1. Algorithm and Data Structure

1.1 Introduction

An algorithm is a well-defined and finite set of steps/instructions to solve any given specific problem when a proper input is provided to the algorithm.

The algorithm has following characteristics:

1. **Input**: An algorithm takes zero or more quantity as input.

2. **Output**: An algorithm gives at least one quantity as output as a result of its execution.

3. **Definiteness**: The set of instructions in an algorithm must be clear and unambiguous.

4. **Finiteness**: The set of instructions in an algorithm must be finite and eventually terminates the algorithm.

5. **Effectiveness**: Along with the definiteness, an algorithm must be simple enough to be solved by a paper and pencil without implementing it on any computer or computational machine.

1.2 Data, Information, Knowledge

Data refers to raw, unorganized facts and figures, such as numbers, words, or observations.
Information is data that has been processed, organized, and given meaning, providing insight or value.
Knowledge is the application of information and experience to make informed decisions, solve problems, or create new ideas.

To illustrate the progression:

Data: "The temperature is 25°C."
Information: "The temperature is higher than usual today."
Knowledge: "We should take precautions to stay cool and hydrated due to the heat."

Data structure

Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required. The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

- **Arrays**: A collection of elements, each identified by an index or a key. Arrays provide fast access to elements but may require resizing if the number of elements changes.
- **Linked Lists**: A sequence of nodes, where each node holds data and a reference to the next node in the sequence. Linked lists are dynamic and allow for efficient insertions and deletions.

- **Stacks**: A Last-In-First-Out (LIFO) data structure where elements are added and removed from the top. Stacks are often used for managing function calls and undo operations.
- **Queues**: A First-In-First-Out (FIFO) data structure where elements are added at one end (rear) and removed from the other end (front). Queues are commonly used for managing tasks in a systematic order.

1.3 Abstract Data Types (ADT)

an ADT defines a set of values and a set of operations that can be performed on those values, without specifying how the values are stored or how the operations are implemented.

Examples of abstract data types include:

Stack: a last-in, first-out (LIFO) data structure with push and pop operations
Queue: a first-in, first-out (FIFO) data structure with enqueue and dequeue operations
Set: a collection of unique values with add, remove, and contains operations
Map (or Dictionary): a collection of key-value pairs with put, get, and remove operations

ADTs are useful because they allow programmers to write code that uses a data structure without worrying about the implementation details. This makes the code more modular, reusable, and easier to maintain.

1.4 Data Structure Classification

A Data Structure delivers a structured set of variables related to each other in various ways. It forms the basis of a programming tool that signifies the relationship between the data elements and allows programmers to process the data efficiently.
We can classify Data Structures into two categories:

1. Primitive Data Structure
2. Non-Primitive Data Structure

The figure shows the different classifications of Data Structures.

1.5 Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.
3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).

4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
   Linear Data Structures
   Non-Linear Data Structures

1.6 Linear and Non-linear

Linear Data Structures
A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered.
   The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code. **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

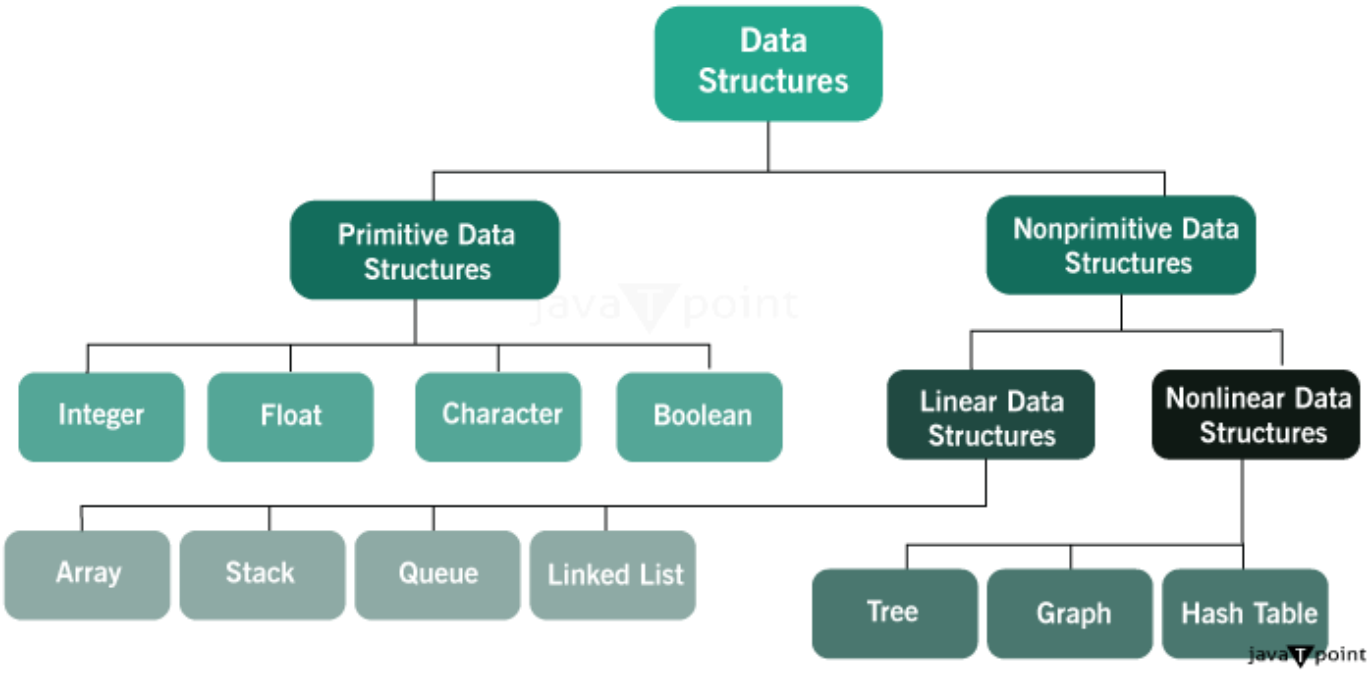


Figure-1.1: Data Structure Classification

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

1.7 Persistent and Ephemeral data structures

- **Persistent Data Structure:**

A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. They can be considered as 'immutable' as updates are not in-place. A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. E.g. Linked List

- **Ephemeral Data Structure:**

Ephemeral Data Structure is a Data Structure which is neither persistent nor partially persistent. It does not store previous versions of the Data Structure. Only the current state of Data Structure is maintained. E.g.

