# DS UNIT 1

[ 4-marks ]

**1. Define Data Structure. Explain its importance.**

- A **data structure** is a way of organizing and storing data in memory so that it can be used efficiently.
- It defines how data elements are arranged, accessed, and modified.
- Proper data structures make programs **faster and easier to manage**.
- They help in efficient **data access, insertion, deletion, and searching**.
- Choosing the right data structure reduces **time complexity** and **memory usage**.
- Data structures are important for handling **large amounts of data**, such as in databases, operating systems, and real-time applications.

**2. List and explain basic operations on data structures.**

Basic operations performed on data structures are:

- **Traversal:** Visiting each element of the data structure once.
- **Insertion:** Adding a new element at a specific position.
- **Deletion:** Removing an existing element from the structure.
- **Searching:** Finding the location of a particular element.
- **Sorting:** Arranging elements in ascending or descending order.
- **Merging:** Combining two data structures into one.

These operations help in effective manipulation and management of stored data.

**3. What is an array? Mention its advantages and disadvantages.**

- An **array** is a linear data structure that stores elements of the **same data type** in contiguous memory locations.
- Each element is accessed using an **index value**.

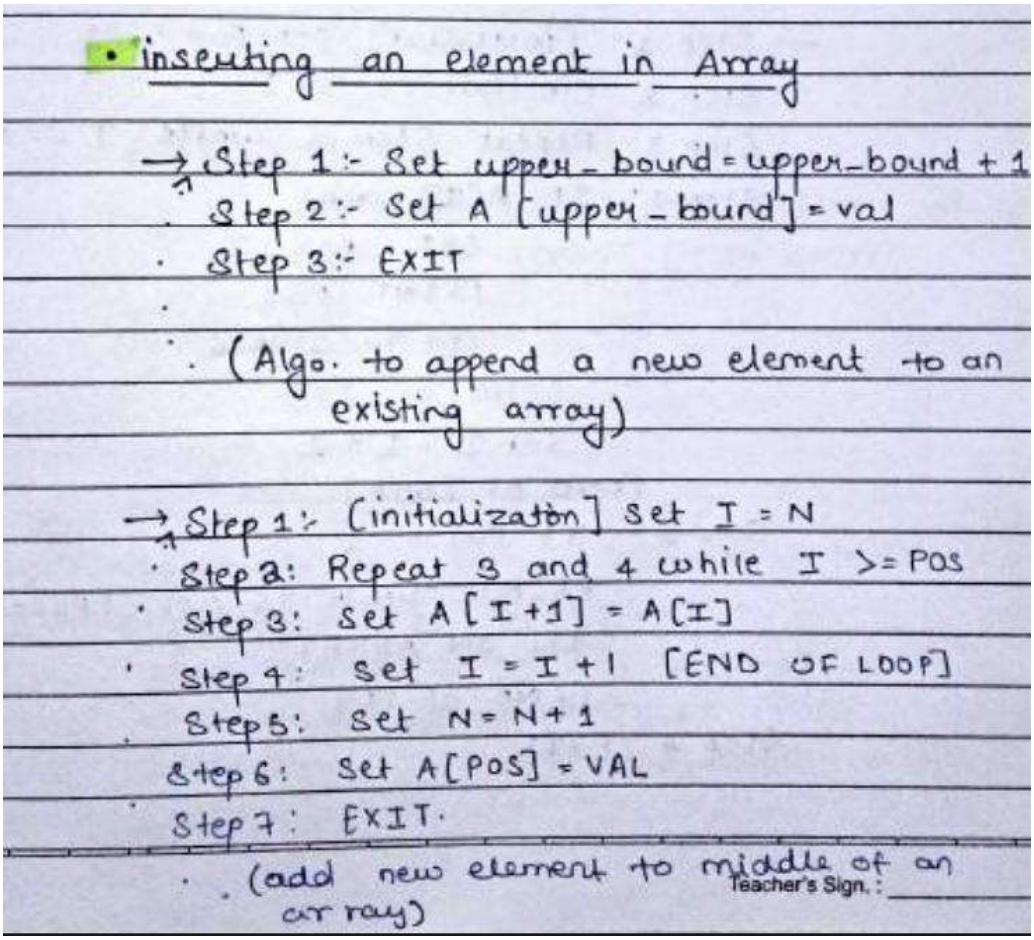
**Advantages:**

- Fast access using index
- Easy to traverse
- Efficient for storing fixed-size data

