E-CONTENT

BSCS 33

DESIGN AND ANALYSIS OF ALGORITHMS

3RD SEMESTER

B.SC COMPUTER SCIENCE

UNIT – I

ALGORITHM AND ANALYSIS

1.1 INTRODUCTION OF AN ALGORITHM

- The term algorithm was coined by Persian Mathematician Al-Khowarizmi in 9th century. He set the simple rules used to perform some calculations either by hand or more usually on a machine.
- An algorithm is a set of instructions designed to perform a specific task. This can be a simple process, such as multiplying two numbers, or a complex operation, such as playing a compressed video file.
- The famous algorithm named Euclid's algorithm which is used to find GCD (Greatest Common Divisor) of two numbers.
- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.

1.2 DEFINITION OF AN ALGORITHM

" The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time. "

1.3 CHARACTERISTICS OF AN ALGORITHM

An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps and their inputs / outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithm must terminate after a finite number of steps.
- **Feasibility** − Algorithm should be feasible with the available resources.

- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.
- **Definiteness** - Each step of an algorithm must be precisely defined.
- **Effectiveness** - An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

1.4 COMPLEXITY OF ALGORITHM

- It is very convenient to classify algorithms based on the relative amount of time or relative amount of space.
- They require and specify the growth of time / space requirements as a function of the input size.
- Thus, we have the notions of Time Complexity and Space Complexity
- **Time Complexity**: Running time of the program as a function of the size of input
- **Space Complexity**: Amount of space required by the algorithm.

1.5 CATEGORIES OF AN ALGORITHM

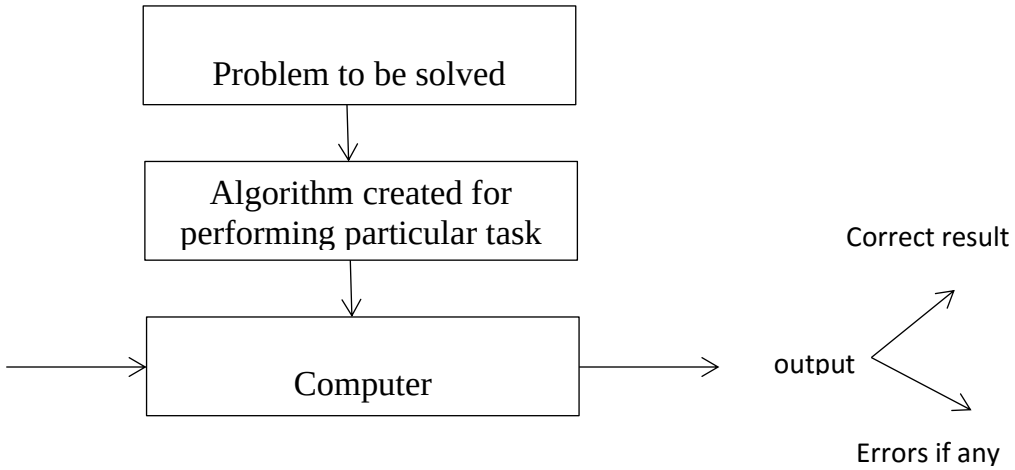
The following are some important categories of algorithms −

1. **Search** − Algorithm to search an item in a data structure.
2. **Sort** − Algorithm to sort items in a certain order.
3. **Insert** − Algorithm to insert item in a data structure.
4. **Update** − Algorithm to update an existing item in a data structure.
5. **Delete** − Algorithm to delete an existing item from a data structure.

1.6 HOW TO WRITE AN ALGORITHM

- First understand the problem.
- After understanding the problem, create an algorithm carefully for given problem.

- Then the algorithm is converted into any one of the computer understandable language and given to device (computer).
- The computer then executes it with some set of input and the result is produced as an output.