1.8 Program

Program is systematic way of writing step by step instructions to be given to computer in the language understood by computer. The program controls the activities of the computer to perform the intended task as per the instructions written in program.

There are various ways of writing programs and various computer languages are used to achieve this task. Prior to writing any program directly using any programming language, it is customary to write or depict these instructions to be given to computer in human understandable form without dependency on any language.

Some of the techniques are Flowchart, Algorithm and Pseudo code which are discussed in this section.

1.8.1 Flowchart

Flowchart is the simplest technique to represent the step by step instructions given to the computer into pictorial form. There are basic symbols to draw flowcharts and these symbols have some specific meaning and purpose. These symbols are depicted in figure-1.2 below.

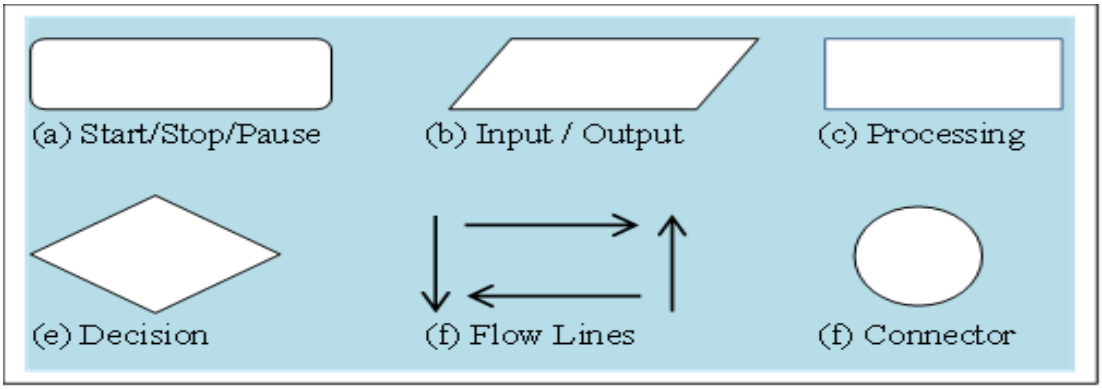


Figure-1.2: Basic Symbols of Flowchart

Let us take one simple example to understand the use of symbols for drawing flowchart. Draw a flowchart to read two numbers from user, perform addition operation on these numbers, and print the result onto the screen.

The flowchart for above given problem statement is shown in figure-1.3 below.

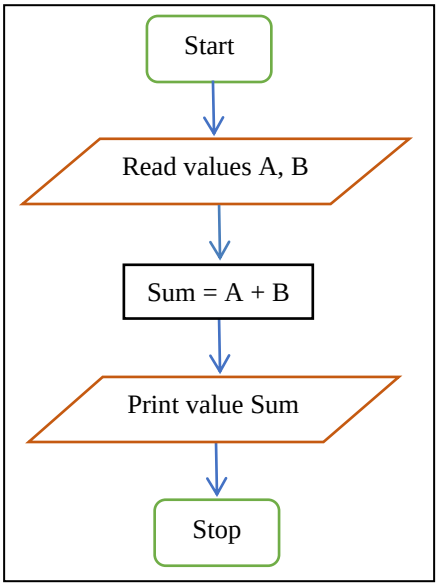


Figure-1.3: Flowchart for sum of two numbers

More flowchart structures will be discussed in subsequent sections.

1.8.2 Algorithm

An algorithm is a step wise solution of given problem, written in English or any known language of communication. It is plan of writing program logic to solve any given specific problem. Logic is thinking process by which one can write correct steps or instruction to solve the problem under consideration.

The algorithm should be precise and unambiguous and should get executed in finite amount of time to give the desired output.

The generic template of an algorithm is as given in figure-1.4.

Algorithm <Name of Algorithm>
Input*:* The input value(s) to be given to the algorithm
Output: The result after processing input value(s)
Steps:
The processing steps or instructions written in natural language

Figure-1.4: Algorithm Template

Let us write an algorithm to read two numbers from user, perform addition operation on these numbers, and print the result onto the screen.

The precise steps to solve above given problem can be written as follows:

Algorithm AddTwoNumbers
Input: Two numeric values
Output: Sum of the two numeric values
Steps:

1. Start.
2. Read two values as variables A and B from user.
3. Add two numbers stored in variable A & B and store result in variable Sum
4. Print value of variable Sum onto the screen.
5. Stop.

The above algorithm is the simplest one and it is very straight forward. But in real programming world an algorithm can take complex to complicated form. More algorithms will be discussed in subsequent sections.

1.8.3 Pseudo Code

The pseudo code is a tool or technique of putting the logic of given problem or program in form of imitate code. The code is again a program or systematic steps wise instructions given to the computer in some programming language. This code is written in some natural language like English. This code sometimes may resemble some known programming languages like C, C++, FORTRAN, etc. though in practice it is independent of any programming language.

Let us write pseudo code to read two numbers from user, perform addition operation on these numbers, and print the result onto the screen.

```
BEGIN
        Read(A,B);
        Sum = A + B;
        Print(Sum);
END
```

1.9 Programming

Programming is a process of writing the sequence of computer instructions or program into some know languages which are understood by computer hardware. The program is a planning and so programming is also planning or writing code in some computer language like C,C++, Java, etc.

In practical sense one may even write a program or write program logic in any form even without using any programming language. The pseudo can also be treated as programming. But in real computer world programming is treated as writing actual program logic in some computer programming language.

1.10 Programming Logic Structures

Programming is an art and not a science, i.e. one need to practice of writing program to be a skilled programmer. To do some meaningful programming in computer world one need to understand the different programming logic structures followed in standard practices of computer programs. These logic structures are:

1. Sequential
2. Conditional
3. Iteration or Loop

1.10.1 Sequential Structure

In sequential logic structure a programmer can write a program in very simple, straight and sequential steps without and twist in logic of program.

The example of sequential logic is pseudo code we have seen in section-1.3.3 in which we wrote simple and straight five step logic to add two numeric values given by user and print the resultant sum onto the screen.

1.10.2 Conditional Structure

Conditional logic structure of program is decision making task to be performed by computer to perform one among the many steps written in program. This logic structure is also known as selection logic as selection of a statement or statements is done based on certain constraints given in problem statement.

Let us draw a flowchart, an algorithm and pseudo code for a simple problem. Read two numbers from user and print the largest among the two numbers.

