**Day 16 docs - 11th July 2025**

**Task -1 :**

**Write an algorithm / steps for selection sort.**

### **Algorithm Steps:**

#### Step 1: Start from the end

* We look for the largest number between index 0 to 4 (full array).
* The maximum is 64.
* Swap it with the last element (index 4):  
   → 64 and 11 are swapped →  
   New array: [11, 25, 12, 22, 64]

#### Step 2: Move to index 3

* Now we look at index 0 to 3.
* Max is 25.
* Swap 25 with the element at index 3 (22) →  
   New array: [11, 22, 12, 25, 64]

#### Step 3: Move to index 2

* Look at index 0 to 2.
* Max is 22.
* Swap 22 with index 2 (12) →  
   New array: [11, 12, 22, 25, 64]

#### **Step 4: Move to index 1**

* Look at index 0 to 1.
* Max is 12.
* Swap with itself → No change  
   Array remains: [11, 12, 22, 25, 64]

**Task -2:**

**Write a pseudo code for the selection sort**

**Pseudocode:**

procedure selectionSort(array, n)

for i ← n-1 to 1 step -1 do

maxIdx ← i

for j ← 0 to i-1 do

if array[j] ≥ array[maxIdx] then

maxIdx ← j

swap array[i] and array[maxIdx]

end procedure

**Task -3:**

### **Algorithm for Bubble Sort:**

1. Repeat the process n-1 times.
2. In each pass, compare each pair of adjacent elements.
3. If the left number is greater than the right, then swap them.
4. After each pass, the largest number bubbles up to its correct position at the end.
5. Continue until the whole array is sorted.

**Task - 4:**

**Pseudocode:**

procedure bubbleSort(array, n)

for i ← n-1 to 1 do

for j ← 1 to i do

if array[j-1] > array[j] then

swap array[j-1] and array[j]

end procedure

**Task - 5:**

### **Algorithm for Insertion Sort:**

1. Start from the second element (index 1), call it next.
2. Compare next with the elements before it.
3. Shift all larger elements one position to the right.
4. Insert next in the correct sorted position.
5. Repeat for all elements in the array.

**Task - 6:**

**Pseudocode:**

procedure insertionSort(array, n)

for i ← 1 to n-1 do

next ← array[i]

j ← i - 1

while j ≥ 0 and array[j] > next do

array[j + 1] ← array[j]

j ← j - 1

end while

array[j + 1] ← next

end procedure

**Task - 10:**

## **Advantages of Bubble Sort:**

1. Simple to understand and implement  
   * Very easy logic for beginners.
2. No extra space required  
   * It sorts the array in-place, using only a small amount of extra memory.
3. Detects if the array is already sorted  
   * With a small change (adding a swapped flag), it can finish early if no swaps happen.
4. Works well for small or nearly-sorted data

## **Disadvantages / Limitations of Bubble Sort (Poor Performance):**

1. Very slow for large lists  
   * Time Complexity is O(n²) in the worst and average case.
2. Performs unnecessary comparisons  
   * Even if the list is almost sorted, it still compares many elements.
3. Inefficient compared to other sorting algorithms  
   * Insertion Sort, Merge Sort, Quick Sort, and others are much faster.
4. Wastes time on already sorted parts of the list  
   * Without optimization, it blindly keeps checking sorted elements.
5. Not suitable for real-world big data applications

**Task - 12:**

## **Merge Sort Algorithm**

### **Step-by-Step Algorithm:**

1. If the array has 1 or 0 elements, it is already sorted — return.
2. Divide the array into two halves:  
   * Left half: from start to middle
   * Right half: from middle+1 to end
3. Recursively sort both halves.
4. Merge the two sorted halves into a single sorted array.

**Task - 13:**

**Pseudocode**

MERGE\_SORT(array, left, right)

if left < right then

mid ← (left + right) / 2

// Recursively sort first half

MERGE\_SORT(array, left, mid)

// Recursively sort second half

MERGE\_SORT(array, mid + 1, right)

// Merge the two halves

MERGE(array, left, mid, right)

end if

**Task -15:**

## **Quick Sort Algorithm:**

1. If the array has one or zero elements, it's already sorted.
2. Select a pivot (commonly the last element).
3. Partition the array into:  
   * Left part (elements < pivot)
   * Pivot (middle)
   * Right part (elements > pivot)
4. Recursively apply Quick Sort on left and right parts.

**Task - 16:**

**Pseudocode**

QUICK\_SORT(array, low, high)

if low < high then

Step 1: Partition the array

pivotIndex ← PARTITION(array, low, high)

Step 2: Recursively sort the two halves

QUICK\_SORT(array, low, pivotIndex - 1)

QUICK\_SORT(array, pivotIndex + 1, high)

end if