**WEEK1: E-commerce Platform Search Function**

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

Ans: Big O notation is a formal way to express how the execution time or memory usage of an algorithm grows relative to the size of the input. Rather than giving precise measurements, it focuses on the rate of growth, which helps in understanding how an algorithm will perform as the input becomes large. It is primarily used to analyze the worst-case behavior, allowing programmers to choose the most efficient algorithm when scaling up. Big O abstracts away machine-specific factors and constant multipliers, providing a clear, hardware-independent comparison of algorithms.

Below are some commonly used Big O notations:

* **O(1): Constant Time**  
  The runtime or space requirement remains the same regardless of input size. For example, accessing an array element by index is O(1) because it takes the same time no matter how large the array is.
* **O(log n): Logarithmic Time**  
  The runtime increases slowly even as the input size increases significantly. Binary search is a classic example, where the dataset is halved at each step.
* **O(n): Linear Time**  
  The runtime increases proportionally with input size. For example, a linear search that checks each element one by one has O(n) time complexity.
* **O(n log n): Linearithmic Time**  
  This is common in efficient sorting algorithms like merge sort and heap sort. The algorithm performs log n operations for each of the n elements.
* **O(n²): Quadratic Time**  
  The runtime grows with the square of the input size. This is often seen in algorithms with nested loops, such as bubble sort or selection sort.
* **O(2ⁿ): Exponential Time**  
  The runtime doubles with each additional input element. Algorithms that explore all combinations, like brute-force solutions for the traveling salesman problem, fall into this category.
* **O(n!): Factorial Time**  
  The most inefficient class of algorithms, where time complexity grows factorially with input size. This is seen in exhaustive search algorithms using permutations.

Big O notation helps developers evaluate and compare algorithms based on how well they scale. It provides a language to discuss performance without relying on specific hardware or actual runtime. By understanding the time and space complexity, we can select the most efficient algorithm for a particular use case—especially important in large-scale systems like e-commerce platforms, search engines, and databases.

2] Best, Average, and Worst-Case Scenarios for Search Operations

Ans: **Linear Search:**

Best Case :O(1)The element is found at the very beginning of the list, so only one comparison is required.

Average Case :O(n)The element is somewhere in the middle of the list, so on average, half of the elements are checked before finding the target.

Worst Case :O(n)The element is at the end of the list or not present at all, so every element in the list must be checked.

**Binary Search:**

Best Case :O(1)The element is found on the first check, which happens when the middle element of the list is the target.

Average Case :O(log n)The element is found after a few divisions of the list. Each division cuts the search space in half, so the number of comparisons grows logarithmically.

Worst Case :O(log n)The element is not present or is found only after the maximum number of divisions. Even in this case, the number of steps grows logarithmically with the input size.

3] Compare the time complexity of linear and binary search algorithms.

Ans:

**Linear Search** and **Binary Search** are two fundamental searching algorithms, but they differ significantly in terms of **time complexity and performance**.

* **Linear Search** has a time complexity of:

**Best Case:** O(1) : when the target is the first element.

**Average Case:** O(n) :the target is somewhere in the middle.

**Worst Case:** O(n) :the target is the last element or not present at all.

It checks each element one by one, making it inefficient for large datasets.

* **Binary Search**, on the other hand, has a time complexity of:

**Best Case:** O(1) : when the target is the middle element.

**Average/Worst Case:** O(log n) :the list is repeatedly divided in half.

It only works on **sorted data**, but is much faster due to its divide-and-conquer approach.

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

Ans:For an e-commerce platform, where users may search among thousands or millions of products, binary search is more suitable. This is because:

* Speed Matters: Binary search provides **l**ogarithmic performance, which is far better for large data.
* Data Can Be Pre-sorted or Indexed: In real-world systems, product catalogs are usually sorted or indexed by product ID or name, making them ideal for binary search.
* User Experience**:** Faster searches mean quicker responses, leading to a smoother and more satisfying user experience.

Therefore, binary search is the better choice for a scalable and responsive search system in an e-commerce platform. However, if the data is unsorted and performance is not a major concern, linear search could be used for its simplicity.