TEST TIME ON TIME AND SPACE COMPLEXITY

1.The space complexity of a recursive function is proportional to:

- The number of iterations
- **The number of function calls**
- The number of conditional statements
- The number of variables used

Ans:
The space complexity of a recursive function is proportional to:

The number of function calls.

Here's why:

- Each recursive call creates a new **stack frame** on the call stack. This frame stores information about the local variables, arguments, and return address of the current function call.
- As the function keeps calling itself, more and more stack frames are created, leading to increased space usage.
- The maximum depth of the recursion tree determines the maximum number of frames on the stack at any given time. This depth reflects the maximum number of function calls.
- The space used by each frame itself might be constant or vary depending on the size of local variables and arguments, but the overall complexity is still dominated by the number of frames.

Other options:

- **Number of iterations:** This might be relevant for iterative implementations, but not directly applicable to recursion.
- **Number of conditional statements:** Conditional statements themselves don't use significant space, only the data they operate on might affect space usage.
- **Number of variables used:** The number of variables in a single frame might contribute to the space consumed by each frame, but ultimately, the number of frames still dominates the overall complexity.

Therefore, focusing on the number of function calls allows us to accurately gauge the space complexity of a recursive function.

2. To verify whether a function grows faster or slower than the other function, we have some asymptotic or mathematical notations, which is_________.

- Big Omega **Ω** (f)
- Big Theta θ (f)
- Big Oh O (f)
- **All of the above**

Ans:
All of the above are correct!

we can use all three asymptotic notations (Big Oh O(f), Big Omega Ω(f), and Big Theta Θ(f)) to verify whether one function grows faster or slower than another. Let's explore how each can be used:

Big Oh O(f): This notation tells you the upper bound of a function's growth rate. If f(n) = O(g(n)), it means that f(n) grows at most as fast as g(n) asymptotically. So, if you compare two functions like f(n) = n^2 and g(n) = n^3, then f(n) = O(g(n)) because n^2 grows slower than n^3 as n approaches infinity.

Big Omega Ω(f): This notation indicates the lower bound of a function's growth rate. If f(n) = Ω(g(n)), it means that f(n) grows at least as fast as g(n) asymptotically. In the previous example, f(n) cannot be Ω(g(n)) because it never grows as fast as n^3.

Big Theta Θ(f): This notation represents the tightest bound, encompassing both the upper and lower bounds. If f(n) = Θ(g(n)), it means that f(n) grows at the same rate as g(n) asymptotically.

Therefore, depending on the specific comparison you want to make, all three notations can be helpful in determining which function grows faster or slower.

3. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

- X will always be a better choice for small inputs
- Y will always be a better choice for small inputs
- **X will always be a better choice for large inputs**
- X will always be a better choice for all inputs

Ans: **X will always be a better choice for large inputs**

Asymptotic efficiency describes the growth rate of an algorithm's performance as the input size increases. If X is asymptotically more efficient than Y, it implies that, for sufficiently large inputs, the performance of X will be better than that of Y. However, it doesn't necessarily mean that X is always a better choice for all inputs, especially for small inputs where constant factors and lower-order terms can have a significant impact.

4.Which of the following is an example of an O(1) space complexity algorithm?

- Quicksort
- Merge sort
- Bubble sort
- **Counting sort**

Ans: **Counting sort**

Counting sort is a non-comparison-based sorting algorithm that achieves O(1) space complexity because it only uses a constant amount of additional space, regardless of the input size. This is in contrast to algorithms like Quicksort, Merge sort, and Bubble sort, which typically require O(log n), O(n), and O(1) space complexities, respectively.

5. To measure Time complexity of an algorithm Big O notation is used which

- describes limiting behavior of the function
- characterizes a function based on growth of function
- upper bound on growth rate of the function
- **all of the mentioned**

Ans: All of the mentioned statements contribute to understanding the role of Big O notation in measuring the time complexity of an algorithm:

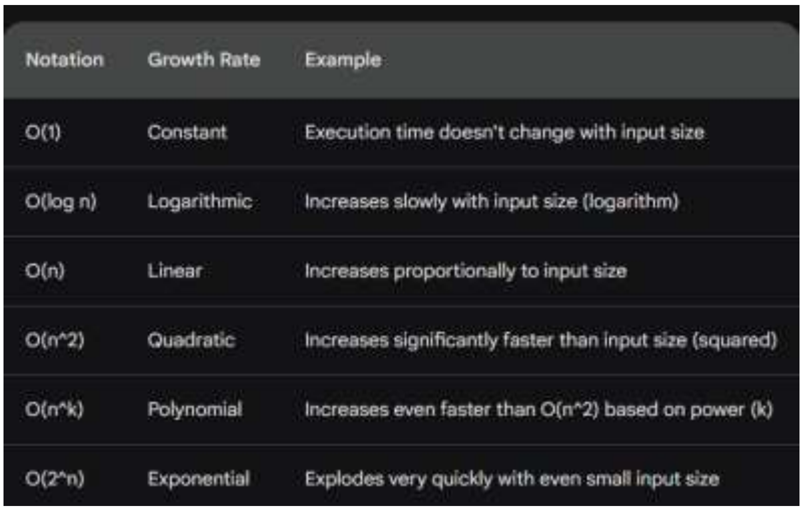
- **Describes limiting behaviour of the function:** Big O notation focuses on how the time taken by an algorithm grows as the input size approaches infinity. It ignores constant factors and lower-order terms, providing a general idea of the algorithm's efficiency for large inputs.
- **Characterizes a function based on growth of function:** By classifying algorithms into categories like O(n), O(log n), O(n^2), etc., Big O notation gives us a high-level understanding of how the time complexity scales with input size. This aids in comparing different algorithms and making informed choices.
- **Upper bound on growth rate of the function:** While Big O notation doesn't give the exact time taken, it gives a guaranteed upper bound. This means the actual time taken can be less than or equal to the specified Big O complexity. This upper bound helps us understand the worst-case scenario for an algorithm's performance.

