# **Day 18 - 18 July 2025**

**Document Name:**Day 18 - hmuvvala@ - Hari Gopal Muvvala

### Task 04 – What are the Applications of Heap Sort?

#### Real-World Applications of Heap Sort:

1. **Priority Queues**
   * Heap sort is based on the **heap data structure**, which is the foundation for **priority queues**.
   * Examples: CPU job scheduling, Dijkstra’s algorithm for shortest path.
2. **Heaps in Graph Algorithms**
   * Algorithms like **Dijkstra’s** and **Prim’s** use **min-heaps** internally to pick the next minimum element efficiently.
3. **Order Statistics (kth largest/smallest)**
   * We can use heap sort logic to find the **kth largest or smallest** element efficiently.
4. **Scheduling Systems**
   * When multiple tasks with different priorities need to be scheduled, heaps help sort them by priority.
5. **Real-Time Systems (Deterministic Performance)**
   * Since Heap Sort always takes **O(n log n)** time (unlike Quick Sort), it is preferred in systems where **predictable timing is important**.
6. **Memory-Constrained Devices**
   * It is an **in-place sorting algorithm** (no extra memory used), so it’s good for systems with **limited memory**.

#### Where NOT to Use Heap Sort:

* When **stability is important** (e.g., if two equal items must stay in the same order).
* When sorting small datasets — **insertion sort** may be faster.

### Task 05 – BFS vs Recursive DFS

#### Q: Any difference between BFS and recursive-style DFS approaches?

##### BFS (Breadth-First Search)

* **Searches level by level**
* Always explores all neighbors at the **current depth** before going deeper
* Uses a **Queue**
* Typically **iterative**

###### Characteristics:

* Finds the **shortest path** in unweighted graphs
* Good for **layered problems** (shortest distance, fewest moves, etc.)

##### DFS (Depth-First Search – Recursive Style)

* Explores **deepest nodes first**
* Backtracks once it reaches a dead end
* Implemented using **recursion** (uses the **call stack**)

###### Characteristics:

* Good for problems like:
  + **Cycle detection**
  + **Pathfinding**
  + **Topological sort**
* More **memory efficient** than BFS if graph is sparse

##### Key Difference from Trainer’s Line:

“*It will achieve the same result but emphasizes on recursive style using same level-order logic with explicit queue management.*”

This means:

* **Both can reach all nodes**, but their **order and approach differ**
* **BFS is queue-based**, always iterative
* **DFS can be recursive**, relies on function call stack

#### 📌 Notes Summary for Task 5:

| **Aspect** | **BFS** | **Recursive DFS** |
| --- | --- | --- |
| Strategy | Level-wise | Depth-first |
| Uses | Queue | Call Stack (Recursion) |
| Path Found | Shortest path (unweighted) | Not guaranteed shortest |
| Memory | More (queue grows wide) | Less (stack goes deep) |
| Nature | Iterative | Recursive |

### Task 06 – How Heap Sort Works (Explanation for Notes / Interview)

#### Step-by-Step Process of Heap Sort:

1. **Build a Max Heap** from the input array
   * A Max Heap is a binary tree where each parent node is **greater than or equal** to its children
   * The largest element ends up at the root (index 0)
2. **Swap the root element** (largest) with the last element in the heap
   * Now, the largest element is placed in its correct sorted position
3. **Reduce the heap size by 1**
   * Exclude the last element from the heap (it’s already sorted)
4. **Heapify the root** again to maintain the Max Heap property
5. **Repeat steps 2–4** until the entire array is sorted

#### Example (Sorting [4, 10, 3, 5, 1]):

1. Build Max Heap → [10, 5, 3, 4, 1]
2. Swap 10 with 1 → [1, 5, 3, 4, 10]
3. Heapify again → [5, 4, 3, 1, 10]
4. Swap 5 with 1 → [1, 4, 3, 5, 10]
5. Heapify again → [4, 1, 3, 5, 10]
6. Continue until → [1, 3, 4, 5, 10]

