**ADS Assignment 1**

1. Define Ɵ in terms of asymptotic complexity with examples.

Sol : - In the realm of analysing algorithmic efficiency, the symbol "Ɵ" is employed to denote asymptotic complexity. This notation facilitates the description of how an algorithm's performance changes as its input size becomes infinitely large. It signifies both an upper and lower boundary that effectively encapsulates the rate of growth of the function within a constant factor. Thus, the runtime or resource usage of an algorithm can be succinctly expressed as Ɵ(g(n)), where "g(n)" represents the function portraying the algorithm's performance in relation to the input size "n."

To elaborate formally, a function f(n) falls under the category of Ɵ(g(n)) if there are positive constants c1, c2, and n0, satisfying the condition that for all n greater than or equal to n0, c1 multiplied by g(n) is less than or equal to f(n), and f(n) is less than or equal to c2 multiplied by g(n). Essentially, this implies that for sufficiently large inputs, the function f(n) experiences growth akin to the function g(n) within certain unchanging constant factors.

\Here are some illustrative examples:

1. Constant Time: Consider an algorithm that executes a single operation irrespective of input size. For instance, fetching an element from an array using an index. In this scenario, the runtime is Ɵ(1), since the operation count remains unaffected by input size.

2. Linear Time: Envisage an algorithm that traverses through an array of size "n" and carries out a constant-time action on each element. The algorithm's runtime scales linearly with input size, yielding an asymptotic complexity of Ɵ(n).

3. Quadratic Time: Imagine an algorithm involving nested loops, both iterating "n" times. Such a construct would lead to a runtime of Ɵ(n^2), reflecting a quadratic growth in operations as input size expands.

4. Logarithmic Time: Binary search typifies an algorithm with logarithmic runtime complexity. At each step, the search space is halved. Consequently, the runtime becomes Ɵ(log n), where the operation count increases logarithmically with input size.

5. Linearithmic Time: Algorithms like merge sort exhibit a runtime complexity of Ɵ(n log n). They combine both linear and logarithmic growth rates, frequently observed in effective sorting and divide-and-conquer algorithms.

6. Exponential Time: Algorithms demanding repeated operations on input, such as recursively computing Fibonacci numbers without optimization, can result in exponential runtime. For instance, calculating the nth Fibonacci number via naive recursion yields a runtime of Ɵ(2^n).

The Ɵ notation offers a coherent comprehension of how algorithms scale as input size escalates, while accounting for both the upper and lower performance bounds.

1. Define Ω in terms of asymptotic complexity with examples.

Sol - In the domain of analyzing asymptotic complexity, the notation "Ω" (pronounced as "omega") is employed to indicate the lower bound of an algorithm or function's growth rate as its input size trends towards infinity. It conveys that the performance of a function won't dip below a certain lower growth rate, characterizing the best-case or minimal behaviour of an algorithm.

To elaborate formally, a function f(n) is categorized as belonging to Ω(g(n)) if there exists a positive constant denoted as "c" and a specific value of "n₀" such that, for all "n" greater than or equal to "n₀," the inequality "c \* g(n) ≤ f(n)" holds true. Essentially, this signifies that the function g(n) serves as a lower limit on the growth rate of f(n) for sufficiently large input sizes.

Here are illustrative examples:

1. Lower Bound for Linear Time: Imagine an algorithm that necessitates examining each element of an array for sorting. The optimal situation entails linear time complexity, as it cannot perform faster than that. In this context, the algorithm's behavior aligns with Ω(n), depicting a lower bound coinciding with linear growth.

2. Lower Bound for Quadratic Time: Consider an algorithm that mandates comparing all possible pairs of elements within an array. Even in the best-case scenario, where minimal comparisons are needed, the time complexity remains quadratic. This gives rise to a Ω(n^2) lower bound.

3. Lower Bound for Logarithmic Time: For algorithms that search a sorted array using a divide-and-conquer strategy, the best-case situation emerges when the target is found instantly. Here, the algorithm's performance corresponds to Ω(log n), signalling a lower bound in harmony with logarithmic growth.

4. Lower Bound for Linearithmic Time: Algorithms that engage in operations on every input element at a minimum, such as merge sort, possess a best-case time complexity of Ω(n log n), as this marks the lower threshold for divide-and-conquer algorithms of this nature.

5. Lower Bound for Constant Time: Algorithms executing a fixed number of operations at all times exhibit a Ω(1) lower bound, indicating that their growth rate remains above constant time complexity.

The Ω notation serves as a tool for comprehending the least resources required as input size escalates, thereby offering insight into the minimal requirements of algorithms.

1. Derive Big Oh(O) for the following functions.
2. f(x)=5n^2+2n

Sol- In the function f(x)=5n^2+2n, the term 5n^2 dominates as n becomes larger. The term 2n is less significant in comparison.

We're interested in the most significant term that determines the growth rate. So, we can drop the 2n term.

We can ignore the coefficient 5 as well.

After removing lower order terms and constants, we're left with the dominant term n^2. Thus, the big O notation for f(x) is O(n^2).

f(x)=5n2+2n is O(n^2)

1. f(x)=100n^100+10

Sol - f(x) ≈ 100n^100 g(x) = n^100

Now, we need to find constants C and x0 such that:

|f(x)| ≤ C \* |g(x)|

|100n^100| ≤ C \* |n^100|

100n^100 ≤ C \* n^100

Dividing both sides by n^100 (assuming n > 0):

100 ≤ C

So, we have found a constant C (C = 100) that satisfies the inequality for all n greater than some x0. This means that f(x) is O(n^100).

In summary, the big O notation for the function f(x) = 100n^100 + 10 is O(n^100).

f(x) = 100n^100 + 10 = O(n^100)

1. f(x)=2^n+15n^3

Sol - f(x)=2^n+15n^3

2^n is an exponential function, and it grows extremely fast as n increases. It will dominate the growth rate of the function for large values of n.

15n^3 is a polynomial function of n and grows at a much slower rate compared to 2^n.

When we're looking for the Big O notation of a function, we're interested in the dominant term that determines the growth rate. In this case, 2^n is the dominant term. Therefore, the Big O notation for f(x) is O(2^n).

f(x)=2^n+15n^3= O(2^n)

1. f(x)=2^n-100n^2

Sol – 2^n: This term grows exponentially with n.

−100n^2: This term is a quadratic polynomial that decreases as n increases.

For large values of n, the exponential term 2^n will dominate the growth of the function. This means that the quadratic term −100n^2 will become relatively insignificant compared to the exponential growth of 2^n.

Therefore, the big O notation for f(x) is O(2^n), indicating an exponential growth rate dominated by the exponential term.

f(x)=2^n-100n^2 = O(2^n)