6.Which of the following is the most efficient algorithm in terms of time complexity?

- **O(n)**
- O(n^2)
- O(n log n)
- O(2^n)

Ans: **O(n)**

- **O(n) - linear time:** While this has good performance for small datasets, its growth rate with increasing input size (n) is directly proportional, leading to slower execution for large datasets.
- **O(n^2) - quadratic time:** This complexity indicates even faster growth than linear. As n increases, the execution time increases quadratically, becoming very slow for larger inputs.
- **O(2^n) - exponential time:** This complexity signifies the worst-case scenario, where the execution time explodes exponentially with increasing input size. This makes it impractical for even moderately large datasets.
- **O(n log n) - logarithmic-linear time:** This complexity represents a sweet spot between linear and logarithmic growth. The "log n" factor keeps the growth slower than linear, resulting in significantly better performance for large datasets compared to the other options



Notation	Growth Rate	Example
O(1)	Constant	Execution time doesn't change with input size
O(log n)	Logarithmic	Increases slowly with input size (logarithm)
O(n)	Linear	Increases proportionally to input size
O(n^2)	Quadratic	Increases significantly faster than input size (squared)
O(n^k)	Polynomial	Increases even faster than O(n^2) based on power (k)
O(2^n)	Exponential	Explodes very quickly with even small input size

7.What is the time complexity of an algorithm that performs n/2 operations in the worst-case scenario?

- **O(n)**
- O(log n)
- O(n log n)
- O(n^2)

Ans:
The time complexity of an algorithm that performs n/2 operations in the worst-case scenario is **O(n)**.

While it might seem intuitive to assume a complexity directly related to the number of operations (n/2), Big O notation focuses on the **asymptotic growth rate** as the input size (n) approaches infinity.

Here's the reasoning:

- Even though the number of operations is directly proportional to n (n/2), the constant factor of 1/2 doesn't affect the dominant growth rate when n becomes very large.
- As n increases, the number of operations (n/2) will also increase, but **at the same rate as n**. For example, doubling the input size (from n to 2n) will double the number of operations (from n/2 to 2n/2).
- This linear growth is captured by O(n) notation, indicating that the execution time increases proportionally to the input size in the worst case scenario

8. O(log n) is?

- constant asymptotic notations
- **logarithmic asymptotic notations**
- polynomial asymptotic notations
- quadratic asymptotic notations

Ans:
O(log n) is indeed a **logarithmic asymptotic notation**. Let's explore why:

Big O notation is used to categorize the **growth rate** of a function's execution time as the input size (n) grows towards infinity. It ignores constant factors and lower-order terms, focusing on the dominant trend.

Logarithmic functions like log(n) grow much slower than other common functions like n (linear), n^2 (quadratic), or even n^(1/2) (power of n). As n increases, the value of log(n) increases, but at a much slower pace.

Therefore, algorithms with an **O(log n) complexity** have their execution time **increase logarithmically** with the input size. This means that doubling the input size will only increase the execution time by a small, constant factor. For instance, searching a sorted array using binary search has O(log n) complexity, meaning it becomes incredibly efficient for large datasets compared to alternatives with higher complexities.

9.What is the time complexity of the following code:

```
int a = 0; for (i = 0; i < N; i++) {

for (j = N; j > i; j--) { a = a + i + j;

}

}
```

- O(N)
- O(N*log(N))
- O(N * Sqrt(N))
- **O(N*N)**

Ans:

O(N^2)

The code contains nested loops where the outer loop runs N times, and for each iteration of the outer loop, the inner loop runs from N to i times. The total number of iterations is the sum of the first N natural numbers, which is proportional to N^2. Therefore, the time complexity of the code is **O(N^2).**

10.What is the space complexity of an algorithm that uses an array of size n to store the input, and another array of size m to store the output?

- O(1)
- O(n)
- O(m)
- **O(n + m)**

Ans: **O(n + m)**

The space complexity is determined by the sum of the space required for the input array (O(n)) and the space required for the output array (O(m)). Therefore, the overall space complexity is O(n + m).