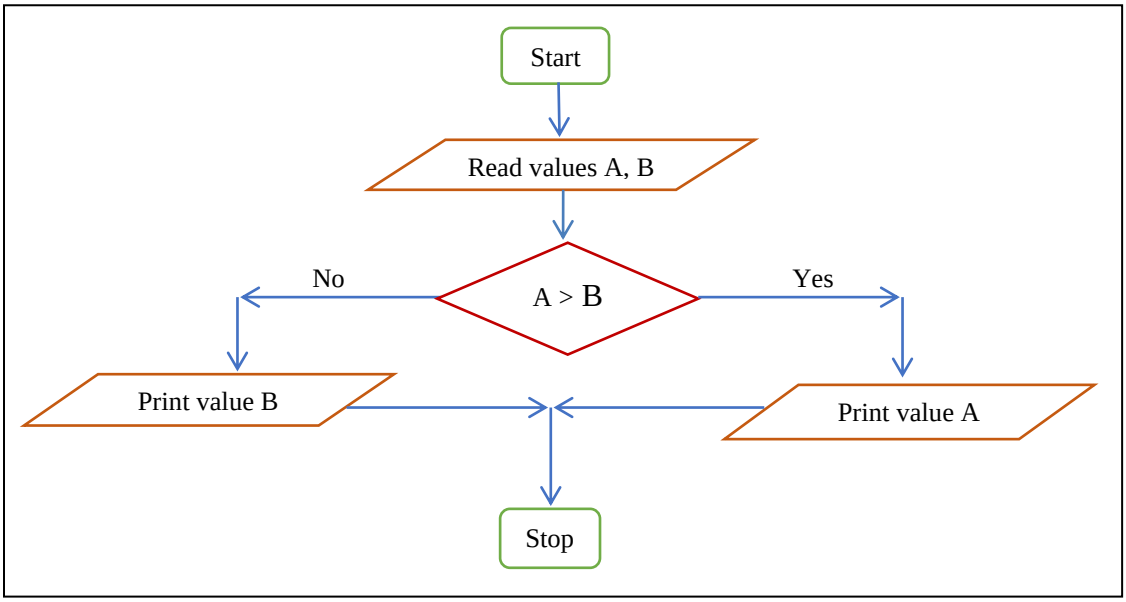


Figure-1.5: Flowchart for finding largest of two numbers

Algorithm LargerTwoNumbers
Input: Two numeric values
Output: Print largest among two numeric values
Steps:

1. Start.
2. Read two values as variables A and B from user.
3. If value of variable A is greater than value of B then goto step-4; otherwise go to step-5
4. Print value of variable A onto the screen. Go to step-6
5. Print value of variable B onto the screen.
6. Stop.

Figure-1.6: Algorithm for finding largest of two numbers

In pseudo code of figure-1.7 some mathematical symbols are use. In pseudo code all mathematical symbols like +, -, x, /, >, < , >=, <=, = can be used freely. Along with this some language specific key words may be used to make program more elegant and understanding. An algorithm or a pseudo code is simple but powerful technique of putting logic of program in form of instructions to computer.

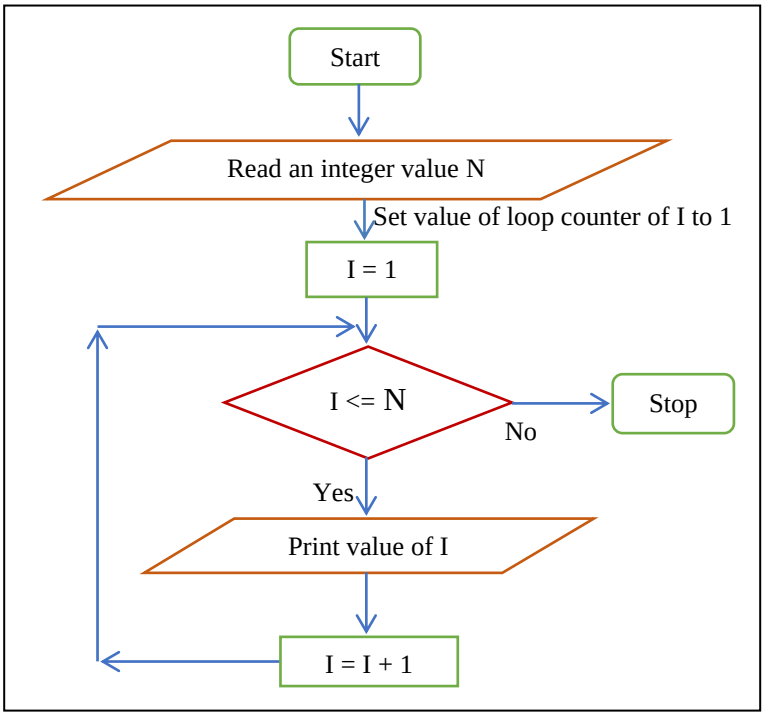
```
Begin
    Read (A, B)
    If (A > B) then
        Print value of variable A onto the screen.
    Else
        Print value of variable B onto the screen.
    End if
End
```

Figure-1.7: Pseudo code for finding largest of two numbers

1.10.3 Loop Structure

When we want computer to repeat an instruction or set of instructions till certain condition is met then loop structure also called as iteration logic is used. An instruction or set of instructions is executed repeatedly till some terminating condition is reached by the iterative program logic. The iterative logic can be represented using flowchart, an algorithm or a pseudo code.

Let us draw a flowchart, an algorithm and pseudo code for a simple problem of iteration. Print numbers from 1 to N onto the screen. The 'N' value is an integer value given as input by user.



1.10.4 Programming languages

Depending on the level of computer knowledge, a computer expert may write computer program in various computer languages and these languages are broadly categorized as:

High Level Language (HLL)
Assembly Level Language (ALL)
Low Level Language (LLL)

1.11 Analysis of Algorithms

Let us now understand how to do analysis of *non-recursive or an iterative* algorithm.

1.11.1 Non-recursive algorithm analysis

1. 11.1.1 Tabular Method

Example-1.
Analyze the algorithm which reads two numbers from user as input, performs addition of two numbers and displays the result.

Algorithm:

1. Read (a,b);
2. Sum=a+b;
3. Print (Sum)

Table:

Statement No.	Frequency Count (FC)
1.	1
2.	1
3.	1
Total	**3**

Frequency Count (FC): It is the number of times each statement is getting executed in code.