1.7 RULES FOR WRITING AN ALGORITHM

- The algorithm is basically divided into two sections-

  Algorithm heading

  Algorithm body

Algorithm heading
It consists of name of algorithm, problem description, input and output

Algorithm Body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement

- Rules

S.N	Heading Section

1	Algorithm is a procedure consists of heading and body. The heading consists of keyword Algorithm and name of the algorithm and parameter list. The syntax is Algorithm name (p1,p2, ….,pn) Keyword name of the algorithm parameter list

2	Then in the heading section we should write the following things
	
	// Problem Description
	// Input
	// Output

S.N	Body Section Written using programming language constructs		
	Statement used in algorithm	**Purpose of the Statement**	**Equivalent Programming language construct**
1	read	Getting input	scanf , cin , gets, ….
2	write	Display Output	printf, cout, puts, …..
3	←	Assignment Operator	=
4	and or not	Logical Operators	&& || !
5	>= > < <= = ≠	Relational Operators	>= > < <= == !=
6	+ - * / %	Arithmetic Operators	+ - * / %
7	Variables first letter must be a alphabet . Keywords not allowed. Example Sum, sum7	Variables , identifiers naming	Rules followed by respective programming language
8	If ( ) then {	Simple if	If ( ) {

	}		}

9	If ( ) then
	{
	
	}
	Else
	{
	
	}
		If – else statement	If ( )
			{
			
			}
			Else
			{
			
			}

10	{ }	Group of instructions enclosed with	{ }

11	for I ⟵1 to n do
	{
	
	}
		Looping Statements	for (i=1;i<=n;i++)
			{
			
			}

12	While(condition)
	{
	
	}
			While(condition)
			{
			
			}

13	Repeat
	{
	
	}
	Until(condition)
			Do
			{
			
			}
			While(condition);

14	//	comment	/* */ multiline comment // single line comment
15	Break	Comes out of loop	break

1.8 EXAMPLES OF AN ALGORITHM

1. Addition of two numbers

```
Algorithm add2num( )
    // it is used to find addition of two numbers
    // Inputs – 2 -  a, b
    // Output – 1 – c
    read a
    read b
    c ⟵ a + b
    write c
```

2. Find the given number is odd / even

```
Algorithm oddeven(  n )
    // it is used to find the given number is odd or even
    // Inputs – 1 -  n
    // Output  - odd / even
    r  ⟵n % 2
    If r = 0  then
        Write " Even Number"
    Else
        Write " Odd Number"
```

3. Sum of first n numbers

```
Algorithm sumn(  n )
    // it is used to find sum of first n numbers
    // Inputs – 1 -  n
    // Output  - sum(n)   - s
    s = 0
    For i ⟵1 to n do
        s ⟵ s + i
    Write s
```

4. Factorial of the given number n

```
Algorithm fact(  n )
    // it is used to find factorial of given number n
    // Inputs – 1 -  n
    // Output  - factorial – f
    f = 1
    For i ⟵1 to n do
        f ⟵ f * i
    Write f
```

5. Find the given number is Positive / Negative

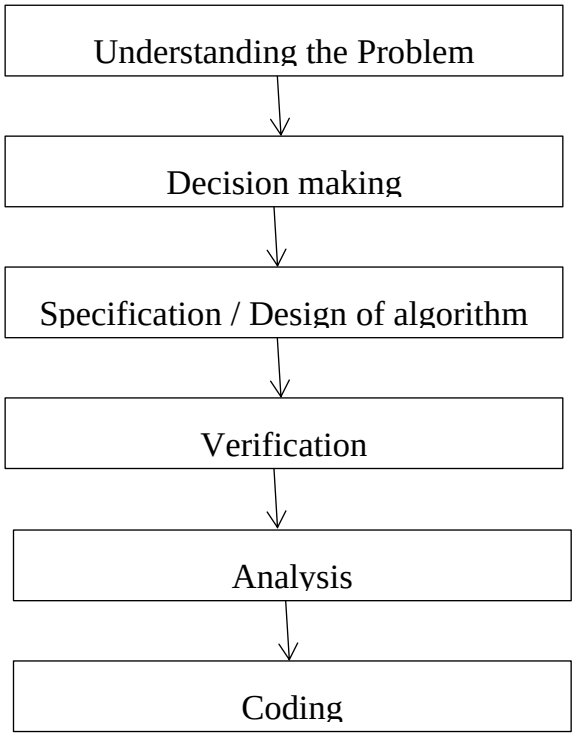
```
Algorithm posneg(  n )
    // it is used to find the given number is +ve / -ve
    // Inputs – 1 -  n
    // Output  - positive / negative
```

```
If n >= 0 then
        Write " Positive Number"
Else

        Write " Negative Number"
```