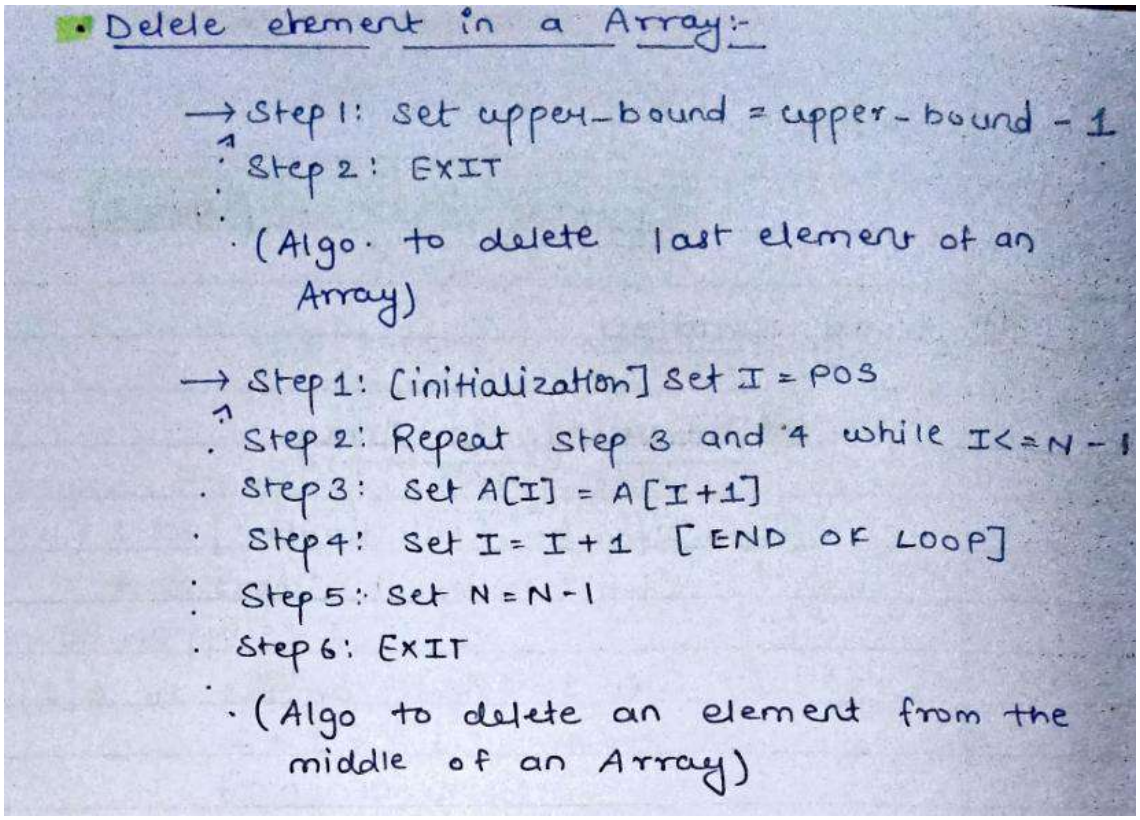
**Disadvantages:**

- Fixed size (cannot grow dynamically)
- Insertion and deletion are time-consuming
- Wastage of memory if size is not fully used

**4. Write an algorithm for insertionn of an element in an array.**



**5. Write an algorithm for deletion of an element from an array.**




**6. Define Time Complexity and Space Complexity**

- Time Complexity refers to the amount of time an algorithm takes to run as the size of input data increases. It helps in analyzing how fast or slow an algorithm performs for large inputs.
- Time complexity is expressed using mathematical notations such as Big-O, Big-Ω, and Big-Θ. These notations focus on the growth rate rather than exact execution time, which may vary across systems.
- Space Complexity refers to the total memory required by an algorithm during its execution. It includes memory for input data and additional temporary variables.
- Space complexity is important in systems where memory is limited or data size is very large.

**7. Explain Big-O Notation with one example**

- Big-O notation is used to describe the worst-case performance of an algorithm. It shows how execution time increases as the input size grows.
- Big-O ignores constant values and lower-order terms. This helps in comparing algorithms based on their efficiency rather than machine-dependent factors.
- It is mainly used to predict performance for large input sizes. This helps developers select efficient algorithms.
- Example: In linear search, each element is checked one by one. For n elements, the maximum number of comparisons is n, so the time complexity is O(n).

**8. Write a short note on Linear Search**

- Linear search is a basic searching technique where each element is examined sequentially. The search continues until the required element is found or the list ends.
- It does not require the data to be sorted. Because of this, linear search is simple to implement and understand.
- Linear search is suitable for small datasets where performance is not critical. However, it becomes inefficient for large datasets.
- The worst-case time complexity of linear search is O(n). It is commonly used when data is unsorted or frequently updated.

**9. Difference between Linear Search and Binary Search**

- Linear search checks elements one by one starting from the first element. Binary search repeatedly divides the sorted list into two halves to find the element quickly.
- Linear search can be applied to both sorted and unsorted data. Binary search works only on sorted data.
- Linear search is simple and easy to implement. Binary search is faster but requires careful implementation.
- The time complexity of linear search is O(n) in the worst case. Binary search has a time complexity of O(log n), making it efficient for large datasets.

[ 5- marks ]

**1. Traversal, Insertion, and Deletion operations on arrays**

**Traversal :** Traversal is the process of accessing each element of an array one by one. It is mainly used to display, search, or update array elements.

- During traversal, we start from the first index of the array and continue till the last index. The size of the array does not change during traversal.
  **Example:** For the array {10, 20, 30}, traversal prints 10, then 20, and then 30.

**Insertion :** Insertion is the process of adding a new element at a specified position in an array. Since arrays have fixed size, space must be created before insertion.

