# **Day 16 - 11 July 2025**

**Document Name:**Day 16 - hmuvvala@ - Hari Gopal Muvvala

### Task 1: Selection sort - Algorithm

1. Start sorting an array of elements in ascending order.
2. Begin from the first index (index 0) and move up to the second last index (array.length - 2).
3. At each step, look at all elements from the current index to the end of the array.
4. Find the smallest element in that range.
5. Swap that smallest element with the element at the current index.
6. Repeat this for every index, growing the sorted portion from the left side.
7. When only one element is left at the end, the array is sorted.

### Task 2: Pseudocode for Selection Sort

function selectionSort(array):

for i from 0 to array.length - 2:

set minIndex = i

for j from i + 1 to array.length - 1:

if array[j] < array[minIndex]:

set minIndex = j

swap array[i] with array[minIndex]

### Task 4: Algorithm for Bubble Sort

1. Start sorting an array of elements in ascending order.
2. Begin from the first index of the array and repeat the process until the second last index.
3. At each pass:
   * Compare each pair of adjacent elements in the array from start to end.
   * If the left element is greater than the right, swap them.
4. After each pass, the largest unsorted element will have “bubbled” to its correct position at the end.
5. Repeat this for all elements, reducing the unsorted portion from the end after each pass.
6. Continue until the array is fully sorted.

### Task 5: Pseudocode for Bubble Sort

function bubbleSort(array):

for i from 0 to array.length - 2:

for j from 0 to array.length - i - 2:

if array[j] > array[j + 1]:

swap array[j] with array[j + 1]

### Task 7: Algorithm / Steps for Insertion Sort

1. Start sorting an array of elements in ascending order.
2. Begin from the second element (index 1) and move to the end of the array.
3. At each step, pick the current element (this is the one to insert).
4. Compare it with the elements on the left side (already sorted part).
5. While the left elements are greater than the current element:
   * Shift those elements one position to the right.
6. Insert the current element at its correct position.
7. Repeat this process for all elements until the array is fully sorted.

### Task 8: Pseudocode for Insertion Sort

function insertionSort(array):

for i from 1 to array.length - 1:

set current = array[i]

set j = i - 1

while j >= 0 and array[j] > current:

array[j + 1] = array[j] // shift element to the right

j = j - 1

array[j + 1] = current // insert current in correct position

### Task 10: Bubble Sort – Advantages and Disadvantages

#### Advantages:

* Easy to understand and write the code.
* Works without using extra memory.
* Keeps the order of the same values (stable sort).
* Can stop early if already sorted (with small change in code).

#### Disadvantages:

* Very slow for big arrays.
* It does too many swaps even if not needed.
* Not useful in real-time or large applications.
* Without optimization, it goes on even if it is already sorted.

### Task 12: Algorithm / Steps for Merge Sort

1. Start sorting an array of elements in ascending order.
2. If the array has only one element, it is already sorted. Stop.
3. Otherwise:
   * Divide the array into two halves (left and right).
   * Recursively sort both halves using the same steps.
4. After both halves are sorted, merge them together:
   * Compare the elements from both halves one by one.
   * Place the smaller element into a new merged array.
   * Continue until all elements from both sides are merged in order.
5. Return or store the merged array as the sorted result.

### Task 13: Pseudocode for Merge Sort

function mergeSort(array, low, high):

if low < high:

mid = (low + high) / 2

mergeSort(array, low, mid)

mergeSort(array, mid + 1, high)

merge(array, low, mid, high)

function merge(array, low, mid, high):

create temporary array temp[]

set left = low

set right = mid + 1

set index = 0

while left <= mid and right <= high:

if array[left] <= array[right]:

temp[index] = array[left]

left = left + 1

else:

temp[index] = array[right]

right = right + 1

index = index + 1

while left <= mid:

temp[index] = array[left]

left = left + 1

index = index + 1

while right <= high:

temp[index] = array[right]

right = right + 1

index = index + 1

copy all elements from temp[] back into array[low...high]

### Task 15: Quick Sort – Algorithm / Steps

Quick Sort also uses Divide and Conquer like Merge Sort.

But instead of merging sorted halves, it picks a pivot and partitions the array around it.

1. Start sorting an array of elements in ascending order.
2. If the array has only one element (or low ≥ high), stop — it’s already sorted.
3. Pick a **pivot element** (commonly the first or last element).
4. Partition the array:
   * Move all elements **smaller than the pivot** to the left side.
   * Move all elements **greater than or equal to the pivot** to the right side.
   * Place the pivot at its **correct sorted position**.
5. Recursively apply the same steps to the **left subarray** and the **right subarray**.
6. Continue until the whole array is sorted.

