Week2:

Algorithm Analysis – Space Complexity, Time Complexity. Run time analysis. Asymptomatic notations, Big-O Notation, Omega Notation, Theta Notation.

ALGORITHM:

- ➢ 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.
- ➢ Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.
- ➢ From the data structure point of view, following are some important categories of algorithms –
  - Search − Algorithm to search an item in a data structure.
  - Sort − Algorithm to sort items in a certain order.
  - Insert − Algorithm to insert item in a data structure.
  - Update − Algorithm to update an existing item in a data structure.
  - Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics. An algorithm should have the following characteristics:

- Clear and Unambiguous: Algorithm should be clear and 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.
- Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- Finite-ness: The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- Feasible: The algorithm must be simple, generic and practical, such that it can be executed upon will 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 same, as expected.

Advantages and Disadvantages of Algorithm

Advantages of Algorithms:

 It is easy to understand.

- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps therefore it easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Different approach to design an algorithm

1. Top-Down Approach: A top-down approach starts with identifying major components of system or program decomposing them into their lower-level components iterating until desired level of module complexity is achieved. In this we start with topmost module & incrementally add modules that it calls.

2. Bottom-Up Approach: A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components. Starting from very bottom, operations that provide layer of abstraction are implemented

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Instead it is problem and resource dependent. Algorithms are never written to support a particular programming code.
- As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write algorithm
- Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.
- Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

First Method:

Step 1 − START

Step 2 − declare three integers a, b & c

Step 3 − define values of a & b

Step 4 − add values of a & b

Step 5 − store output of step 4 to c

Step 6 − print c

Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

Second Method:

Step 1 − START ADD

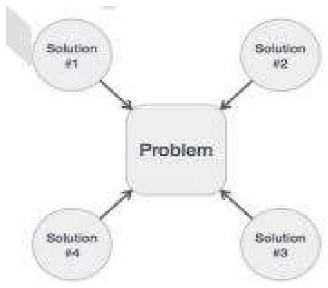
Step 2 − get values of a & b

Step 3 − c ← a + b

Step 4 − display c

Step 5 − STOP

- In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.
- We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways as shown below in the diagram



ALGORITHM COMPLEXITY

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

- Space Factor − Space is measured by counting the maximum memory space

required by the algorithm.

The complexity of

an 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.

Time Complexity

- Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by

the algorithm in its life cycle. The space required by an algorithm is equal to the sum of

the following two components −

1) A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
2) A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part

