Models of Computations and Complexity Measures

To analyze an algorithm's efficiency, we use **models of computation** and **complexity measures**. These help to quantify how well an algorithm performs in terms of time and space under different conditions.

Asymptotic Notations

Asymptotic notations describe the growth of an algorithm's time or space complexity in relation to the input size nnn. The most commonly used notations are:

- **Big O Notation (O(f(n)))**: Describes the **worst-case** time or space complexity. It gives an upper bound on the growth rate of the algorithm.
- **Big Omega Notation (Ω(f(n)))**: Describes the **best-case** time or space complexity. It provides a lower bound on the growth rate.
- **Theta Notation (Θ(f(n)))**: Describes the **average-case** or **tight bound**. It defines both upper and lower bounds.

1.1 Time Complexity

Time complexity represents the amount of time an algorithm takes as a function of the input size, n. Time complexity is generally analyzed in three cases:

1.1.1 Worst Case Time Complexity

- The **worst-case** time complexity gives the **maximum** amount of time an algorithm will take to run on any input of size n.
- It represents the upper bound, i.e., the maximum number of steps taken by the algorithm in the worst-case scenario.

Example: For **linear search**, the worst-case occurs when the element to be found is at the last position or not present in the array. If there are n elements, the time complexity is O(n).

1.1.2 Best Case Time Complexity

- The **best-case** time complexity gives the **minimum** time the algorithm will take on an input of size n.
- It represents the lower bound, i.e., the minimum number of steps the algorithm will take.

Example: In a **linear search**, the best case occurs when the element to be found is at the first position. Thus, the time complexity is O(1).

1.1.3 Average Case Time Complexity

- The **average-case** time complexity gives the expected time an algorithm takes for a random input of size n.
- It is the average number of steps the algorithm will take over all possible inputs.

Example: In a **linear search**, if we assume the element is equally likely to be at any position, on average, it will take n/2 steps. Thus, the average time complexity is O(n).

1.2 Space Complexity

Space complexity measures the amount of memory an algorithm uses as a function of the input size n. Like time complexity, it can be analyzed in worst-case, best-case, and average-case scenarios.

Space complexity takes into account both:

1. **Auxiliary space**: Temporary extra space or memory required by an algorithm.
2. **Input space**: Memory required for the input.

---

2. Performance Analysis Based on Measured Complexity

The performance of an algorithm can be analyzed by studying its time and space complexity. The key steps in **performance analysis** include:

1. **Identifying Input Size**: Determine the size of the input, typically denoted by n.
2. **Identifying the Basic Operation**: The basic operation is the part of the algorithm that contributes the most to the running time (e.g., comparisons, assignments).
3. **Counting the Basic Operations**: Count how often the basic operation is executed as a function of n.
4. **Classifying the Complexity**: Use asymptotic notations (Big O, Big Omega, or Theta) to classify the time or space complexity of the algorithm.

---

3. Calculation of Time Complexity

To calculate the time complexity, follow these steps:

1. **Identify the Loop Structure**:
   - **Single loop**: If there is a single loop that runs n times, the time complexity is O(n).
   - **Nested loops**: If there are nested loops, multiply the number of iterations for each loop.
2. **Identify Recursion**:
   - **Divide and Conquer algorithms** often involve recursive functions. In these cases, use **recurrence relations** to solve the time complexity.
3. **Additive vs Multiplicative Time Complexities**:
   - **Additive**: When two independent parts of the algorithm are executed, their complexities are added. For example, if one part takes O(n) and another takes O(m), the total time complexity is O(n+m).
   - **Multiplicative**: If one part of the algorithm depends on another, their complexities are multiplied. For example, a nested loop runs nnn times inside another loop of n, resulting in $O(n \times n) = O(n^2)$.

Examples:

Single Loop

```java
for (int i = 0; i < n; i++) {
    // constant time operation
}
```

The loop runs n times, so the time complexity is O(n).

Nested Loop

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // constant time operation
    }
}
```

- The outer loop runs n times, and for each iteration of the outer loop, the inner loop also runs n times.
- Therefore, the time complexity is $O(n \times n) = O(n^2)$.

Recursive Function

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

The Fibonacci function has two recursive calls for each n, and the depth of recursion is n. Therefore, the time complexity is $O(2^n)$.