Analysis:

Here each statement is getting executed only once so the total FC is 3. So this algorithm takes constant time which is 3 units.

Here unit is used as this value is system dependent. On some computer it may take 3 seconds to run the above code, on other it may be milliseconds or nanoseconds or picoseconds or even less than that.

Example-2.
Analyze the algorithm which reads two numbers from user as input, finds largest among of two numbers and displays the largest.

Algorithm:

1. Read (a,b);

2. If (a > b)
3. Print("The largest is a");
   Else
4. Print("The largest is b");

Table:

Statement No.	(FC)	
	If (a>b) true	**If (a>b) false[else part]**
1.	1	1
2.	1	1
3.	1	0
4.	0	1
Total	**3**	**3**

Analysis:

Here statement no. 1 is getting executed only once irrespective of the value of 'a' or 'b'. Statement no. 2 is executed to test if 'a' is greater than 'b' but this statement is executed even if 'a' is less than 'b'. Statement no. 3 is executed only if a>b is true . Statement no. 4 is executed only if a>b is false. So the total FC of an algorithm is 3 time units in both the cases of 'if' and 'else'.

Example-3.

Analyze the algorithm which reads number n from user as input and prints all the numbers from 1 to n.

Algorithm:

1. Read (N);
2. I=1;
3. While (I <= N)
   {
4. Print(I);
5. I=I+1;
   }

Table:

Statement No.	FC
1.	1
2.	1
3.	N+1
4.	N
5.	N
Total	**3N+3**

Analysis:

Here statement no. 1 & 2 are getting executed only once. Statement no. 3 is executed N+1 times, N times for true condition and 1 time more for false condition to come out of loop. The statements inside the loop are executed only till test condition of while is true. As loop test condition is true till N, statement 4 & 5 are executed N times. So the total FC of an algorithm is 3N+3 time units.

Example-4.
Analyze the code snippet given below.

Algorithm:

```
1. For(I=1; I<=N; I++)
2.     For(J=1; J<=M; J++)
3.         X=X+1;
```

Table:

Statement No.	FC
1.	N+1
2.	N .(M+1)
3.	N . M
Total	**2N.M+2N+1**

Analysis:

Here we assume that for is composite single statement though it contains three statements in it. Statement no. 1 is executed N+1 times, N times for true condition and 1 time more for false condition to come out of loop. Statement no. 2 will get executed M+1 times as a single statement but as it is part of outer while loop its execution is repeated each time when test condition of outer loop is true. So the total execution of a statement no. 2 is N x 9M+1) times. The statement 3 is executed till inner loop test condition is true, i.e. M times but inner loop is again a part of outer loop which is executed N times so the total time required is N x M times. So the total FC of an algorithm is 2N.M+2N+1 time units.

Example-5.
Analyze the code snippet given below.

Algorithm:

```
1. For(I=1; I<=N; I++)
2.     For(J=1; J<=I; J++)
3.         X=X+1;
```

Table:

In the table below the iterations for I and J loop counter variables of outer and inner loops respectively is given. For each I value of inner loop, outer loop counter value J is restricted. When I=1, the value of J=1,2. When I=2, the value of j=1,2,3 and so on. When I=N, the value of J=1,2,3,…….N,N+1.

The total of each statement 1 , 2 & 3 is given after each iteration of the loop execution.

Loop Counter I	Loop Counter J	(FC)		
		Statement 1.	**Statement 2.**	**Statement 3.**
1	1	1	1	1
	2	0	1	0
Total		**1**	**2**	**1**
2	1	1	1	1
	2	0	1	1
	3	0	1	0
Total		**1**	**3**	**2**
3	1	1	1	1
	2	0	1	1
	3	0	1	1
	4	0	1	0
	Total	**1**	**4**	**3**
N	1	1	1	1
	2	0	1	1
	3	0	1	1
	.	0	1	1
	.	0	1	1
	.	0	1	1
	.	0	1	1
	N	0	1	1
	N+1	0	1	0
	Total	**1**	**N+1**	**N**

Analysis:

Here to analyze each statement we need to add the total count of the execution of each iteration.

Statement No.1:

Total count = 1 +1 +1+1 +1……..N times

= (N+1) times [why?]

So the total time taken by statement no. 1 is (N+1) time units. N times for true value of test condition and 1 time for false condition to come out of loop.

Statement No.2:

Total count = 2 +3+4 +5……..+N+N+1

= (1+2 +3+4 +5……..+N)+N (after rearranging the terms)

$$=\sum_{i=1}^{N} i + N = (1+2+3+\ldots\ldots+N) + N = [N(N+1)/2] + N = (N^2+N)/2 + N$$

$$=(N^2+3N)/2$$

So the total time taken by statement no. 2 is $(N^2+3N)/2$ time units.

Statement No.3:

Total count = 1+2 +3+4 +5……..+N

$$=\sum_{i=1}^{N} i = 1+2+3+\ldots\ldots+N = N(N+1)/2 = (N^2+N)/2$$

$$=(N^2+N)/2$$

So the total time taken by statement no. 3 is $(N^2+N)/2$ time units.

The total time taken by an algorithm or code snippet is sum of the time taken by all three statements of the code.

Total FC = $(N+1) + (N^2+3N)/2 + (N^2+N)/2$

$= (2N+2+ N^2+3N+ N^2+N)/2$

$= (2N^2 +6N+2)/2$

$=N^2 +3N+1$

Example-6.

Analyze the code snippet given below.

Algorithm:

```
1.  For(I=1; I<=N; I++)
2.    For(J=0; J<=N-I; J++)
3.       If(A[J] > A[J+1])
         { //Code for exchange of A[J] & A[J+1] location data.
4.          Temp=A[J];
5.          A[J]=A[J+1];
6.          A[J+1]=temp;
          }
```

Table:

In the table below the iterations for I and J loop counter variables of outer and inner loops respectively is given. For each I value of inner loop, outer loop counter value J is restricted to N-I. When I=1, the value of J=0 to N-2. When I=2, the value of j=0 to N-3 and so on. When I=N-1, the value of J=0 to N-(N-1), i.e. J=0 to 1.

Above code snippet is standard bubble sort. For any sorting algorithm we need to analyze the no. of comparisons or exchanges done for that sort.