- To insert an element, existing elements are shifted one position to the right. This shifting ensures that the new element fits at the required position.
  **Example:** Inserting 25 at position 2 in {10, 20, 30} results in {10, 25, 20, 30}.

**Deletion :** Deletion is the process of removing an element from a specified position in an array. After deletion, the empty space must be filled.

- To fill the gap, all elements after the deleted element are shifted one position to the left. This keeps the array continuous in memory.
  **Example:** Deleting the element at position 2 from {10, 20, 30} results in {10, 30}.

**2. Comparison of Bubble Sort, Insertion Sort, and Selection Sort**

**Bubble Sort :** Bubble Sort works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. With each pass, the largest element moves to the end of the array.

- It is very easy to understand and implement, especially for beginners. However, it performs a large number of comparisons.
- Bubble Sort is inefficient for large datasets because it requires many passes. It is mostly used for learning purposes rather than real applications.

**Insertion Sort :** Insertion Sort works by taking one element at a time and inserting it into its correct position in the sorted part of the array. The sorted portion grows gradually.

- It performs well when the data is already partially sorted. This makes it faster than Bubble Sort in many cases.
- Insertion Sort is simple, stable, and efficient for small datasets. It is commonly used in real-world applications for small inputs.

**Selection Sort:** Selection Sort works by selecting the smallest element from the unsorted portion of the array. The selected element is placed at the correct position in each pass.

- It performs fewer swaps compared to Bubble Sort. However, it still requires scanning the entire array repeatedly.
- Selection Sort is easy to implement but slow for large datasets. It is mainly used for educational purposes.

**3. Explain Bubble Sort algorithm with example**

**Bubble Sort** is a simple sorting algorithm that works by repeatedly comparing adjacent elements.
If two adjacent elements are in the wrong order, they are swapped.

- This process continues for multiple passes until the array becomes sorted.
- After each pass, the largest element "bubbles up" to its correct position at the end of the array.
- Bubble sort is easy to understand and implement, but it is slow for large datasets.

**Example:**
Array: {5, 3, 8, 2}

- Pass 1: {3, 5, 2, 8}
- Pass 2: {3, 2, 5, 8}
- Pass 3: {2, 3, 5, 8}
- The final sorted array is {2, 3, 5, 8}.
- Worst-case time complexity of Bubble Sort is **$O(n^2)$**.

**4 Explain Big-O, Big-Ω, and Big-θ notations with suitable examples.**

- **Big-O notation** represents the worst-case time complexity of an algorithm. It shows the maximum time an algorithm may take as input size increases.
  It helps in analyzing performance for very large inputs and ignores constants and lower-order terms.
  Example: In linear search, if the element is at the last position, all n elements are checked, so time complexity is O(n).

- **Big-Ω notation** represents the best-case time complexity of an algorithm. It shows the minimum time required when conditions are ideal.
  It helps understand how fast an algorithm can perform in the best situation.
  Example: In linear search, if the element is found at the first position, time complexity is Ω(1).

- **Big-θ notation** represents the average or exact time complexity. It gives both upper and lower bounds.
  It shows the true growth rate of an algorithm.
  Example: Linear search has θ(n) time complexity on average.

**5 Explain Insertion Sort with algorithm and time complexity.**

- Insertion Sort is a simple sorting technique that works like arranging cards in hand. It builds the sorted array one element at a time.
- The array is divided into a sorted part and an unsorted part. Elements from the unsorted part are placed into the correct position.

Algorithm (Insertion Sort):

- Start from the second element of the array.
- Compare the current element with previous elements.
- Shift all larger elements one position to the right.
- Insert the current element at its correct position.
- Repeat until the array is sorted.
- Example: {8, 3, 5, 2} → {3, 8} → {3, 5, 8} → {2, 3, 5, 8}
- Best case: O(n) when array is already sorted.
- Worst case: $O(n^2)$ when array is in reverse order.

**6 Explain Selection Sort with algorithm and example.**

- Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the smallest element from the unsorted part.
- The selected smallest element is placed at the correct position in the array.

Algorithm (Selection Sort):

- Start from the first index of the array.
- Find the smallest element in the unsorted portion.
- Swap it with the element at the current position.
- Move to the next position and repeat the steps.
- Example:
  {64, 25, 12, 22}
  First pass → {12, 25, 64, 22}
  Second pass → {12, 22, 64, 25}
  Final → {12, 22, 25, 64}
- Selection Sort always performs the same number of comparisons.
- Time complexity: $O(n^2)$ in best, average, and worst cases.

**7 Differentiate between Array and Queue.**

- **Array** is a linear data structure that stores elements in **contiguous memory locations**, meaning all elements are stored next to each other in memory. Each element can be accessed directly using its **index value**, which makes access very fast.
  **Queue** is also a linear data structure, but it follows the **FIFO (First In First Out)** principle, where the element inserted first is removed first.
- In **arrays**, insertion and deletion operations can be performed at **any position**. However, these operations require shifting elements, which makes them time-consuming.
  In **queues**, insertion is allowed only at the **rear** and deletion only at the **front**, which maintains order.
- Arrays support **random access**, meaning any element can be accessed directly in constant time.
  Queues do **not support random access** and elements must be accessed sequentially.
- Arrays are mainly used to store **fixed-size data** such as lists of marks or numbers.
  Queues are widely used in **scheduling, buffering, and real-time systems** like printer queues and CPU scheduling.
- Arrays are easy to implement but inefficient for frequent insertion and deletion.
  Queues are efficient for managing **continuous and sequential data flow**.