1.9 ALGORITHM DESIGN

- The steps used to design an algorithm are,
    1. Understanding the problem
    2. Decision Making
    3. **Specification of algorithm**
    4. Algorithm verification
    5. Analysis of an algorithm
    6. Implementation or coding of an algorithm



1. Understanding the problem

   This is the first step in designing the algorithm. Read the problem carefully and ask questions for clarifying the doubts about the problem. Find what the necessary inputs for solving the problem. Input is called instance. It is important to decide the range of inputs. So this is an important step and it should not be skipped at all.

2. Decision making on

- Capabilities of computational devices

  It is necessary to know the computational capabilities of devices on which the algorithm is running. It is a sequential or parallel algorithm. In Sequential algorithm the instructions are executed one after another. In parallel algorithm the instructions are executed in parallel.

- Choice for either exact or approximate problem solving method

  The next decision is to decide whether the problem is to be solved exactly or approximately. If the problem needs to be solved correctly then we need exact algorithm. Otherwise if the problem is so complex that we won't get the exact solution then in that situation we need choose approximation algorithm.

- Data Structure

  Data structure and algorithm work together and these are interdependent. So choice for proper data structure is required before designing the actual algorithm. Array, Linked List, ….

- Algorithmic Strategies

  It is general approach by which many problems can be solved algorithmically. It is also called as algorithmic technique / algorithmic paradigm. Some of the techniques are,

  Brute force **:** straight forward method.

  Divide and Conquer : problem divided into sub problems.

  Greedy technique : to solve problem optimal solution is made.

  Back Tracking **:** this based on trial and error method.

3. Specification of Algorithm

There are various ways by which we can specify an algorithm. They are Flowchart, Natural language and Pseudo code.

Natural Language:

It is very simple to specify an algorithm using natural language.

For example : addition of two numbers

Step1 : read the first number as a

Step 2 : read the second number as b

Step 3 : add the two number and store the result in c

Step 4 : display the result c.

Pseudo code

It is a combination of natural language and programming language constructs.

It is more precise than natural language.

It is more useful from implementation point of view.

