WEEK – 02

ALGORITHM ANALYSIS

- **Algorithm**
- Algorithm is a step-by-step procedure.
- It defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of languages.

- **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**
- Unambiguous − Algorithm should be clear and unambiguous.
- 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 − Algorithms must terminate after a finite number of steps.
- Feasibility − Should be feasible with the available resources

- **Algorithm Analysis**

- Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation.
- A Priori 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.
- A Posterior Analysis − 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.

Space Complexity

- Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.
- For calculating the space complexity, we need to know the value of memory used by different type of data type variables.

- Ex: int z = a + b + c
- In the above expression a, b, c and z are all integer variables.
- Hence memory required for it is 4*4 i,e 16 bytes

Time Complexity

- Time complexity of an algorithm represents total time required by the program to run till its completion.
- Time Complexity is most commonly estimated by counting the number of steps performed by any algorithm to finish execution.
- The time complexity of algorithms is most commonly expressed using the asymptotic notations.

Example 1

- Lets consider code to print “Hello world”.
- print(“Hello world”)
- Above code “Hello World” is printed only once on the screen.
- So, time complexity is constant: O(1) i.e. every time constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.

Example 2

```
i, n = 8;
  for i in range(n):
       print("Hello Word")
```

In the above code “Hello World” is printed only n times on the screen, as the value of n can change. So, time complexity is linear: O(n) i.e. every time linear amount of time is required to execute code.

Example 3

```
i,j n = 8;
  for i in range(n):
         for j in range(n):
       print("Hello Word")
```

- In the above code, both outer and inner for loops are executed n times each.
- So, the total number of times the statement got executed is
- $T(n) = n * n$ i.e. n^2
- Hence the time complexity for nested for loops is $O(n^2)$

Asymptotic Notations

- These are the notations used to represent the running time or time complexity of an algorithm.

Three types are:

- Big O Notation (O Notation)
- Omega Notation (Ω Notation)
- Theta Notation (Θ Notation)

Understanding these notations, requires the knowledge of three things.

- Worst case efficiency – in which the algorithm runs for longer duration of time for the given input of size n.
  Ex: In searching operation, if the key element is present at the last position or if key element is not present, then algorithm will take longer time.

- Best case efficiency – in which the algorithm runs for short duration of time for the given input of size n.
  Ex: In searching operation, if the key element is present at the first position, then algorithm will take short time.

- Average case efficiency – it is between the above two cases.
  Ex: In searching operation, if the key element is present at the middle position, then algorithm will take average time.

Big O Notations

- Big-O notation represents the upper bound of the running time of an algorithm.
- Thus, it gives the worst-case complexity of an algorithm.

Omega Notations

- Omega notation represents the lower bound of the running time of an algorithm.
- Thus, it provides the best case complexity of an algorithm.

Theta Notations

- It represents the upper and the lower bound of the running time of an algorithm.
- It is used for analyzing the average-case complexity of an algorithm.