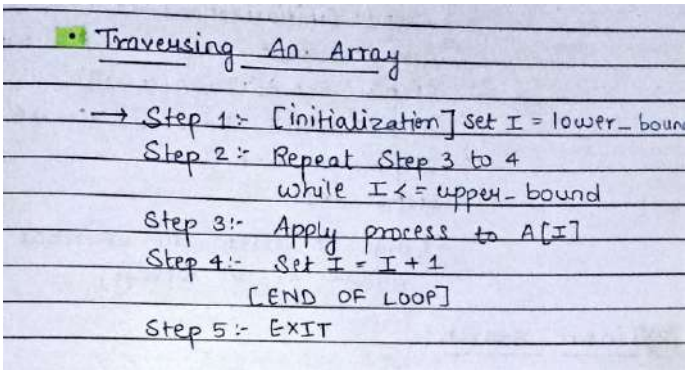
**[ 10 -marks ]**

**1 Explain arrays as a data structure. Describe insertion, deletion, and traversal operations with algorithms.**

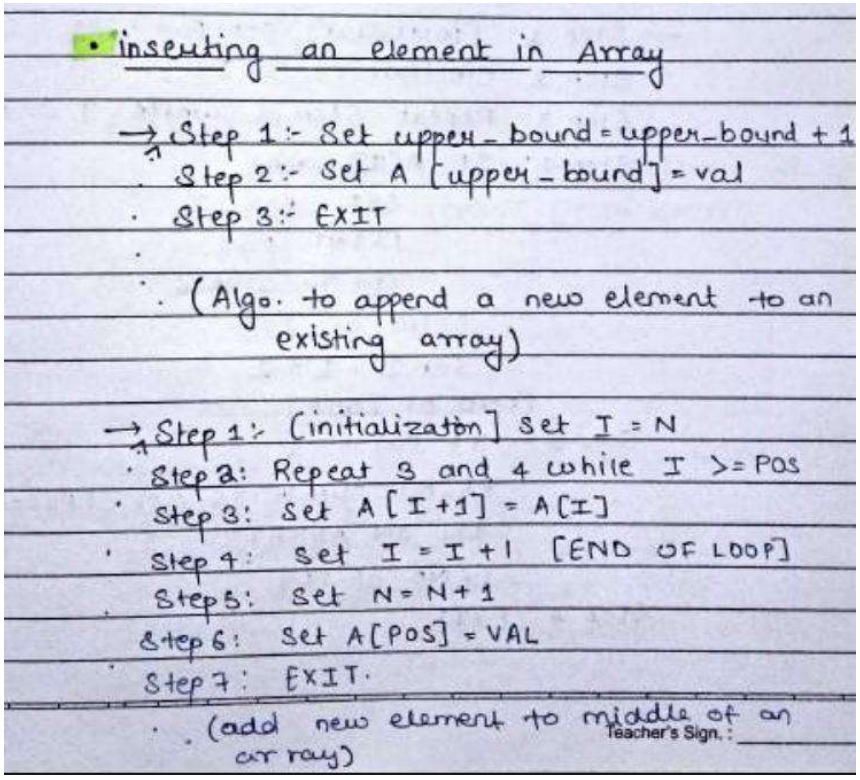
An array is a linear data structure used to store a collection of elements of the same data type in contiguous memory locations. Each element in an array is identified by an index, which helps in fast access. Arrays are widely used because they allow easy storage and retrieval of data using index values.

Arrays are best suited when the number of elements is known in advance. Since memory is allocated continuously, accessing any element takes constant time. However, arrays have a fixed size, which makes insertion and deletion costly operations.

