_0>= 1**

Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as
3n + 2 = O(n)

Big - Omega Notation (Ω)

Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

It always indicates **minimum time** required by an algorithm for all input values.

It describes the **best case** of an algorithm time complexity.

Let function **f(n)** be the time complexity of an algorithm

g(n) is the most significant term.

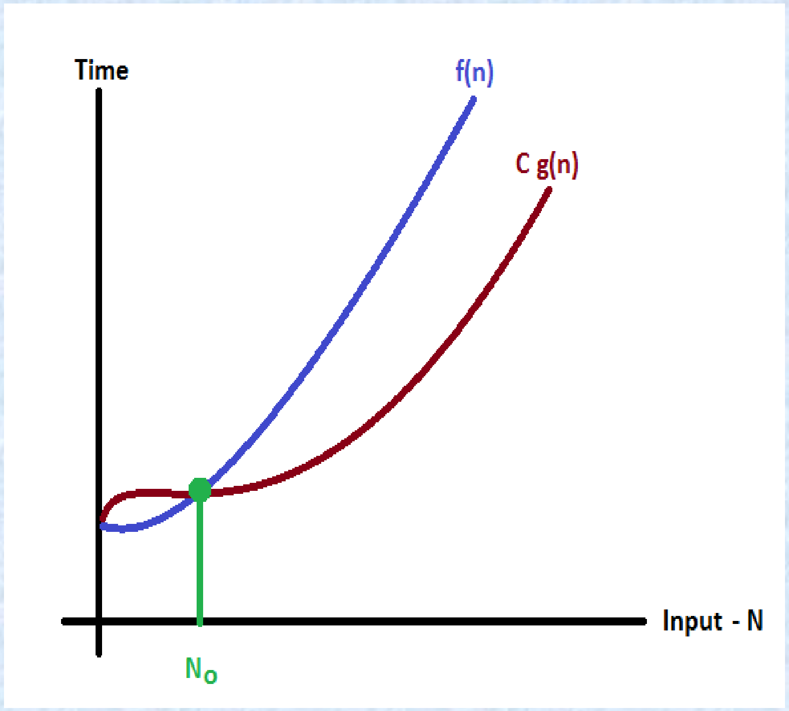
If $f(n) >= C \times g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$.

Then we can represent **f(n)** as **Ω(g(n))**.

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

◀ ▶

Big - Omega Notation (Ω)



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)

⇒3n + 2 <= C n

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

3n + 2 = Ω(n)

Theta Notation (Θ)

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.

Theta notation describes the average case of an algorithm time complexity.

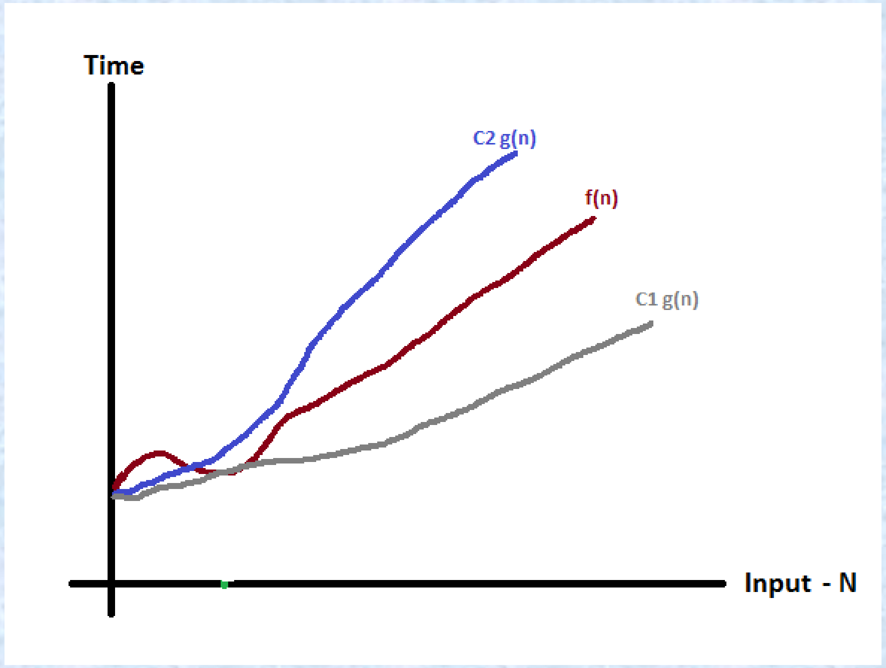
Let function **f(n)** the time complexity of an algorithm

g(n) is the most significant term.

If $\mathbf{C_1\ g(n) <= f(n) >= C_2\ g(n)}$ for all $\mathbf{n >= n_0}$, $\mathbf{C_1, C_2 > 0}$ and $\mathbf{n_0 >= 1}$.

we can represent **f(n) = Θ(g(n))**

Theta Notation (Θ)



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, C_2 > 0** and **n_0>= 1**

C_1 g(n) <= f(n) >= ⇒C_2 g(n)

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 >= 1**. Time complexity is represented as **3n + 2 = Θ(n)**

- Input instance for which algorithm take minimum possible time is called _______________
- ________ serves as upper bound of the performance measured.