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

Ans. Big O notation is a mathematical notation used in computer science to describe the upper bound or worst-case scenario of the runtime complexity of an algorithm in terms of the input size. It provides a standardized and concise way to express how the performance of an algorithm scales as the size of the input grows.

Understanding Big O Notation: In Big O notation, "O" represents the order of the function, and "f(n)" represents the function describing the algorithm's time complexity in terms of the input size "n." The notation "O(f(n))" signifies that the algorithm's time complexity grows no faster than a specific function of "n." Here, "f(n)" is a mathematical function describing how the algorithm's runtime increases as the input size grows.

For example: O(log n): Logarithmic time complexity, where the algorithm's runtime grows logarithmically with the input size.

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

Ans. Linear Search

Best Case: O(1) - The element is the first one.

Average Case: O(n) - The element is somewhere in the middle.

Worst Case: O(n) - The element is the last one or not present.

Binary Search (requires sorted array)

Best Case: O(1) - The element is the middle one.

Average Case: O(log n) - The element is somewhere in the array, requiring repeated halving.

Worst Case: O(log n) - The element is not present, requiring repeated halving until the array is exhausted.

Interpolation Search (Sorted and Uniformly Distributed List)

Best Case: O(1) - The target element is found in the first probe.

Average Case: O(log log n) - The target element is found after a few probes due to uniform distribution.

Worst Case: O(n) - The distribution is non-uniform, making the search degrade to linear search.

The most efficient way to search an element in a sorted array is binary search and in an unsorted array is linear search.

3)Compare the time complexity of linear and binary search algorithms?

Ans.

**Linear Search:** Linear search scans through each element in a list one by one until it finds the target value or reaches the end of the list.

Time Complexity:

Best Case: 𝑂(1) — This occurs if the target value is the first element in the list.

Average Case: 𝑂(𝑛) — On average, the search will need to examine half of the elements in the list.

Worst Case: 𝑂(𝑛) — This occurs if the target value is at the end of the list or not present at all, requiring the search to check every element.

**Binary Search:** Binary search works on a sorted list by repeatedly dividing the search interval in half. It compares the target value with the middle element of the interval and eliminates half of the search space based on the comparison.

Time Complexity:

Best Case: 𝑂(1) — This occurs if the target value is the middle element of the list.

Average Case: 𝑂(log𝑛) — On average, binary search will require log𝑛 comparisons to find the target value or determine that it is not in the list.

Worst Case: 𝑂(log𝑛) — This occurs when the target value is either present in the list or not, requiring log𝑛 comparisons.

Explanation: Binary search has a time complexity of 𝑂(log𝑛) because it repeatedly halves the search space, making the number of comparisons proportional to the logarithm of the number of elements in the list.

4)Discuss which algorithm is more suitable for your platform and why?

Ans. In case of our program we are inserting all the data manually, so they are unsorted by default. If we sort it the minimum time complexity would be O(nlogn) and then sorting it using binary search would be O(logn) which is not feasible. In such a case when we have unsorted array by default, it is better to use linear search. So in this problem linear search is preferrable as it takes less time.

In problems where the list would be sorted initially, binary search is preferrable.