and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Following is a simple example that tries to explain the concept −

```
Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now,

space depends on data types of given variables and constant types and it will be multiplied accordingly.

ALGORITHM ANALYSIS

Efficiency of an algorithm can be analysed at two different stages, before implementation and after implementation. They are the following –

1) **A Priori Analysis or Asymptotic Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
2) **A Posterior Analysis or Performance Measurement** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

- Algorithm analysis deals with the execution or running time of various operations involved.
- The running time of an operation can be defined as the number of computer instructions executed per operation.
- Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.
- Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.
- In theoretical analysis of algorithms, it is common to estimate their complexity in the
- asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big Oh notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

1) **Best Case Analysis:**
    - If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best-case time complexity.
    - The best-case efficiency of an algorithm is the efficiency for the best case input of size n. Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.
2) **Worst Case Analysis :**

- If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity.
- The worst case efficiency of an algorithm is the efficiency for the worst case input of size n. The algorithm runs the longest among all the possible inputs of
- the similar size because of this input of size n

3) Average Case Analysis

- If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.
- Average case analysis provides necessary information about an algorithm's behaviour on a typical or random input. You must make some assumption about the possible inputs of size n to analyze the average case efficiency of algorithm.

Asymptotic Notations

- A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.
- To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity.
- The asymptotic notations help to represent the time complexity in a shorthand way.
- It can generally be represented as the fastest possible, slowest possible or average possible.
- The notations such as O (Big-O), Ώ (Omega), and θ (Theta) are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

Big-O Notation

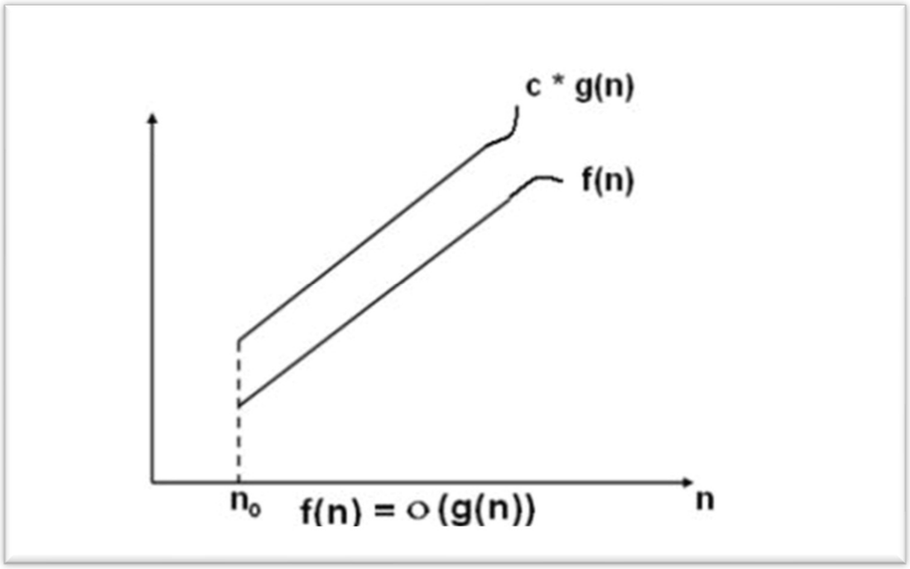
- 'O' is the representation for Big-O notation.
- Big -O is the method used to express the upper bound of the running time of an algorithm.
- Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm
- This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem.
- Big-O is defined as: $f(n) \leq c * g(n)$

  where, n can be any number of inputs or outputs

  f(n) as well as g(n) are two non-negative functions.

Note: These functions are true only if there is a constant c and a non-negative integer n0 such that,n ≥ n0.

- The Big-O can also be denoted as f(n) = O(g(n)), where f(n) and g(n) are two non -negative functions and f(n) < g(n) if g(n) is multiple of some constant c.
- The graphical representation of f(n) = O(g(n)) is shown in graph as shown below, where the running time increases considerably when n increases.



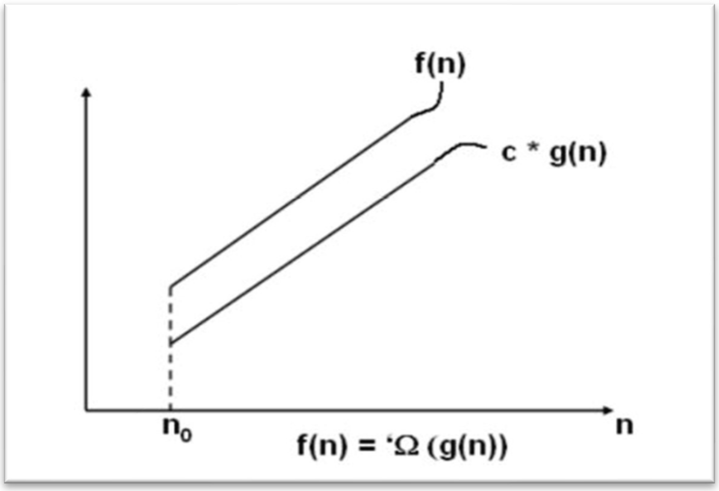
Omega Notation

- 'Ω' is the representation for Omega notation.
- Omega describes the manner in which an algorithm performs in the best case time complexity.
- This notation provides the minimum amount of time taken by an algorithm to compute a problem. Thus, it is considered that omega gives the "lower bound" of the algorithm's run-time.
- Omega is defined as:f(n) ≥ c ∗ g(n)

  Where, n is any number of inputs or outputs and f(n) and g(n) are two non-negative

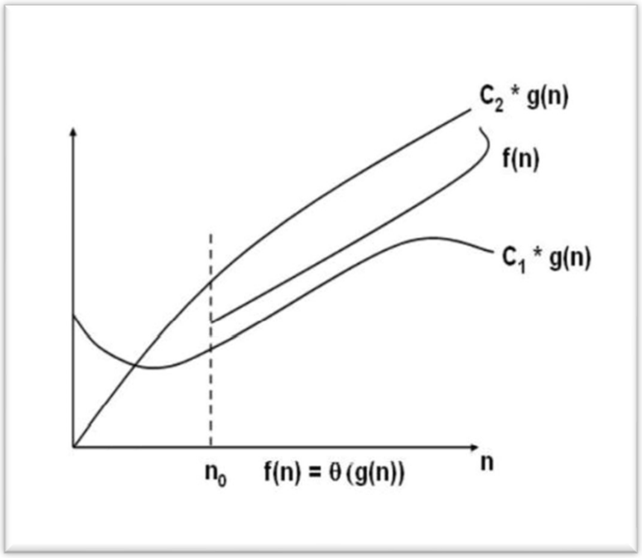
  functions. These functions are true only if there is a constant c and a non-negative integer n0 such that n>n0

- Omega can also be denoted as f(n) = Ώ (g(n)) where, f(n) is equal to Omega of g(n).The graphical representation of f(n) = Ώ (g(n)) is shown below.



Theta Notation:

- 'θ' is the representation for Theta notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:
  $c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all n>n0
  Where, n is any number of inputs or outputs and f(n) and g(n) are two non-negative functions. These functions are true only if there are two constants namely, c1, c2, and a non-negative integer n0.
- Theta can also be denoted as f(n) = θ(g(n)) where, f (n)is equal to Theta of g(n). The graphical representation of f(n) = θ(g(n)) is as shown below.


C_2 * g(n)
f(n)
C_1 * g(n)
n_0
f(n) = θ (g(n))
n