Loop Counter I	**Loop Counter J**	**J < N-I value is true**	**No. of Comparisons for statements 4,5 & 6**
1	0	0<N-1 is true	1
	1	1<N-1 is true	1
	2	2<N-1 is true	1
	.	true	1
	.	true	1
	.	.	.
	N-1	N-1<=N-1 is true	1
	N	N<=N-1 is False	0
Total			**N-1**
2	0	0<N-2 is true	1
	1	1<N-2 is true	1
	2	2<N-2 is true	1
	.	true	1
	.	true	1
	.	.	.
	N-2	N-2<=N-2 is true	1
	N-1	N-1<=N-2 is False	0
Total			**N-2**
.			**.**
.			**.**
.			**.**
.			**.**
N-1	0	0<N-(N-1) is true	1
	1	1<=1 is true	1
	2	2<=1 is true	0
Total			**2**
N	0	0<=N-N is true	1
	1	1<=N-N is false	
Total			**1**

Analysis:

Here we have to now add the total no. of comparisons done till the outer loop will get terminated.

So Total FC = N+(N-1)+(N-1)+……+2+1

=1+2+3+ . . ……(N-2)+(N-1)+N (After rearranging the terms)

$= \sum_{i=1}^{N} i = 1+2+3+\ldots\ldots+N = N(N+1)/2 = (N^2+N)/2$

2. Standard Series Method (Shortcut Method)

We have seen that tabular method is easy to perform analysis of any code or an algorithm but it's quite lengthy too. We now introduce a short cut method to do analysis of code. To use this method we need to know the formulas of standard arithmetic, geometric or harmonic series.

Let us first revise these formulas before we start the actual analysis of any given algorithm.

Arithmetic series

The summation

1.

$$\sum_{i=1}^{n} i = 1+2+3+\ldots\ldots+n = n(n+1)/2$$

2.

$$\sum_{i=0}^{n} i^2 = 0^2+1^2+2^2+3^2+\ldots\ldots+n^2 = n(n+1)(2n+1)/6$$

3.

$$\sum_{i=0}^{n} i^3 = 0^3+1^3+2^3+3^3+\ldots\ldots+ n^3 = n^2(n+1)^2/4$$

Geometric series

1.

$$\sum_{i=0}^{n} x^i = 1+x+x^2+\ldots\ldots\ldots+x^n = (x^{n+1} - 1)/(x-1)$$

where $x \neq 1$, is real number.

2.

When the summation is infinite and $|x| < 1$ we have,

$$\sum_{i=0}^{\infty} x^i = 1+x+x^2+\ldots\ldots\ldots+x^{\infty} = 1/(1-x)$$

Harmonic series

For positive integers *n*, the n^{th} ***harmonic number*** is

1.

$$H_n = 1 + 1/2 + 1/3+ ¼ +\ldots..+1/n$$

n

$$\sum_{k=1} 1/k = \log(n) + O(1)$$

2.

$$\sum_{k=0}^{n} kx^k = x/(1-x)^2$$

for $|x| < 1$

Telescoping series

For any sequence $a_0, a_1, ..., a_n$,

1.

$$\sum_{k=0}^{n} (a_k - a_{k-1}) = a_n - a_0$$

2.

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n$$

3.

$$\sum_{k=1}^{n-1} 1/k(k+1) = \sum_{k=1}^{n-1} [1/k - 1/(k+1)] = 1 - 1/n$$

here,

$1/k(k+1) = 1/k - 1/(k+1)$

Let us now solve some problems using shortcut method.

Example-1.

Analyze the code snippet given below.

Algorithm:

```
1. For(i=1;i<=N; i++)
2.     For(j=1; j<=N; j++)
3.         X=X+1;
```

Analysis:

Statement No.1

FC= N+1 (do you agree? why?)

Statement No.2

$$FC= \sum_{i=1}^{N} \sum_{j=1}^{N+1} (1)$$

$$FC= \sum_{i=1}^{N} (N+1)= (N+1)\sum_{i=1}^{N} (1)$$

FC= $(N+1)(N)=N^2+N$ time units.

In above analysis we assume for loop as single composite statement and denote it as (1). Thus the inner j loop iterates for N+1 times till outer i loop iterates for N times.

Statement No.3

$$FC= \sum_{i=1}^{N} \sum_{j=1}^{N} (1)$$

$$FC= \sum_{i=1}^{N} N= N \sum_{i=1}^{N} (1)$$

FC= $(N)(N)=N^2$ time units.

In above analysis, statement 3 is constant statement and will get executed till inner j loop iterates for N times and j loop will be executed till outer i loop iterates for N times.

Example-2.
Analyze the code snippet given below.

Algorithm:

```
1. For(i=1; i<=N; i++)
2.     For(j=1; j<=i; j++)
3.         X=X+1;
```

Analysis:

Statement No.1

FC= N+1 (why?)

Statement No.2

$$N \quad i+1$$

$$FC= \sum_{i=1}^{N} \sum_{j=1}^{i} (1)$$

$$FC= \sum_{i=1}^{N} (i+1) = \sum_{i=1}^{N} i + \sum_{i=1}^{N} 1$$

FC=(1+2+3+……..+N) + N

FC= [N(N+1)/2] +N= (N^2+N+2N)/2= (N^2+3N)/2time units.

Here the inner j loop iterates for i+1 times till outer i loop iterates for N times.

Statement No.3

$$FC= \sum_{i=1}^{N} \sum_{j=1}^{i} (1)$$

$$FC= \sum_{i=1}^{N} i = (1+2+3+……..+N)$$

FC= N (N+1)/2 = (N^2 + N)/2 time units.

In above analysis, statement 3 is constant statement and will get executed till inner j loop iterates for i times and j loop will be executed till outer i loop iterates for N times.

1. 11.1.2 Recursive algorithm analysis

Let us now understand how to do analysis of *recursive or non-iterative* algorithm.

A **recurrence** is an equation that describes a function in terms of itself by using smaller inputs.
The expression

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

Describes the running time for a function contains recursion.

Some more examples of recursions.

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ n + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

There are three different ways of doing the analysis of recursive algorithms and these methods are:

1. Substitution
2. Iteration
3. Master Theorem

1. The substitution method

- "making a good guess method"
- Guess the form of the answer, then
- use induction to find the constants and show that solution works

Consider an example below.

$$T(n) = 2T(\sqrt{n}) + \lg n$$

- Simplify it by letting m = lg n ➔ $n = 2^m$
- Rename S(m) = $T(2^m)$
- S(m) = 2S(m/2) + m = O(m lg m)
- Changing back from S(m) to T(n), we obtain
- T(n) = $T(2^m)$
  = S(m)
  = O(m lg m)
  = O(lg n lg lg n) [by replacing m=lg n]

2. Iteration method:

- Expand the recurrence *k* times
- Work some algebra to express as a summation
- Evaluate the summation

Consider an example below.