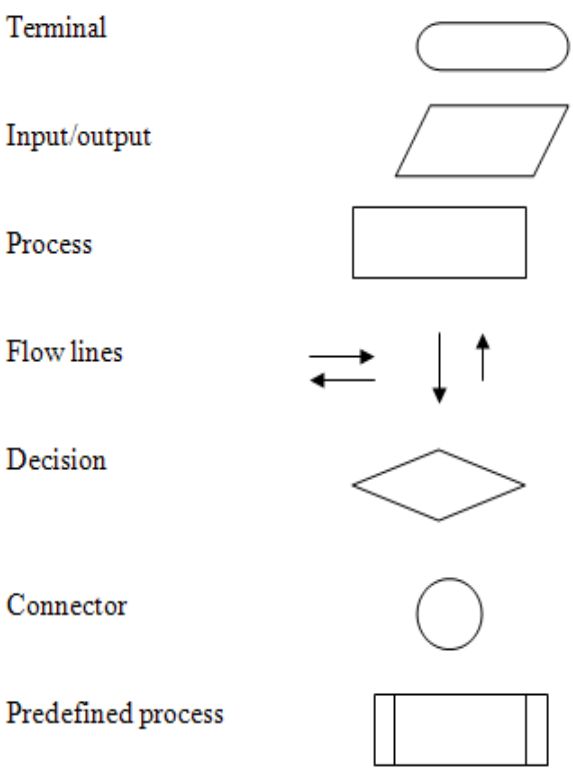
For example : addition of two numbers

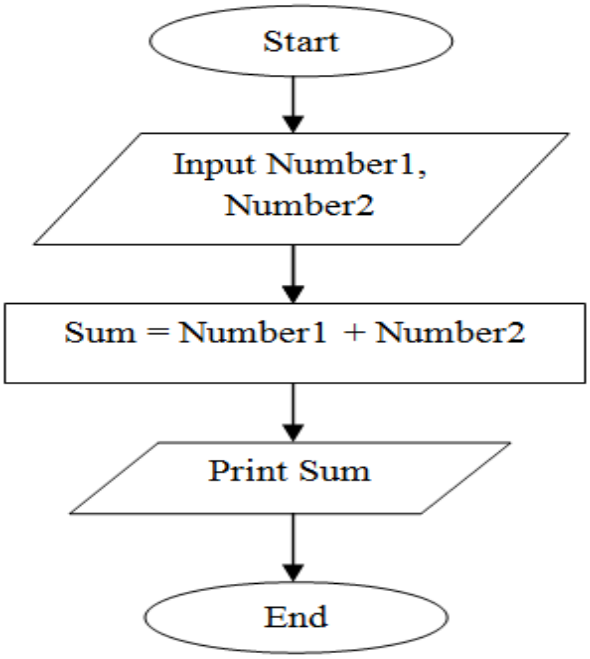
```
Algorithm sum ( )
        //  addition of two numbers
        // Inputs – 2 -  a, b
        // Output – 1 – c
        read a
        read b
        c ⟵ a + b
        write c
```

Flowchat

The graphical / pictorial representation of algorithm is called flowchart.

Typical symbols used in the flowchart are :





4. Verification of algorithm

Algorithm verification means checking correctness of an algorithm. We normally check whether the algorithm gives correct output in finite amount of time for a valid set of input. In this phase check the range of inputs and the outputs.

5. Analysis of algorithm

Factors to be considered while analysing an algorithm are :

- Time efficiency of an algorithm
  Amount of time taken by an algorithm to run. By computing time complexity we come to know that the algorithm is fast or slow.

- Space efficiency of an algorithm
  Amount of space required by an algorithm. By computing space complexity we come to know that the algorithm requires more or less space.

- Simplicity of an algorithm
  Simplicity means generating sequence of instructions which are easy to understand. It is important characteristics of an algorithm.

- Generality of an algorithm

  Generality shows that sometimes it becomes easier to design an algorithm in more general way rather than designing it for particular set of input.

- Range of input

  Range of inputs comes in picture when we execute an algorithm. The design of an algorithm should be such that it should handle the range of input which is the most natural to corresponding problem.