#### Key Properties of Heap Sort:

| **Property** | **Value** |
| --- | --- |
| Time Complexity | O(n log n) (all cases) |
| Space Complexity | O(1) (in-place) |
| Stability | ❌ Not stable |
| Sorting Order | Ascending (via Max Heap) or Descending (via Min Heap) |

#### 📌 Final Notes (write this in your notebook):

* Heap Sort is a **comparison-based**, **in-place**, and **universal time** sorting algorithm
* It is best when **memory is limited** and performance must be **predictable**
* Not suitable when **stability is required**

### Task 07 – How Do Recursive Functions Maintain State?

#### Question from Trainer:

How does a recursive function “remember” local variables in each call?

#### Answer:

#### Using the Call Stack

When a recursive function is called:

* The current function’s state (parameters, local variables, return address) is **pushed onto the call stack**
* When that recursive call completes, it **pops** and resumes the previous one
* This is how recursion “remembers” values from earlier calls

#### 🧠 Layman Explanation:

Imagine a **stack of plates**:

* You keep adding one plate (function call) on top
* You always remove (complete) the **topmost** one first

➡️ This is **LIFO (Last In, First Out)** — the core behavior of recursion

#### 🔹 Java Example: Factorial (shows state maintenance)

int factorial(int n) {

if (n == 1) return 1;

return n \* factorial(n - 1);

}

#### For factorial(4), the call stack looks like:

| **Function Call** | **Local n** |
| --- | --- |
| factorial(4) | 4 |
| factorial(3) | 3 |
| factorial(2) | 2 |
| factorial(1) | 1 |

Each call waits for the next one, keeping its **own local value of n**.

#### 🔍 Summary Table for Notes:

| **Option from MCQ-style question** | **Correct?** | **Reason** |
| --- | --- | --- |
| 1. New thread per recursive call | ❌ | Recursion doesn’t use new threads |
| 2. Global/static variables | ❌ | Local variables are used, not global |
| 3. ✅ **Call stack tracks local variables** | ✅ | ✔ Correct! State is maintained on stack |
| 4. Heap memory replicates the structure | ❌ | Heap is for objects, not call state |

#### 🗒 Final Note:

**Each recursive call has its own independent local state**, maintained on the **call stack** until the call returns.

### Task 09 (Quiz Recap – Priority Queue)

#### Q1: Which property of a priority queue differentiates it from a regular queue?

**Correct Answer:**

**3. Elements are dequeued based on their priority, not their insertion order**, often implemented using a binary heap.

This is the key difference:

* **Regular Queue**: FIFO (First In First Out)
* **Priority Queue**: Based on **priority values**, not order of arrival

#### Q2: What is the main purpose of using a binary heap in a priority queue?

**Correct Answer:**

**3. To guarantee constant-time insertion and logarithmic-time deletion**

* In a **Min Heap** (or Max Heap), the **highest/lowest priority element** is always at the top.
* Insert → O(log n)
* Delete (min/max) → O(log n)
* Peek → O(1)

### Task 15: Algorithm for Radix Sort

1. Find the maximum number in the array to determine the number of digits.

2. Start with the least significant digit (LSD).

3. Use counting sort as a subroutine to sort numbers based on current digit.

4. Repeat step 3 for every digit (units, tens, hundreds, etc.) until the most significant digit.

### Task 16: Pseudocode for Radix Sort

RadixSort(arr, n)

max ← find maximum element in arr

exp ← 1

while (max / exp) > 0

CountingSort(arr, n, exp)

exp ← exp \* 10

CountingSort(arr, n, exp)

output[n] ← empty array

count[10] ← all zero

for i ← 0 to n-1

index ← (arr[i] / exp) % 10

count[index]++

for i ← 1 to 9

count[i] += count[i-1]

for i ← n-1 downto 0

index ← (arr[i] / exp) % 10

output[count[index] - 1] ← arr[i]

count[index]--

for i ← 0 to n-1

arr[i] ← output[i]