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

- $T(n) = c + T(n\text{-}1) = c + c + T(n\text{-}2)$
  $= 2c + T(n\text{-}2) = 2c + c + T(n\text{-}3)$
  $= 3c + T(n\text{-}3) \ldots kc + T(n\text{-}k)$
  $= ck + T(n\text{-}k)$
- So far for $n \geq k$ we have
  - $\boldsymbol{T(n) = ck + T(n\text{-}k)}$
- To stop the recursion, we should have
  - $n - k = 0$ ➔ $k = n$
  - $\boldsymbol{T(n) = cn + T(0) = cn}$
- Thus in general $\boldsymbol{T(n) = O(n)}$

Solved Examples:

1. Write a recursive binary search and determine its time complexity.

Solution:

```
Int recBinarySearch(int A[],int L, int B, int key)
{
    int mid;
    if(L > U)
      Return -1;// if Lower bound crosses Upper bound then return -1, i.e. number not found
    Mid=(L+U)/2;
    If(key == A[Mid])
       return [Mid+1]; // if key matches value at location Mid then return that location
    If(key > A[Mid])// if key to be found is larger than current mid location
         return recBinarySearch(A,Mid+1,U) // call recursively algorithm to search key in upper half
   else  //otherwise if key to be found is smaller than current mid location
            return recBinarySearch(A,L,Mid-1); // call recursively algorithm to search key in lower half
}
```

Analysis of Algorithm:

Now consider the sorted array A of size n. The recurrence relation can be written as:

(1) As the algorithm takes only one recursive call if n=1, it will return immediately with constant time of T(1).

(2) But if n>1 than the recursion function will be called on for each n/2 elements lying either at left half of Mid-1 location if key < A[Mid] or at the right half after Mid+1 location if key is larger than A[Mid] location.

This can be shown using recurrence function as:

$$T(n) = \begin{cases} 1 & \text{for } n=1 \\ T(n/2) + C & \text{for } n > 1 \end{cases}$$

Now solving it with repeated substitution method we get,

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

$T(n) = T(n/4) + 2C$

$T(n) = T(n/8) + 3C$

.

.

.

$T(n) = T(n/2^i) + iC$ ----------(1)

Now Let $n/2^i = 1$

$n=2^i$

$2^i = n$ --------(2)

$i=\log_2 n$ --------(3)

Now putting equation (2) & (3) in equation (1) we get,

$T(n)= T(n/n) + \log_2 n\ C$
$T(n)= T(1) + C \log_2 n$

But from above recurrence relation T(1) = 1. So putting it in above equation we get,

$T(n) = 1 + C \log_2 n$

Ignoring the constants we get,

$T(n) = O(\log_2 n)$

2. Write a recurrence relation for following code and find its complexity.

```
Function Sq(n)
if n = 0 then
return 0
else
return 2n+Sq(n-1) -1
End Function
```

Solution:

This can be shown using recurrence function as:

$$T(n) = \begin{cases} 0 & \text{for } n=0 \\ 2n+T(n-1)-1 + C & \text{for } n > 1 \end{cases}$$

Now solving it with repeated substitution method we get,

$T(n) = 2n + T(n-1) -1 +C$
$= 2n+[2(n-1)+T(n-2) -1+C]-1 +C$
$= 2n+[(2n-2)+T(n-2) -1+C]-1 +C$
$= 2n+(2n-2)+T(n-2) -1+C-1 +C$
$= 2n+2n-2+T(n-2) -2+2C$
$= 2[n+ n-1] + T(n-2) -2 + 2C$
$= 2[n + n-1 + n-2] + T(n-3) – 3 + 3C$
$= 2[n + n-1 + n-2 + n-3] + T(n-4) -4 + 4C$
$= \ldots$
$= 2[n + n-1 + n-2 + n-3 + \ldots + (n-k+1)] + T(n-k) – k +kC$
$=2[(n-k+1) + \ldots\ldots+ n-3 + n-2 + n-1 + n] + T(n-k) – k +kC$ [After rearranging the terms]

$$= 2\sum_{i=n-k+1}^{n} i + T(n-k) – k +kC \quad , \text{for } n \geq k$$

To stop the recursion, we should have n - k = 0 ➔ k = n

$$T(n) = 2\sum_{i=1}^{n} i + T(0) – n +nC \quad , \text{for } n \geq k$$

$$= 2 \sum_{i=1}^{n} i + T(0) + nC - n$$

$$= 2 \sum_{i=1}^{n} i + T(0) + n(C-1)$$

$= 2 (n (n+1)/2) + 0 + n(C-1)$

$= n(n+1) + n(C-1)$

$= O(n^2)$ ---- by ignoring lower order term

1.12 Best Case, Worst Case, Average Case

1.12.1 Worst-case running time of an algorithm

It is the Maximum time required for program execution, saying that it is longest running time for **any** input of size n.

It is also an ***upper bound*** on the running time for any input 'n' which guarantees that the algorithm will never take time longer than the upper bound.

Example:

a. Sort a set of numbers in increasing order; and the data is in decreasing order.
b. The worst case can occur fairly often in searching a database for a particular piece of information.

1.12.2 Average-case running time

It is an Average time required for program execution. It may be difficult to define what "average" means but in practical sense we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.

Example:

a. Searching a key in an array sequentially and key is located near to the mid location of an array.

1.12.3 Best-case running time of an algorithm

It is the minimum time required for program execution, saying that it is the minimum running time for **any** input of size n.

It is also a ***lower bound*** on the running time for any input 'n' which guarantees that the algorithm will take time the minimum time greater than or equal to the lower bound.

Example:

a. Sort a set of numbers in increasing order; and the data is already in increasing order.
b. Searching given key in the set of data items which are uniformly distributed and key found at the beginning location of an array.