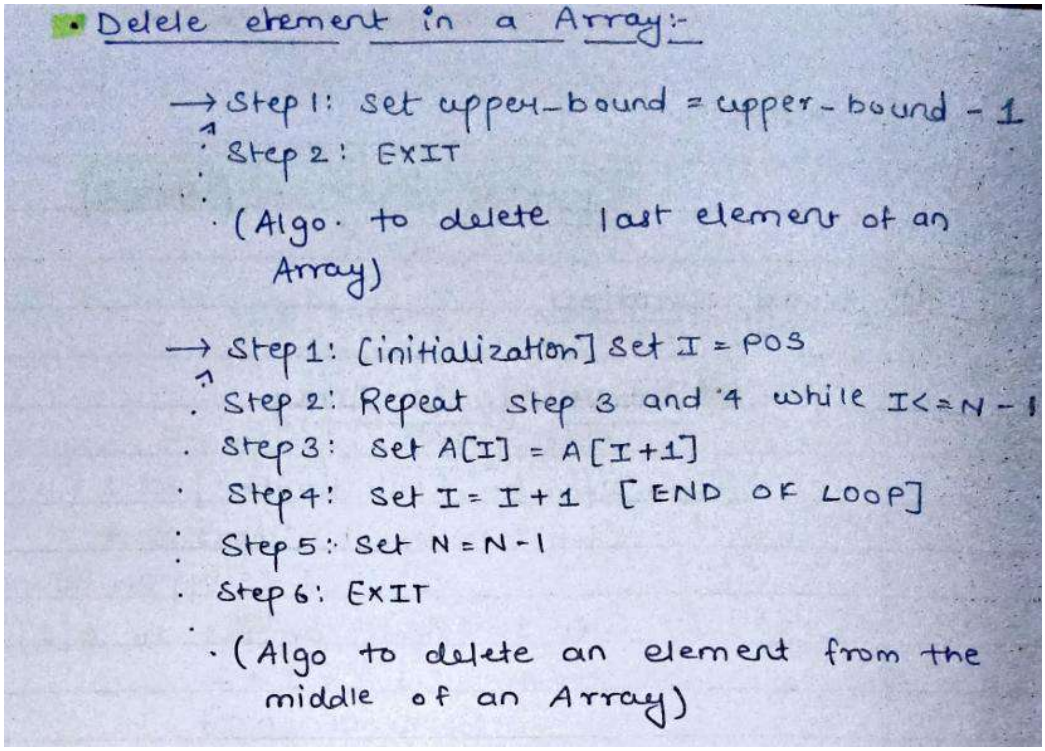
**Traversal Operation :** Traversal means visiting each element of the array one by one.




**Insertion Operation :** Insertion means adding a new element at a specific position.



**Deletion Operation :** Deletion means removing an element from a given position.



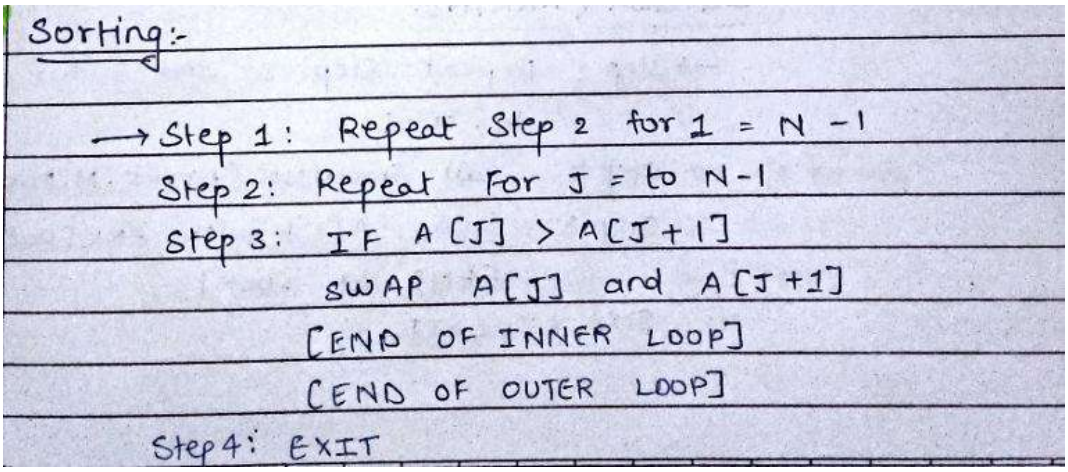

**2 Explain Bubble Sort in detail with algorithm, example, and time complexity analysis.**

**Bubble Sort** is a simple comparison-based sorting algorithm. It works by repeatedly comparing **adjacent elements** and swapping them if they are in the wrong order.

This process is repeated for multiple passes until the array becomes sorted. The largest element moves to the end of the array in each pass, similar to a bubble rising to the surface.

Bubble sort is easy to understand and implement, making it suitable for beginners. However, it is inefficient for large datasets due to its high time complexity.

Algorithm –




Consider the array: [5, 3, 8, 2]

Bubble Sort compares adjacent elements and swaps them if they are in the wrong order.

**Pass 1**

- Compare 5 and 3 → 5 > 3 → Swap [3, 5, 8, 2]
- Compare 5 and 8 → 5 < 8 → No swap [3, 5, 8, 2]
- Compare 8 and 2 → 8 > 2 → Swap [3, 5, 2, 8]

Largest element 8 is now at the end.

**Pass 2**

- Compare 3 and 5 → No swap [3, 5, 2, 8]
- Compare 5 and 2 → 5 > 2 → Swap [3, 2, 5, 8]

Second largest element 5 is placed correctly.