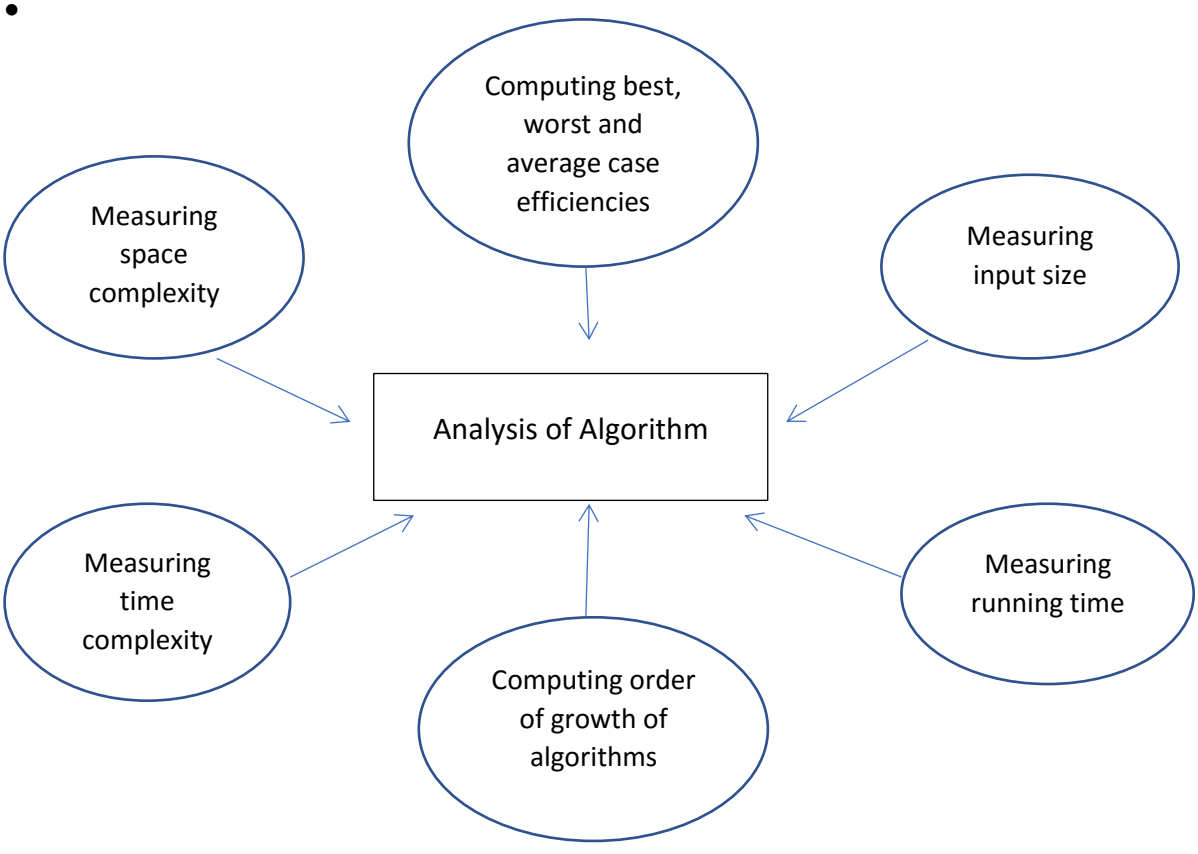
6. Implementation of an algorithm

   It is done by suitable programming language. For example : if an algorithm consists of object and methods then it will be better to implement such algorithm using some object oriented programming language like c++ or JAVA.

1.10 PERFORMANCE ANALYSIS

- The efficiency of an algorithm can be decided by measuring the performance o an algorithm. We can measure the performance of an algorithm by computing two factors.
  - ✓ Amount of **time** required by an algorithm to execute
  - ✓ Amount of **space** required by an algorithm

- This is popularly known as **Time Complexity** and **Space Complexity** of an algorithm.
-



1.10.1 Space complexity of Algorithm

- Space complexity is the amount of memory used by the algorithm to execute and produce the result.
- Algorithm uses memory space for four reasons
  - Instruction Space
    - It is the amount of memory used to save the compiled version of instructions.
    - It is neglected for find space complexity.
  - Environmental Stack Space
    - Sometimes an algorithm maybe called inside another algorithm. In such a situation, the current variables are pushed onto the stack, where they wait for further execution and then the call to the inside algorithm is made.
    - It is also neglected for find space complexity.
  - Data Space / Input Space
    - Amount of space used by the variables and constants.
    - It is needed to find the space complexity.
  - Auxiliary space / Temporary Space
    - It is an extra space or the temporary space used by the algorithm during the execution.
    - It is needed to find the space complexity.
- Therefore Space Complexity = Data Space + Auxiliary Space

  SP = DS + AS or

  SP = IS (input space) + Temporary Space
- Example 1

```
Algorithm add2num( )
        read a
        read b
        c ⟵ a + b
        write c
SP      = DS + AS       //    three variables a , b,c  each 2 bytes so 6 bytes
        = 3 * 2  + 0        // no temporary space
        = 6
```

- Example 2

```
Algorithm add2num( )
        read a
        read b
        c ⟵ a + b
        return  c
SP      = DS + AS       //    three variables a , b,c  each 2 bytes so 6 bytes
        = 3 * 2  + 2      //   temporary space  ie return statement
        = 8
```

1.10.2 Time complexity of Algorithm

- Time complexity of an algorithm is the amount of time required by an algorithm to run to completion.
- It is difficult to compute the time complexity in terms of physically clocked time. The execution time depends on many factors such as
    - System load
    - Number of other programs running
    - Instruction set used
    - Speed of underlying hardware.
- The time complexity is therefore given in terms of frequency count.
- Frequency count is a count denoting number of times of execution of statement.
- If the input size is longer then, the algorithm usually runs for longer time, is known as **measuring the input size**.
- **Measuring running time :**
    - First identify the important operation of an algorithm. This operation is called basic operation.
    - Then compute total number of time taken by this basic operation. It is calculated by the formula,
    - $T(n) = C_{op} + C(n)$ ← no. of times the operation needs to be executed

