1. Understand Asymptotic Notation:

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

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm’s time or space complexity.

* Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* It provides an upper limit on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O
* It’s denoted as O(f(n)), where f(n) is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size n.

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

1. Worst Case Analysis (Mostly used)

* In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
* For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one.
* This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

2. Best Case Analysis (Very Rarely used)

* In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
* For linear search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

3. Average Case Analysis (Rarely used)

* In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
* We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](http://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). We take (n+1) to consider the case when the element is not present.

1. Analysis:

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

| Feature | Linear Search | Binary Search |
| --- | --- | --- |
| Best Case | O(1) (first element match) | O(1) (middle element match) |
| Average Case | O(n) | O(log n) |
| Worst Case | O(n) | O(log n) |
| Input Requirement | No sorting required | Must be sorted |
| Implementation | Simple | Slightly more complex |

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

According to me ,Binary Search is more suitable, because:

| Reason | Explanation |
| --- | --- |
| Speed | E-commerce platforms have thousands to millions of products. Binary search handles large input sizes efficiently with O(log n) performance. |
| Scalability | Works well as the number of products grows—essential for high-performance search systems. |
| Sorted Data | Most platforms store products in sorted or indexed formats (e.g., in databases) so sorting is already handled. |
| Frequent Searching | Users constantly search for products, so optimized search speed is essential for user satisfaction and performance. |