**Data Structures**

**and**

**Algorithms**

Firstly, before getting into data structures and algorithms we need to understand some important topic how these data structures and algorithms are measured / rated in terms of efficiency and performance.

This means we need to study about the metrics used to evaluate the efficiency and performance of data structures and algorithms.

Mainly, the performance of any data structure or algorithm is rated based on two metrics.

1. Time complexity.
2. Space complexity.

**Time complexity:**

Time complexity means the time taken by the any data structure or algorithm to accomplish the given task.

**Space complexity:**

Space complexity means how much space/memory occupied by any data structure or algorithm to accomplish the given task.

**Note:** However, time and space complexity depend on various other factors like hardware, operating system, processor etc. But we don’t consider these factors while calculating the complexity of a data structure or algorithm.

**Order of growth:**

As the name suggest it tells us in which order the execution time or space/memory occupied by a data structure or algorithm is increasing as we scale up the input.

Example: the execution time of an algorithm is increasing with the increasing in the input.

If input = 10 ==🡺 execution time = 100 milli seconds.

If input = 20 ==🡺 execution time = 200 milli seconds.

Here the execution time is increasing linearly with the input. So, order of growth is linear. The same will be applicable for space complexity also.

To represent the time complexity mathematically we will use asymptotic notations. Asymptotic notations are mathematical tools to represent time complexity of any data structure or algorithm for asymptotic analysis.

We will use following 3 asymptotic notations more frequently.

1. Big-O Notation. ===🡺 upper bound
2. Theta(Ɵ) Notation. ===🡺 in between upper and lower bound
3. Omega(Ω) Notation. ===🡺 lower bound

**Big-O Notation:**

Big-O notation represents the upper bound of the running time of an algorithm or data structure. This means it gives the maximum time or space taken by an algorithm or data structure.

Thus, it gives the worst-case complexity of an algorithm.

**Omega Notation:**

Omega notation represents the lower bound of the running time of an algorithm or data structure. This means it gives the minimum time or space taken by an algorithm or data structure.

Thus, it provides best case complexity of an algorithm.

**Theta Notation:**

Theta Notation represents the middle bound of the running time of an algorithm or data structure. This means it gives the average time or space taken by an algorithm or data structure.

Thus, it provides average case complexity of an algorithm.

Mostly, we are interested in finding the maximum time required by an algorithm or data structure (i.e., Big-O Notation).

If the worst-case complexity / maximum time required is less then it’s more efficient.

**Most commonly used time complexity equations**

We will use 6-time complexity equations more frequently.

1. O (1) ===🡺 Constant Time Complexity (Best)
2. O (n) ===🡺 Linear Time Complexity
3. O (log n) ===🡺 Logarithmic Time Complexity
4. O (n log n) ===🡺 Logarithmic Time Complexity
5. O (n2) ===🡺 Exponential Time Complexity
6. O (2n) ===🡺 Exponential Time Complexity
7. O (n!) ===🡺 Factorial Time Complexity

If we arrange the equations based on time complexity from best to worst.

O (1) === Rank 1 (Best)

O (log n) === Rank 2 (Better)

O (n) === Rank 3 (Good)

Not good to use after Rank 3

O (n log n) === Rank 4 (Bad)

O (n2) === Rank 5 (Very bad)

O (2n) === Rank 6 (Very bad)

O (n!) === Rank 7 (Worst)

The value in the parenthesis represents the number of operations to perform before the function can finish.

In O (1) time complexity the function has to perform only 1 operation, no matter how big or small the input is.

In O(n) time complexity the function has to perform ‘n’ operations. So, operations has to perform will increase with the size of input.

**Properties of Asymptotic Notations:** Not that important

**1.** **General Properties**

If

f(n) ===🡺 O(g(n))

then

a\*f(n) ===is also===🡺 O(g(n))

where

a = constant.

**Example:**

f(n) = 2n²+5 ===🡺 O(n²) Hence, Highest degree of polynomial is 2.

then

7 \* f(n) = 7(2n²+5) = 14n²+35 ===is also===🡺 O(n²)

Similarly, this property satisfies for both Theta(Θ) and Omega(Ω) notation also.

**2. Reflexive Properties**

If

f(n) is given

then

f(n) is O(f(n)).

**Example:**

 f(n) = n²

then

O(n²) i.e. O(f(n))

Similarly, this property satisfies for both Θ and Ω notation.

**3. Transitive Properties**

If

f(n) ===🡺 O(g(n)) **AND** g(n) ===🡺 O(h(n)).

then

f(n) = O(h(n)).

This is simple like a == b == c Then a == c.

**Example:**

If

f(n) = n

g(n) = n²

h(n)=n³

n ===🡺 O(n²) and n² ===🡺 O(n³)

then

n is O(n³)

Similarly, this property satisfies for both Θ and Ω notation.

**4. Symmetric Properties**

If

f(n) ===🡺 Θ(g(n))

then

g(n) ===🡺 Θ(f(n))

This is simply like if a == b then b == a.

**Example:**

If f(n) = n² and g(n) = n²

then f(n) = Θ(n²) and g(n) = Θ(n²)

**This property only satisfies for Θ notation.**

**5. Transpose Symmetric Properties**

If f(n) ===🡺 O(g(n))

Then g(n) ===🡺 Ω (f(n)).

**Example:**

If f(n) = n

g(n) = n²

then n ===🡺 O(n²) and n² ===🡺 Ω (n).

**This property only satisfies for O and Ω notations**.

**6. Some More Properties:**

1. If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))
2. If f(n) = O(g(n)) and d(n)=O(e(n))  
   then f(n) + d(n) = O( max( g(n), e(n) ))  
   Example: f(n) = n i.e O(n)  
   d(n) = n² i.e O(n²)  
   then f(n) + d(n) = n + n² i.e O(n²)
3. If f(n)=O(g(n)) and d(n)=O(e(n))  
   then f(n) \* d(n) = O( g(n) \* e(n) )  
   Example: f(n) = n i.e O(n)  
   d(n) = n² i.e O(n²)  
   then f(n) \* d(n) = n \* n² = n³ i.e O(n³)