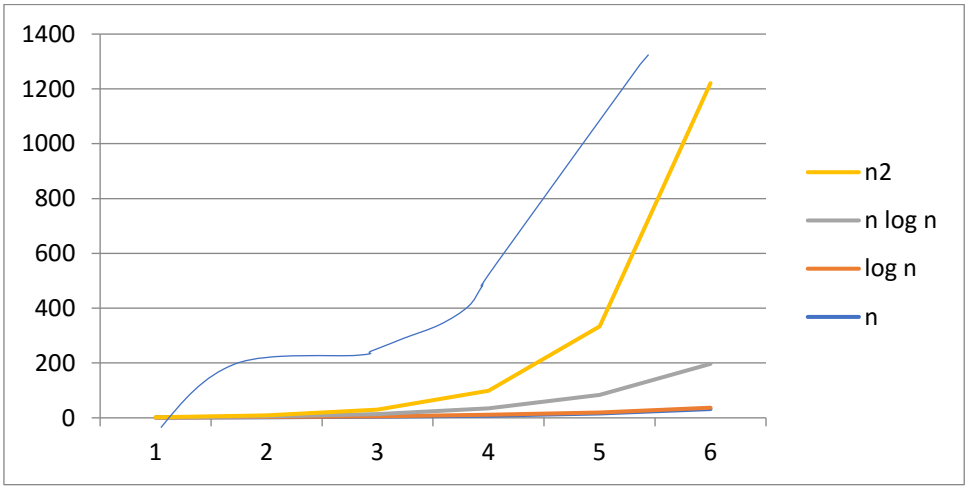
Running time of basic operation time taken by the basic operation

Problem Statement	Input Size	Basic Operation
Searching a key element from the list of n elements	List of n elements	Comparison of key with every element of list
Performing matrix multiplication.	The two matrices with order n x n.	Actual multiplication of the elements in the matrices.
Computing GCD of two numbers	Two numbers	Division

1.10.3 Order of Growth

- Measuring the performance of an algorithm in relation with the input size n is called order of growth. For example, the order of growth for varying input size of n is as given below

n	Log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	250	65536
32	5	160	1024	4,294,967,296



1.10.4 Time Space Tradeoff

- Time space tradeoff basically a situation where either space efficiency can be achieved at the cost of time or timer efficiency can be achieved at the cost of memory.
- Example 1
    - Every program symbol table is created for storing variables and constants.
    - If the symbol table is stored in the program then the time required for searching / storing the variable in the symbol table will be reduced but memory requirement will be more.

        Space Complexity = high

        Time Complexity = low

- On the other hand if we do not store the symbol table in the program then memory will be reduced but the processing time will be more.

  Space Complexity = low

  Time Complexity = high

- Example 2
  - Suppose, in a file, if we store the uncompressed data then reading the data will be an efficient job.

    Space Complexity = high

    Time Complexity = low
  - But if the compressed data is stored then to read such data required more time.

    Space Complexity = low

    Time Complexity = high
- Example 3
  - Reversing the order of elements. If an array A has n elements in ascending order then we have to reverse those elements in descending order. This can be done in two ways.
  - Using another array B store the A array elements in reverse direction. It increase memory but time will be reduced.

    Space Complexity = high

    Time Complexity = low
  - Apply extra logic for the same array to store the elements in reverse direction. It increase time but memory is reduced.

    Space Complexity = low

    Time Complexity = high

1.11 RANDOMIZED ALGORITHM

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- For example, in Randomized Quick Sort, we use random number to pick the next pivot element
- Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.
- For example consider below a randomized version of QuickSort.

randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.

(i) Choose uniformly at random a number from [low..high].
Let the randomly picked number number be **x**.

(ii) Count elements in arr[low..high] that are smaller than arr[x].
Let this count be **sc**.

(iii) Count elements in arr[low..high] that are greater than arr[x].
Let this count be **gc**.

(iv) Let **n** = (high-low+1).
If sc >= n/4 and gc >= n/4, then
x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. randQuickSort(arr, low, sc-1) // Recur for smaller elements

5. randQuickSort(arr, high-gc+1, high) // Recur for greater elements

- The important thing in our analysis is, time taken by step 2 is O(n).