Examples:

1. Analyse the following algorithm Best, Average and Worst case

```
int linearSearch(int a[],int n, int key)
{
        int I;
```

```
        For (i=0;i<n;i++)
         if(key == a[i])
                return (i+1); //return the location of a number

        return (-1); //number not found, so return invalid index
}
```

1.13 Best, Average and Worst-case Analysis

Best case analysis:

If the key to be search is found at the first location (index '0') of an array then the search return with only one comparison so the best case time complexity of the search algorithm is **O(1)**.

Average case analysis:

If the 'n' elements of an array are ***uniformly distributed*** then the key may be found near to the middle location of an array with '*n/2*' comparisons. So, the average case time complexity will be **O(n/2)**.

Worst case analysis:

If the 'n' elements of an array are ***not uniformly distributed*** then the key may be found at the last location of an array with *'n' comparisons* or in case key is not found it will make '*n+1*' number of comparisons. So, the worst case time complexity will be **O(n)**.

2. Analyse the following algorithm Best, Average and Worst case

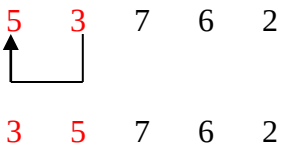
```
void sort (int a[], int n) {
        int i, j;
        for (i = 0; i < n; i++) {
                j = i-1;
                key = a[i];
                while (j >=0 && a[j] > key)
                {
                        a[j+1] = a[j];
                        j = j-1;
                }
                a[j+1] = key;
        }
}
```

Worse case Analysis: $O(n^2)$

When we apply insertion sort on a reverse-sorted array or almost reversed sorted array, it will insert each element at the beginning of the sorted subarray, making it the worst time complexity of insertion sort.

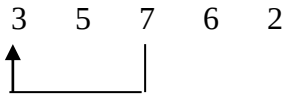
Given Set of data :: 5 3 7 6 2

First Pass::



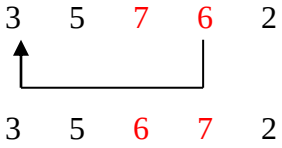
no. of comparisons: 1

Second Pass::



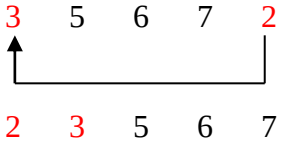
no. of comparisons: 2

Third Pass::



no. of comparisons: 3

Forth Pass::



no. of comparisons: 4

Fifth Pass::

2 3 5 6 7

no. of comparisons: 0

Total compares in general = 1+2+3+…+(n-1)

= (n-1)n/2 = $(n^2-n)/2$= $O(n^2)$

Average case: $O(n^2)$

When the array elements are in random order, the average running time is almost equal to $O(n^2)$. As the array elements are stored randomly, the number of comparisons will be nearly same as the worst case scenario.

Best case: O(n)

When we initiate insertion sort on an already sorted array, it will only compare each element to its predecessor, thereby requiring n steps to sort the already sorted array of n elements.

Algorithm	*B(n)*	*A(n)*	*W(n)*
HornerEval	n	n	n
Towers	2^n	2^n	2^n
LinearSearch	1	n	n
BinarySearch	1	logn	logn
Max, Min , MaxMin	n	n	n
InsertionSort	n	n^2	n^2
MergeSort	nlogn	nlogn	nlogn
HeapSort	nlogn	nlogn	nlogn
QuickSort	nlogn	nlogn	n^2
BubbleSort	n	n^2	n^2
SelectionSort	n^2	n^2	n^2
GnomeSort	n	n^2	n^2

Figure: Best case B(n), Average case A(n) and Worst case W(n) Complexities

1.14 Asymptotic growth, O, Ω, Θ, o and ω notations

Asymptotic Notation

- Describes the behavior of the time or space complexity for large instance characteristic
- Big Oh (O) notation provides an upper bound for the function *f*
- Omega (Ω) notation provides a lower-bound
- Theta (Θ) notation is used when an algorithm can be bounded both from above and below by the same function
- Little oh (o) defines a loose upper bound.
- Little omega(ω) defines a loose lower bound.

Big Oh (O) Notation

Definition:

f(n) = O(g(n)) (read as "*f(n)* is Big Oh of *g(n)*") iff positive constants *c* and n_0 exist such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.

That is, *O(g)* comprises all functions *f*, for which there exists a constant *c* and a number n_0, such that $f(n)$ is ***smaller or equal to*** $c \cdot g(n)$ for all n, $n \geq n_0$.

Omega (Ω) Notation

Definition:

f(n) = Ω(g(n)) (read as "*f(n)* is omega of *g(n)*") iff positive constants *c* and n_0 exist such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.

That is, W(*g*) comprises all functions *f*, for which there exists a constant *c* and a number n_0, such that *f*(*n*) ***is greater or equal*** to $c \cdot g(n)$ for all $n \geq n_0$.

Theta (Θ) Notation

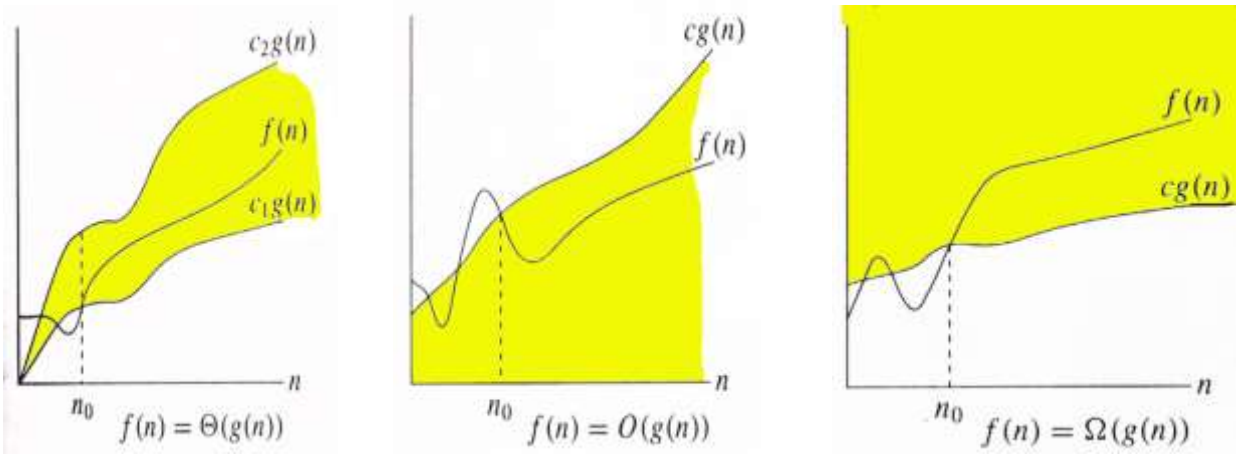
Used when the function *f* can be **bounded both from above and below** by the same function *g*.

