Asymptotic Notation & Search Scenarios:

**O(1) – Constant Time**

In constant time complexity, the operation takes the same amount of time regardless of the size of the input data. For search algorithms, this occurs when the target element is found immediately — typically at the **first inspected position**. Although rare in general search algorithms, this can happen in ideal best-case scenarios (e.g., first item match in a linear search or accessing a value in a hash table).

* **Best Case:** Target is found at the first position.
* **Average/Worst Case:** Not applicable (—) since O(1) only describes fixed-time operations.

**O(log n) – Logarithmic Time**

Logarithmic time complexity means that the algorithm reduces the problem size by a constant factor (usually by half) with each step. This behavior is seen in **binary search**, where the search space is halved after every comparison.

* **Best Case:** The target is found at the middle element during the first comparison.
* **Average Case:** On average, about **log₂ n** comparisons are needed.
* **Worst Case:** In the worst scenario (e.g., target not present or found at one of the ends), **log₂ n** comparisons are still required.

**O(n) – Linear Time**

Linear time complexity means the time required grows linearly with the number of elements. In a **linear search**, each item is checked one-by-one until the target is found or the end is reached.

* **Best Case:** The target is found at the first element.
* **Average Case:** The target is somewhere in the middle — around **n / 2** comparisons.
* **Worst Case:** The target is at the last position or not present at all, requiring **n** comparisons.

Which Algorithm Should an E‑commerce Platform Use?:

**Complexity.**  
Linear search runs in O(n) time because it may need to examine every element one‑by‑one in the worst case. Binary search, on the other hand, operates in O(log n) time; by repeatedly halving the remaining search space, it reaches the target in far fewer comparisons as the data set grows.

**Data requirement.**  
A linear scan works on any collection regardless of order, so you can search an unsorted list immediately. Binary search demands that the data be sorted (or that an ordered index exist) before it can perform its divide‑and‑conquer steps; without that ordering guarantee, its logic breaks down.

**Insert/Delete cost.**  
Appending to—or deleting from—an unsorted list is cheap with linear search: simply add or remove the element, typically in O(1) time for an append. For binary search, each insertion or deletion must preserve order (or update the index), introducing extra overhead that can be O(n) for array‑based structures or logarithmic for balanced trees.

**Memory footprint.**  
A plain linear search needs only the original array, so its memory usage is minimal. Binary search often relies on auxiliary structures—a sorted copy, a B‑tree, or another type of index—to keep data ordered and thus may consume additional memory.

**When each is ideal.**  
Linear search excels on very small collections or on lists that change so frequently that maintaining sort order would be more expensive than the occasional O(n) scan (e.g., a user’s five recently viewed items). Binary search shines on large, mostly read‑only datasets such as an e‑commerce product catalog, or whenever a database or search engine already maintains an ordered index, because its logarithmic lookup time scales far better as the data grows.