1.12 ASYMPTOTIC NOTATIONS

- Asymptotic notation of an algorithm is a mathematical representation of its complexity
- Asymptotic analysis of an algorithm refers to defining the mathematical foundation / framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
- For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly

with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases.

- Similarly, the running time of both operations will be nearly the same if **n** is significantly small.
    - Usually, the time required by an algorithm falls under three types –
        - **Best Case** − Minimum time required for program execution.
        - **Average Case** − Average time required for program execution.
        - **Worst Case** − Maximum time required for program execution.
- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
    - ✓ **Big Oh** **- O** Notation - Upper bound - Worst Case
    - ✓ **Big Omega** **- Ω** Notation - Lower bound - Best Case
    - ✓ **Big Theta** **- θ** Notation - Average bound - Average Case
    - ✓ **Small Oh** **– o** Notation
    - ✓ **Small Omega - ω** Notation

- Majorly, we use THREE types of Asymptotic Notations and those are as follows...
    - ✓ **Big Oh** **- O** Notation
    - ✓ **Big Omega** **- Ω** Notation
    - ✓ **Big Theta** **- θ** Notation

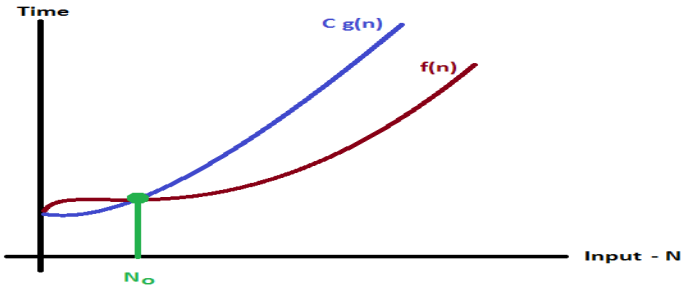
Big - Oh Notation (O)

- ✓ Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- ✓ That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
- ✓ Big - Oh Notation can be defined as follows...

Consider function **f(n)** the time complexity of an algorithm and **g(n).**
If **f(n) <= C g(n)** for all **n >= n_0 >= 1**, **C > 0**.
Then we can represent **f(n)** as **O(g(n))**.

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

- ✓ Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- ✓ In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.
- ✓
- ✓ Example 1

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

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

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

  Solution:

  f(n) = **O(g(n))**

  f(n) <= C g(n)

  3n + 2 <= C n

  Above condition is always TRUE for all values of **C = 4** and **n >= 2**.

  3n + 2 <= 4n
  3(2) + 2 <= 4(2)
  6+2 <= 8
  8 <= 8

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

  3n + 2 = O(n)

Big - Omege Notation (Ω)

- ✓ Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- ✓ 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.
- ✓ Big - Omega Notation can be defined as follows...

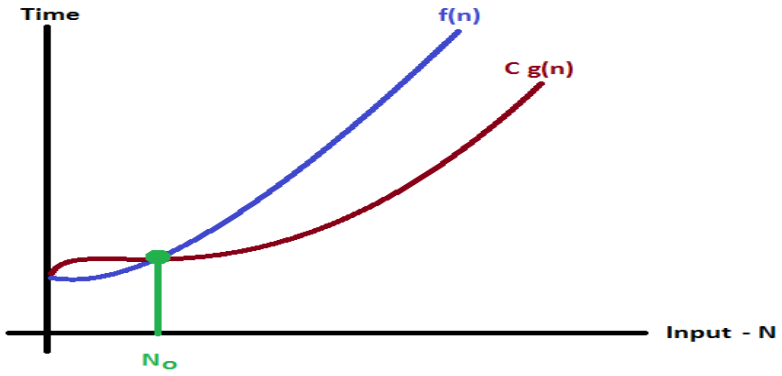
  Consider function **f(n)** the time complexity of an algorithm and **g(n)**.

  If **f(n) >= C g(n)** for all **n >= n_0 >= 1**, **C > 0**.

