Time Complexity

- Time complexity is a measure of the amount of time taken by an algorithm to run as a function of the input size. It helps us understand how the running time of an algorithm increases as the input grows larger.
- Time complexity is typically expressed using big O notation, which provides an upper bound on the growth rate of the algorithm's running time. The notation O(f(n)) represents the upper bound of the growth rate, where f(n) is a function of the input size n.
- Here are some commonly used time complexity classes:

1. **O(1) - Constant Time**: The algorithm takes a constant amount of time regardless of the input size. It is considered the most efficient time complexity.
2. **O(log n) - Logarithmic Time:** The algorithm's running time increases logarithmically with the input size. Algorithms with this complexity often divide the problem into smaller subproblems.
3. **O(n) - Linear Time:** The running time of the algorithm grows linearly with the input size. It implies that the algorithm takes roughly the same amount of time for each element in the input.

Time Complexity

4. O(n log n) - Linearithmic Time: The algorithm's running time grows in a combination of linear and logarithmic terms. Many efficient sorting and searching algorithms, such as merge sort and quicksort, fall into this category.

5**. O(n^2) - Quadratic Time**: The running time of the algorithm grows quadratically with the input size. It usually arises in algorithms that involve nested loops.

6. O(2^n) - Exponential Time: The running time of the algorithm doubles with each additional input element. It is highly inefficient and often impractical for large input sizes.

7. O(n!) - Factorial Time: The running time of the algorithm grows factorially with the input size. It represents the slowest time complexity and is typically associated with brute-force algorithms that examine all possible combinations.

- It's important to note that time complexity analysis focuses on the growth rate of the algorithm's running time rather than the actual running time.
- It helps in comparing algorithms and understanding how they will perform as the input size increases.

Space Complexity

- Space complexity is a measure of the amount of memory or space required by an algorithm to solve a problem as a function of the input size. It helps us understand how the memory usage of an algorithm increases as the input grows larger.

- Similar to time complexity, space complexity is typically expressed using big O notation to provide an upper bound on the growth rate of the algorithm's space requirements. The notation O(f(n)) represents the upper bound of the growth rate, where f(n) is a function of the input size n.
- Here are some commonly used space complexity classes:

1. **O(1) - Constant Space:** The algorithm uses a fixed and constant amount of memory, regardless of the input size. It means that the algorithm has a constant space requirement.
2. **O(n) - Linear Space:** The algorithm's space usage grows linearly with the input size. It indicates that the algorithm stores information in a data structure or an array that scales linearly with the input.
3. **O(n^2) - Quadratic Space:** The algorithm's space usage grows quadratically with the input size. It often arises when an algorithm uses a nested data structure or matrix that requires space for each element or combination of elements in the input.

Space Complexity

4. O(log n) - Logarithmic Space: The algorithm's space usage grows logarithmically with the input size. It typically occurs in algorithms that divide the problem into smaller subproblems and require space for recursive calls or iterative operations.

5. O(n!) - Factorial Space: The algorithm's space usage grows factorially with the input size. It represents the highest space complexity and often occurs in brute-force algorithms that store all possible combinations or permutations.

- It's worth noting that space complexity does not include the input itself, but rather the additional space used by the algorithm during its execution. It considers variables, data structures, recursion stacks, and any other memory required by the algorithm.

- Analyzing the space complexity of an algorithm helps understand its memory requirements and can be crucial when dealing with limited memory environments or optimizing space usage.

1. What is the time, and space complexity of the following code:

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

Options:

1. O(N * M) time, O(1) space
2. O(N + M) time, O(N + M) space
3. O(N + M) time, O(1) space
4. O(N * M) time, O(N + M) space

Ans - 3. O(N + M) time, O(1) space

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

Explanation: The first loop is O(N) and the second loop is O(M).
Since **N** and **M** are **independent variables**, so we can't say which one is the leading term. Therefore **Time complexity** of the given problem will be **O(N+M).**

Since variables size does not depend on the size of the input, therefore **Space Complexity** will be **constant or O(1)**

2. What is the time, and space complexity of the following code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

Options:

1.O(n)
2.O(N log N)
3.O(n^2)
4.O(n^2Logn)

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

Explanation: If you notice, j keeps doubling till it is less than or equal to n. Several times, we can double a number till it is less than n would be log(n). Let's take the examples here.

for n = 16, j = 2, 4, 8, 16

for n = 32, j = 2, 4, 8, 16, 32

So, j would run for O(log n) steps.

i runs for n/2 steps.

So, total steps = O(n/ 2 * log (n)) = **O(n*logn)**