**Pass 3**

- Compare 3 and 2 → 3 > 2 → Swap [2, 3, 5, 8]

**Final Sorted Array -> [2, 3, 5, 8]**

**Time Complexity Analysis:**

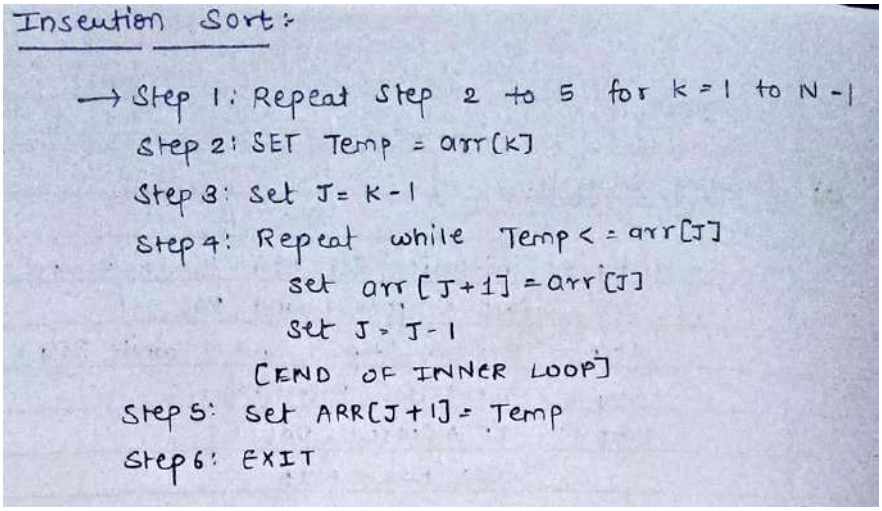
- **Best Case:** O(n) when the array is already sorted
- **Worst Case:** $O(n^2)$ when the array is in reverse order
- **Average Case:** $O(n^2)$

Bubble sort uses constant extra space, so its **space complexity is O(1)**. Although simple, it is not suitable for large datasets but useful for educational purposes.

**3 Explain Insertion Sort and Selection Sort with algorithms and compare their time complexities.**

**1. Insertion Sort**

- Insertion Sort works the way we sort playing cards in our hand.
- It takes one element at a time and inserts it into its correct position in the already sorted part of the array.
- The array is divided into sorted and unsorted parts.
- It is efficient for small datasets or nearly sorted arrays.

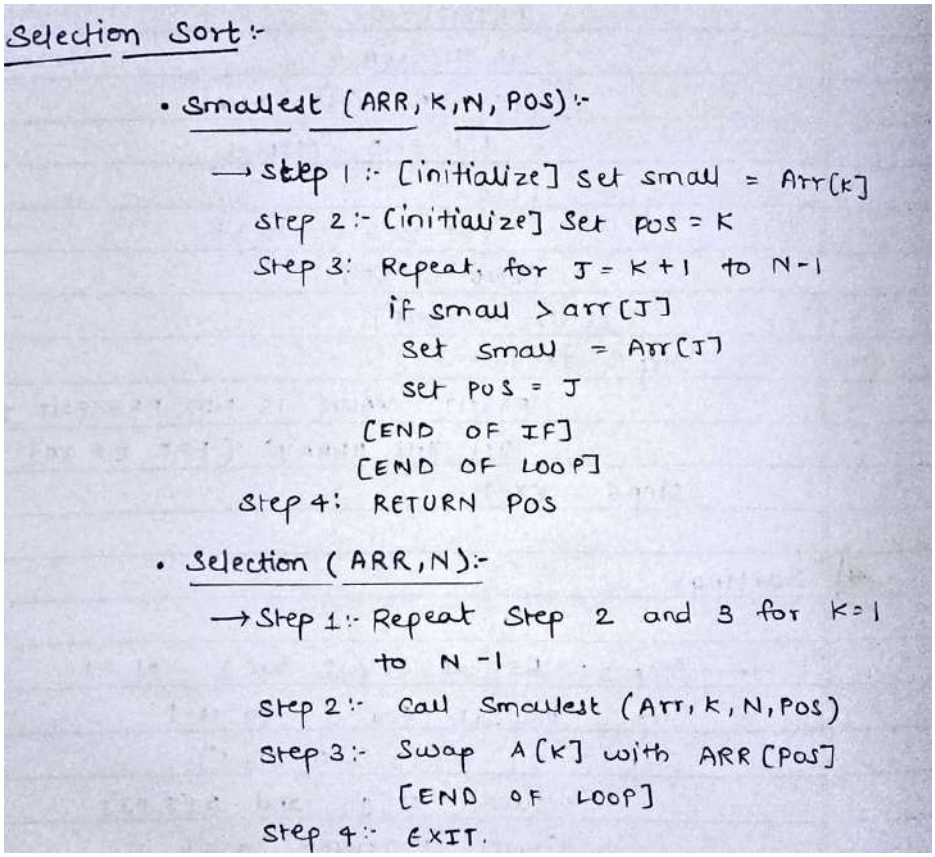



Time Complexity:

- Best case: O(n) (already sorted)
- Average case: $O(n^2)$
- Worst case: $O(n^2)$ (reverse order)

2. Selection Sort

- Selection Sort repeatedly finds the smallest element from the unsorted part of the array.
- The smallest element is swapped with the first element of the unsorted part.
- This process continues until the entire array is sorted.
- It performs fewer swaps but many comparisons.