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

f(n) = Ω(g(n))

- ✓ Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- ✓ In above graph after a particular input value n_0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

- ✓ **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**
Solution:

f(n) = **Ω (g(n))**

f(n) >= C g(n)

3n + 2 >= C n

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

3n + 2 >= 1n
3(1) + 2 >= 1(1)
3+2 >= 1
5 >= 1

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

3n + 2 = Ω(n)

Big - Theta Notation (Θ)

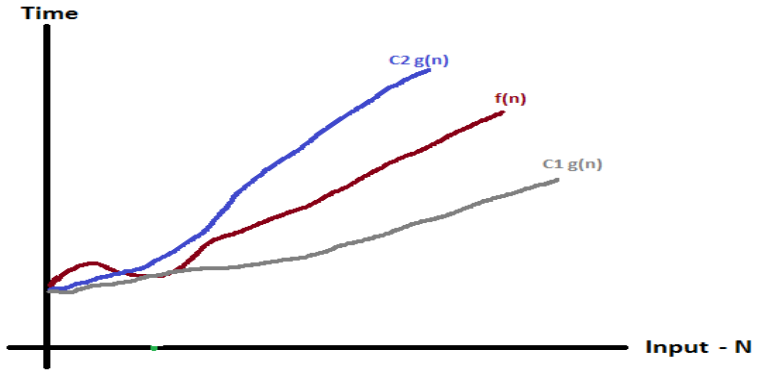
- ✓ Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

- ✓ That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.
- ✓ Big - Theta Notation can be defined as follows...

  Consider function **f(n)** the time complexity of an algorithm and **g(n)**.
  If **C_1 g(n) <= f(n) >= C_2 g(n)** for all **n >= n_0**, **C_1, C_2 > 0** and **n_0 >= 1**.
  Then we can represent **f(n)** as **Θ(g(n))**.

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

- ✓ Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



- ✓ In above graph after a particular input value n_0, always **C_1 g(n)** is less than **f(n)** and **C_2 g(n)** is greater than **f(n)** which indicates the algorithm's average bound.

- ✓ **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, C_2 > 0** and **n>= 1**
  Solution
  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**.
1n <= 3n+2 >= 4n
1(1) <= 3(1)+2 >= 4(1)
1 <= 5 >= 4

By using Big - Theta notation we can represent the time complexity as follows...
3n + 2 = Θ(n)

2 MARK QUESTIONS

1. Define Algorithm.
   An algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.

2. What are the 2 kinds of Algorithm Efficiency
   Time Efficiency-How fast your algorithm runs?
   Space Efficiency-How much extra memory your algorithm needs?

3. How can you specify Algorithms?
   Algorithms can be specified natural language or pseudo code.

4. Differentiate Time Efficiency and Space Efficiency?
   Time Efficiency measured by counting the number of times the algorithms basic operation is executed.
   Space Efficiency is measured by counting the number of extra memory units consumed by the algorithm.

5. What are the features of efficient algorithm?
   • Free of ambiguity
   • Efficient in execution time
   • Concise and compact Completeness
   • Definiteness Finiteness

6. Define Order of Algorithm
   The order of algorithm is a standard notation of an algorithm that has been developed to represent function that bound the computing time for algorithms.The order of an algorithm is a way of defining its efficiency. It is usually referred as O-notation.

7. What are the different types of time complexity?
   The time complexity can be classified into 3 types, they are
   • Worst case analysis
   • Average case analysis
   • Best case analysis

8. What are the three different algorithms used to find the gcd of two numbers?
   The three algorithms used to find the gcd of two numbers are
   - Euclid‘s algorithm
   - Consecutive integer checking algorithm

- Middle school procedure

9. Mention some of the important problem types?

   Some of the important problem types are as follows
   - Sorting
   - Searching
   - String processing
   - Graph problems
   - Combinatorial problems
   - Geometric problems
   - Numerical problems

10. What is order of growth?

    Measuring the performance of an algorithm based on the input size n is called order of growth.

5 MARK QUESTIONS

1. Write short notes on Asymptotic notations
2. Discuss Algorithmic Strategies
3. Explain how algorithm can be specified.
4. Differentiate Space Complexity and Time Complexity.
5. What is meant by order of growth.

10 MARK QUESTIONS

1. Briefly explain the steps involved in algorithm design
2. Explain in detail about rules for writing an algorithm
3. Explain randomized algorithm in detail with example
4. Briefly Explain Performance analysis
5. Discuss Time Space Trade off.