## **Step 1: Understand Asymptotic Notation**

1. Explain Big O notation and how it helps in analyzing algorithms.

**Big O notation** is a mathematical way to describe how an algorithm performs as the size of the input data (**n**) increases. It doesn’t measure the actual execution time, but rather **how the time or space grows** in relation to the input size. It expresses the **worst-case upper bound** of an algorithm's runtime or memory usage, allowing developers to compare different approaches **regardless of hardware or coding language**. For example:

* A **linear search** has time complexity **O(n)** — meaning the time increases linearly with the number of items.
* A **binary search** has time complexity **O(log n)** — making it much faster for large, sorted data sets.

Understanding Big O helps in:

* Choosing the right algorithms for **speed and efficiency**.
* **Predicting performance** bottlenecks before they occur.
* Designing systems that can **scale** with growing data.

1. Describe the best, average, and worst-case scenarios for search operations.

**Best Case:**  
This is the most favorable scenario—when the search target is found in the **first attempt**.  
Example: Searching for a product that appears at the **first position**.  
Time: As low as **O(1)** in linear search.

**Average Case:**  
This represents a **typical or expected case**, based on random input distribution.  
Example: Finding a product somewhere in the **middle** of the list.  
Time: About **O(n/2)** ≈ **O(n)** for linear search.

**Worst Case:**  
This is the **least favorable scenario**—when the search element is **not present**, or appears at the **last position**.  
Example: Looking for a non-existent product in an unsorted list.  
Time: Up to **O(n)** in linear search and **O(log n)** in binary search.

**Step 4: Analysis**

1. Compare the time complexity of linear and binary search algorithms.

When implementing search functionality in an e-commerce platform, it's important to analyze the **efficiency of each algorithm** based on the nature of the dataset.

| **Search Type** | **Time Complexity** | **Best Use Case** |
| --- | --- | --- |
| **Linear Search** | O(n) | Unsorted product lists or small datasets |
| **Binary Search** | O(log n) | Large datasets that are **pre-sorted** by product name, ID, or price |

#### **Linear Search (O(n))**

* Scans each item in the list **sequentially** until the desired item is found or the list ends.
* In the **worst case**, it checks every item — so time grows **linearly** with the size of the data.
* While it's simple and works for **unsorted arrays**, it's **not scalable** for large inventory systems.

#### **Binary Search (O(log n))**

* Works only on **sorted arrays**.
* Divides the search space in half with each step, leading to **logarithmic** performance.
* For example, if there are 1000 products, binary search finds an item in at most **10 steps**.

Linear search is flexible but **inefficient** on large data.

Binary search is **much faster** on sorted data and is more scalable.

1. Discuss which algorithm is more suitable for your platform and why.

In an **e-commerce platform**, users frequently search for products by name, ID, or category. The product database is typically **large and sorted** to support fast browsing, filtering, and recommendations.

I will **prefer Binary Search** when:

* Products are stored in a **sorted array or list**.
* The search must be **real-time and scalable**, such as during flash sales or festive seasons when traffic spikes.
* The platform supports millions of products and must return results in **milliseconds**.

I can use **Linear Search** when:

* The dataset is **very small** (e.g., admin tools or testing environments).
* The products are **unsorted** and don’t justify the overhead of sorting.