### Task 16: Pseudocode for Quick Sort

function quickSort(array, low, high):

if low < high:

pivotIndex = partition(array, low, high)

quickSort(array, low, pivotIndex - 1)

quickSort(array, pivotIndex + 1, high)

function partition(array, low, high):

pivot = array[low]

set i = low + 1

set j = high

while i <= j:

while i <= high and array[i] <= pivot:

i = i + 1

while j >= low and array[j] > pivot:

j = j - 1

if i < j:

swap array[i] and array[j]

swap array[low] and array[j] // place pivot at correct position

return j // return pivot index

### Quick Sort vs Merge Sort — Comparison Table

| **Feature** | **Quick Sort** | **Merge Sort** |
| --- | --- | --- |
| **Approach** | Divide & Conquer (using **partitioning**) | Divide & Conquer (using **merging**) |
| **Speed (Time Complexity)** | Best: O(n log n) |  |
| Average: O(n log n) |  |  |
| Worst: ❌ O(n²) | Best, Avg, Worst: ✅ **O(n log n)** |  |
| **Space Complexity** | ✅ In-place (O(log n) recursion stack) | ❌ Needs extra space (O(n)) for merging |
| **Stability** | ❌ Not stable by default | ✅ Stable |
| **Recursive** | ✅ Yes | ✅ Yes |
| **Faster in Practice?** | ✅ Usually faster than Merge Sort on arrays | ❌ Slightly slower than Quick Sort in most cases |
| **Sorting Mechanism** | Puts pivot in correct place, sorts around it | Merges two sorted subarrays |
| **Use Case** | Fast, in-place sort for arrays | Linked lists, external sorting (files), stable sorting needed |

### Selection Sort – Notes

* Go one by one and select the **smallest** element.
* Swap only once per pass.
* ❌ Always O(n²), even if already sorted.
* ✅ In-place but ❌ not stable by default.
* Good for: learning basics, but not practical.

### Bubble Sort – Notes

* Repeatedly swap adjacent elements if out of order.
* **Largest element bubbles to the end** in each pass.
* ✅ Can optimize with swapped flag → O(n) best case.
* ❌ Still O(n²) in worst case.
* ✅ Stable, ✅ in-place, good for teaching.

### Insertion Sort – Notes

* Like sorting cards in your hand.
* Pick element, compare with left, shift and insert.
* ✅ O(n) best case (already sorted), ❌ O(n²) worst.
* ✅ In-place and ✅ stable.
* Great for small arrays or nearly sorted data.

### Merge Sort – Notes

* Uses **divide and conquer**.
* Recursively splits → sorts → merges back.
* ✅ Always O(n log n), ✅ stable.
* ❌ Needs extra space → not in-place.
* Good for linked lists and external sorting.

### Quick Sort – Notes

* Picks a **pivot**, partitions array, and sorts sides.
* ✅ O(n log n) average, ❌ O(n²) worst if pivot bad.
* ✅ In-place, ❌ not stable.
* Usually **faster than Merge Sort** on arrays.
* Real-world use case: fast sorting in memory.

### Add-on Notes\_Day 15

#### 1. Difference between Binary Tree and Binary Search Tree (BST)

##### Structure:

* Binary Tree: A general tree where each node can have at most two children. No specific order is maintained between nodes.
* Binary Search Tree (BST): A binary tree with a rule — all left children are smaller than the parent, and all right children are greater.

##### Operations:

* Binary Tree: Mainly used for structure-based problems (e.g., expression trees).
* BST: Supports efficient searching, insertion, and deletion based on the ordering rule.

##### Summary:

A BST is a special case of a binary tree where data is ordered to allow faster operations.

#### 2. Why is BST better than linear search in a sorted array

* Linear search checks one element at a time — time complexity is O(n).
* A Binary Search Tree allows searching by skipping half of the data at each level — O(log n) if balanced.
* In a sorted array, binary search (similar logic as BST) is more efficient than scanning one by one.

#### 3. Difference between static and dynamic arrays

| **Feature** | **Static Array** | **Dynamic Array (e.g., ArrayList in Java)** |
| --- | --- | --- |
| Size | Fixed at the time of creation | Can grow/shrink during runtime |
| Memory | May waste memory or overflow | Efficient memory usage |
| Flexibility | Not flexible | More flexible |
| Insertion | Difficult unless at end | Easier insertion, especially at the end |
| Java Example | int[] arr = new int[5]; | ArrayList<Integer> list = new ArrayList<>(); |

#### 4. BFS vs DFS: Preferred for shortest path in unweighted graphs

* BFS is preferred for shortest path in unweighted graphs.
* BFS explores nodes in increasing distance order from the source.
* This ensures the first time we reach a node is via the shortest path.
* DFS may go deep and miss shorter paths.