Algorithm (Selection Sort):

Time Complexity:

- Best case: $O(n^2)$
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

3. Comparison of Time Complexities

- Insertion Sort is faster for small or nearly sorted data. It reduces comparisons when data is already partially sorted.
- Selection Sort always performs the same number of comparisons, regardless of input order.
- Insertion Sort is more adaptive, while Selection Sort is not adaptive.

**4 Discuss algorithm analysis. Explain time complexity, space complexity, Big-O, Big-Ω, and Big-θ notations with examples**

**Algorithm Analysis**

Algorithm analysis is the process of evaluating the efficiency of an algorithm before implementing it. It helps us understand how much time and memory an algorithm will require when the input size increases.

Instead of focusing on machine speed or programming language, algorithm analysis studies the logical performance of an algorithm. This helps programmers choose the most efficient algorithm for a given problem, especially when dealing with large data.

**Time Complexity**

Time complexity refers to the amount of time an algorithm takes to execute as the input size grows. It measures the number of basic operations performed by the algorithm. Time complexity helps compare algorithms based on speed.

Example: In linear search, if there are n elements, the algorithm may check all n elements, so time complexity is O(n).

**Space Complexity**

Space complexity refers to the amount of memory an algorithm needs during execution. It includes memory for input data and extra variables used by the algorithm. Space efficiency is important in systems with limited memory.
Example: Bubble sort uses only a few temporary variables, so its space complexity is O(1).

**Big-O Notation**

Big-O notation represents the worst-case time complexity of an algorithm. It shows the maximum time required for large inputs.
Example: Binary search has a worst-case time complexity of O(log n).

**Big-Ω (Omega) Notation**

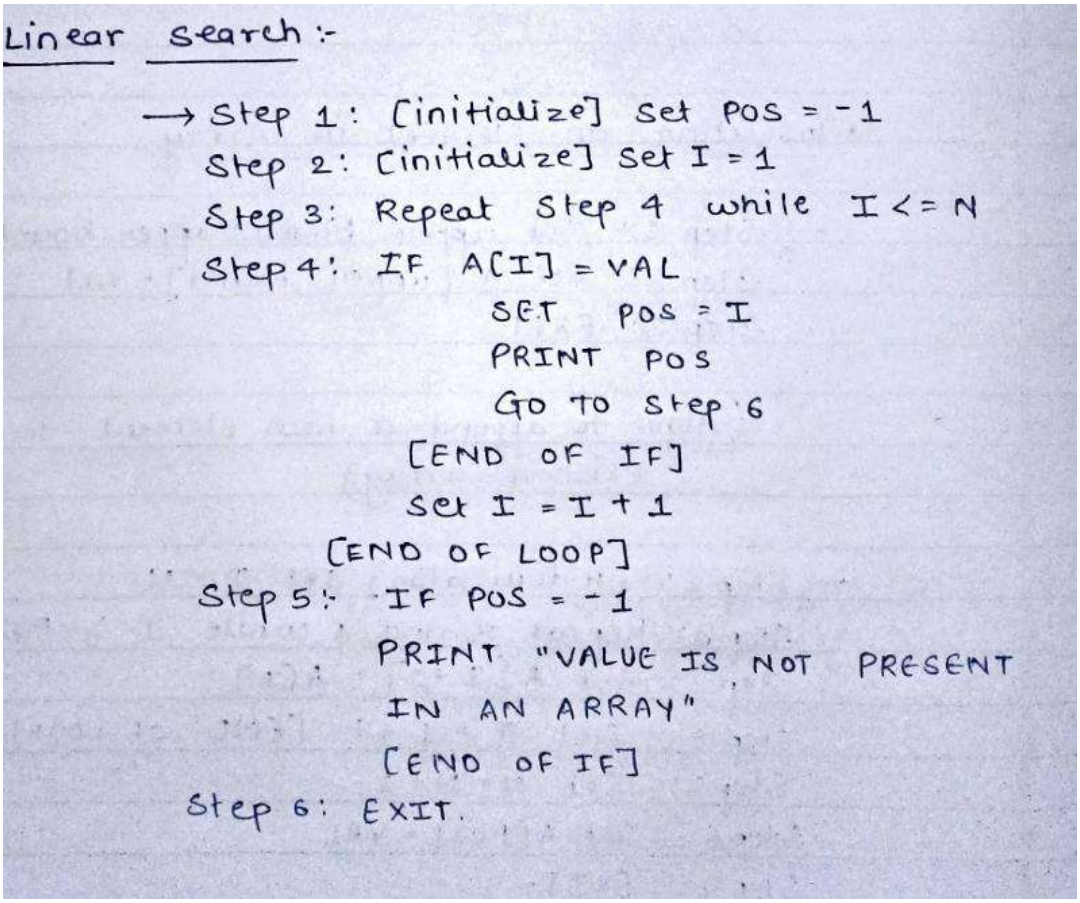
Big-Ω notation represents the best-case time complexity. It shows the minimum time required under ideal conditions.
Example: Linear search has Ω(1) when the element is found at the first position.