Definition:

f(n) = Θ ***(g(n))*** (read as "*f(n)* is theta of *g(n)*") iff positive constants c_1, c_2 and n_0 exist such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$.

That is, *f* lies between c_1 times the function *g* and c_2 times the function *g*, except possibly when *n* is smaller than n_0.

Relations between Θ, *O*, Ω



Little Oh (o) Notation

Definition:

f(n) = o(g(n)) (read as "*f(n)* is Little Oh of *g(n)*") iff positive constants *c* and n_0 exist such that $f(n) < cg(n)$ for all $n, n \geq n_0$.

That is, *O(g)* comprises all functions *f*, for which there exists a constant *c* and a number n_0, such that *f*(*n*) is ***smaller than*** $c \cdot g(n)$ for all $n, n \geq n_0$.

The bound $2n^2 = O(n^2)$ is asymptotically **tight**, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is **not asymptotically tight**.

The main difference is that in f(n) = O(g(n)), *the bound* $0 \leq f(n) \leq cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, *the bound* $0 (n) < cg(n)$ holds for *all constants* $c > 0$. Intuitively, in the *o*-notation, the function *f*(*n*) becomes insignificant relative to *g*(*n*) as *n* approaches infinity; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 .$$

Little Omega (ω) Notation
Definition:
f(n) = Ω(g(n)) (read as "*f(n)* is Little omega of *g(n)*") iff positive constants *c* and n_0 exist such that *f(n)* > *cg(n)* for all *n*, $n \geq n_0$.

That is, W(*g*) comprises all functions *f*, for which there exists a constant *c* and a number n_0, such that *f*(*n*) ***is greater than*** $c \cdot g(n)$ for all $n \geq n_0$.
We use ω -notation to denote a lower bound that is **not asymptotically tight.**

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

Examples:

1. solve $f(n) = 12n^5 + 13n^3+20$ for g(n) with **Θ, O, Ω**
   a. **O notation:**
   $f(n) = 12n^5 + 13n^3+20$
   $12n^5 + 13n^3+20$= **O(g(n))**
   Where $g(n)=n^5$
   ∴ $12n^5 + 13n^3+20$= **$O(n^5)$**
   i.e. $12n^5 + 13n^3+20 \leq c1\,g(n)$
   i.e. $12n^5 + 13n^3+20 \leq c1\ n^5$

   Now here we need to find the constant c1.
   For all $n \geq 1$ and c1=45, we have $f(n) \leq c1\ g(n)$
   HINT:
   Here, see that coefficient of all the terms, i.e. 12+13+20=45. So, choosing any value of c1 greater or equal to 45 will prove it.

   b. **Ω notation**
   $f(n) = 12n^5 + 13n^3+20$
   $12n^5 + 13n^3+20$= **Ω (g(n))**
   Where $g(n)=n^5$
   ∴ $12n^5 + 13n^3+20$= **$O(n^5)$**
   i.e. $12n^5 + 13n^3+20 \geq c1\,g(n)$
   i.e. $12n^5 + 13n^3+20 \geq c1\ n^5$

Now here we need to find the constant c1.
For all $n \geq 1$ and c1=11, we have $f(n)\ c1 \geq g(n)$
HINT:
Here see the coefficient of largest degree term i.e. n^5 term's coefficient. By choosing any value of c1 smaller than 12 will prove it.

c. Θ notation

$f(n) = 12n^5 + 13n^3+20$
$12n^5 + 13n^3+20= \mathbf{\Theta(g(n))}$
Where $g(n)=n^5$
$\therefore 12n^5 + 13n^3+20= \mathbf{\Theta(n^5)}$
i.e. $c1g(n) \leq 12n^5 + 13n^3+20 \leq c2g(n)$
i.e. $c1\ n^5 \leq 12n^5 + 13n^3+20 \leq c2\ n^5$

Now here we need to find the constants c1 and c2.
For all $n \geq 1$ and c1=45 and c2=11, we have $c1g(n) \leq f(n) \leq c2g(n)$
HINT:
As discussed earlier in case of ***O, Ω***

2. Interpret the following equation
$2n^2 + \Theta(n) = \Theta(n^2)$
$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

Solution:

When we write $n = O(n^2)$, we have already defined the equal sign to mean set membership:
$n \in O(n^2)$.

a. $2n^2 + \Theta(n) = \Theta(n^2)$

The formula $2n2 + \Theta(n) = \Theta(n2)$ means that $2n2 + f1(n) = 2n2 + f2(n)$,
where f1(n) is some function in the set $\Theta(n)$. In this case, $f(n) = c1n + c2$, which indeed is in $\Theta(n)$. And f2(n) is some function in the set $\Theta(n2)$.

b. $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

The formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$,
where f(n) is some function in the set $\Theta(n)$. In this case, $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.

1.15 Two fundamental algorithm design paradigms

Divide and Conquer:

Divide and Conquer is a strategy for solving complex problems by breaking them down into smaller, more manageable sub-problems. The process involves:

1. Divide: Split the problem into smaller sub-problems that are more tractable.
2. Conquer: Solve each sub-problem recursively or iteratively.
3. Combine: Combine the solutions to the sub-problems to form the final solution.

Examples of Divide and Conquer algorithms

- Binary Search
- Merge Sort
- Quick Sort
- Fast Fourier Transform (FFT)

Greedy Strategy:

The Greedy Strategy is a paradigm that involves making locally optimal choices with the hope of finding a global optimum solution. In other words, at each step, the algorithm makes the best choice available, without considering the future consequences.

The key characteristics of Greedy algorithms are:

1. Optimal substructure: The problem can be broken down into smaller sub-problems, and the optimal solution to the larger problem can be constructed from the optimal solutions of the sub-problems.
2. Greedy choice: The algorithm makes the locally optimal choice at each step, without considering the future consequences.

Examples of Greedy algorithms include:

- Huffman Coding
- Activity Selection Problem
- Coin Changing Problem
- Knapsack Problem (fractional)

While Greedy algorithms can be efficient and intuitive, they don't always produce the optimal solution. However, they can be useful when the problem has optimal substructure and the greedy choice property holds.