**Big-θ (Theta) Notation**

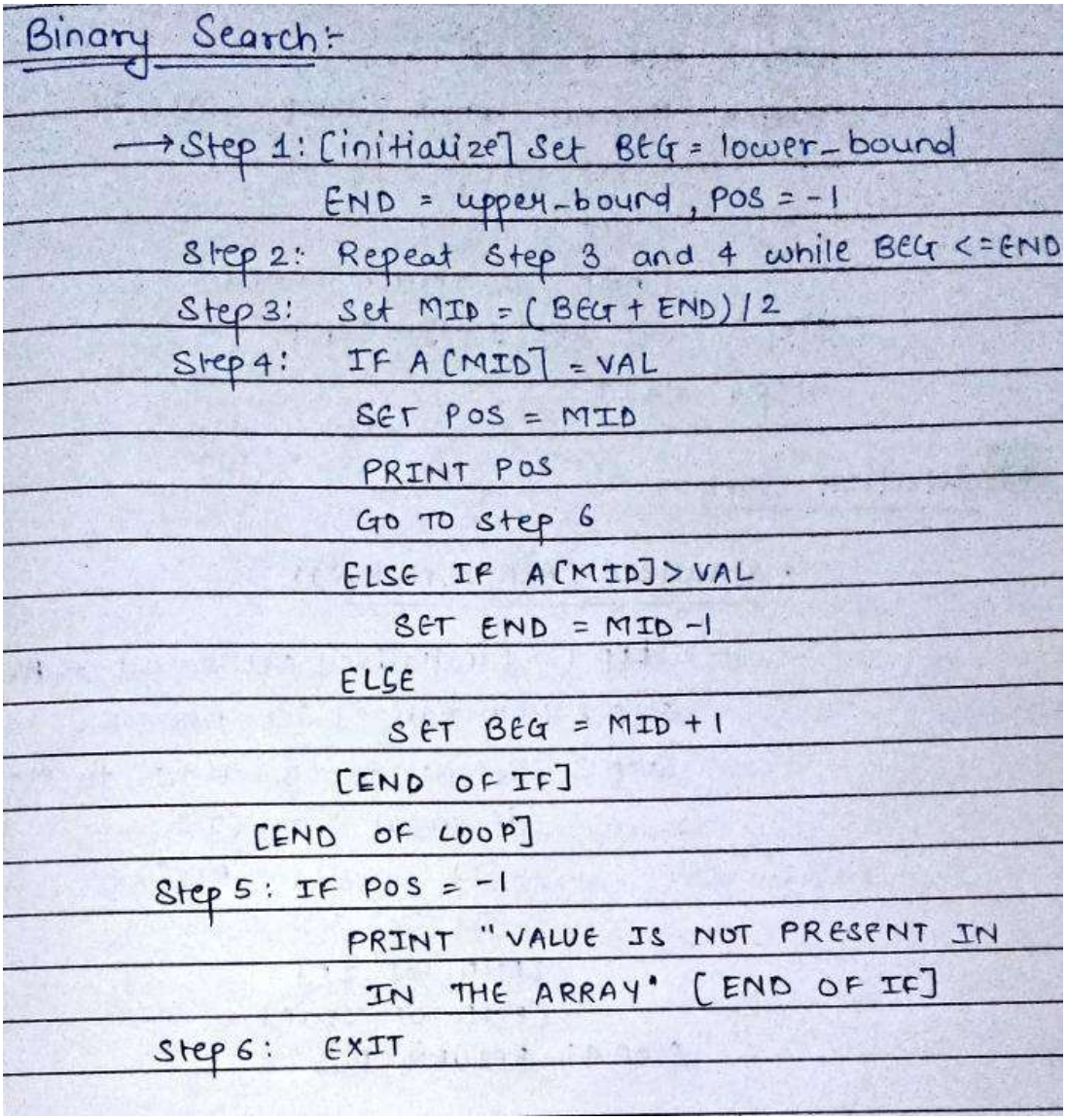
Big-θ notation represents the average or exact time complexity. It provides both upper and lower bounds.
Example: Linear search has θ(n) complexity on average.

**5 Write algorithms for linear search and binary search. Compare their performance.**

**Linear Search :** Linear search is the simplest searching technique. In this method, elements are checked one by one starting from the first element until the required element is found or the list ends. It does not require the data to be sorted, which makes it easy to use.




**Binary Search :** *Binary search is a faster searching technique that works only on sorted arrays. It repeatedly divides the array into two halves and compares the middle element with the search key.*




**Performance Comparison: Linear Search vs Binary Search**

- **Speed of Searching:** Linear search checks each element one by one, so it becomes slow as the number of elements increases. Binary search reduces the search space by half in every step, so it is much faster for large datasets.
- **Time Complexity:** Linear search has a worst-case time complexity of O(n), meaning it may check all elements. Binary search has a worst-case time complexity of O(log n), which grows very slowly even for large inputs.
- **Best Case Performance:** In linear search, the best case occurs when the element is at the first position and takes O(1) time. In binary search, the best case also takes O(1) when the middle element matches the key.
- **Effect of Data Size:** As data size increases, linear search performance degrades significantly. Binary search remains efficient even for very large datasets.