MERGE SORT

1. Definition

Merge Sort is a **divide-and-conquer**, **recursive**, comparison-based sorting algorithm. It divides the array into two halves, sorts each half recursively, and then **merges** the two sorted halves.

2. Key Idea (Divide → Conquer → Combine)

1. **Divide:** Split the array into two halves
2. **Conquer:** Recursively sort both halves
3. **Combine:** Merge the two sorted halves into one sorted list

3. Example (Step-by-Step)

Array: `[8, 4, 7, 3, 2, 6, 5, 1]`

✔ Step 1: Divide

`[8, 4, 7, 3]` and `[2, 6, 5, 1]`

✔ Step 2: Divide further

Left → `[8, 4]`, `[7, 3]`
Right → `[2, 6]`, `[5, 1]`

✔ Step 3: Sort single elements

```
[8] [4] → merge → [4, 8]
[7] [3] → merge → [3, 7]
[2] [6] → merge → [2, 6]
[5] [1] → merge → [1, 5]
```

✔ Step 4: Merge pairs

`[4, 8]` + `[3, 7]` → `[3, 4, 7, 8]`
`[2, 6]` + `[1, 5]` → `[1, 2, 5, 6]`

✔ **Step 5: Merge the final two halves**

`[3, 4, 7, 8]` + `[1, 2, 5, 6]`
→ **Sorted array:** `[1, 2, 3, 4, 5, 6, 7, 8]`

4. Merge Sort Algorithm

```
// Algo MergeSort
// Input: Array A with indexes low and high
// Output: Sorted array A

MergeSort(A, low, high)
{
   if (low < high)
   {
      mid = (low + high) / 2

      MergeSort(A, low, mid)
      MergeSort(A, mid + 1, high)

      Merge(A, low, mid, high)
   }
}


// Algo Merge
// Input: Array A, indexes low, mid, high
// Output: Merged sorted subarray A[low...high]

Merge(A, low, mid, high)
{
   create temporary array B
   i = low
   j = mid + 1
   k = low
   while (i ≤ mid AND j ≤ high)
   {
      if (A[i] ≤ A[j])
      {
         B[k] = A[i]
         i = i + 1
      }
```

```
        else
        {
            B[k] = A[j]
            j = j + 1
        }
        k = k + 1
    }
    // Copy the remaining left subarray
    while (i ≤ mid)
    {
        B[k] = A[i]
        i = i + 1
        k = k + 1
    }
    // Copy the remaining right subarray
    while (j ≤ high)
    {
        B[k] = A[j]
        j = j + 1
        k = k + 1
    }
    // Copy B back to A
    for (x = low; x ≤ high; x = x + 1)
        A[x] = B[x]
}
```

5. Properties

Property	Value
Algorithm Type	Divide & Conquer
Stable	✔ Yes
In-place	❌ No (needs extra space)
Adaptive	❌ No
Recursive	✔ Yes
Suitable for large data	✔ Yes

6. Summary of Worst-Case Efficiency

Component	Worst Case
Recursive Levels	log n
Work per Level	n
Total Time Complexity	**O(n log n)**
Space Complexity	**O(n)**
Comparisons	≤ n log n
Stable?	Yes
In-place?	No

9. Advantages

- Very efficient for large datasets
- Guaranteed **O(n log n)** performance
- Stable
- Parallelizable
- Perfect for linked lists

10. Disadvantages

- Requires extra space O(n)
- Slower than quicksort in practice for arrays
- Not in-place
- Heavy recursive overhead

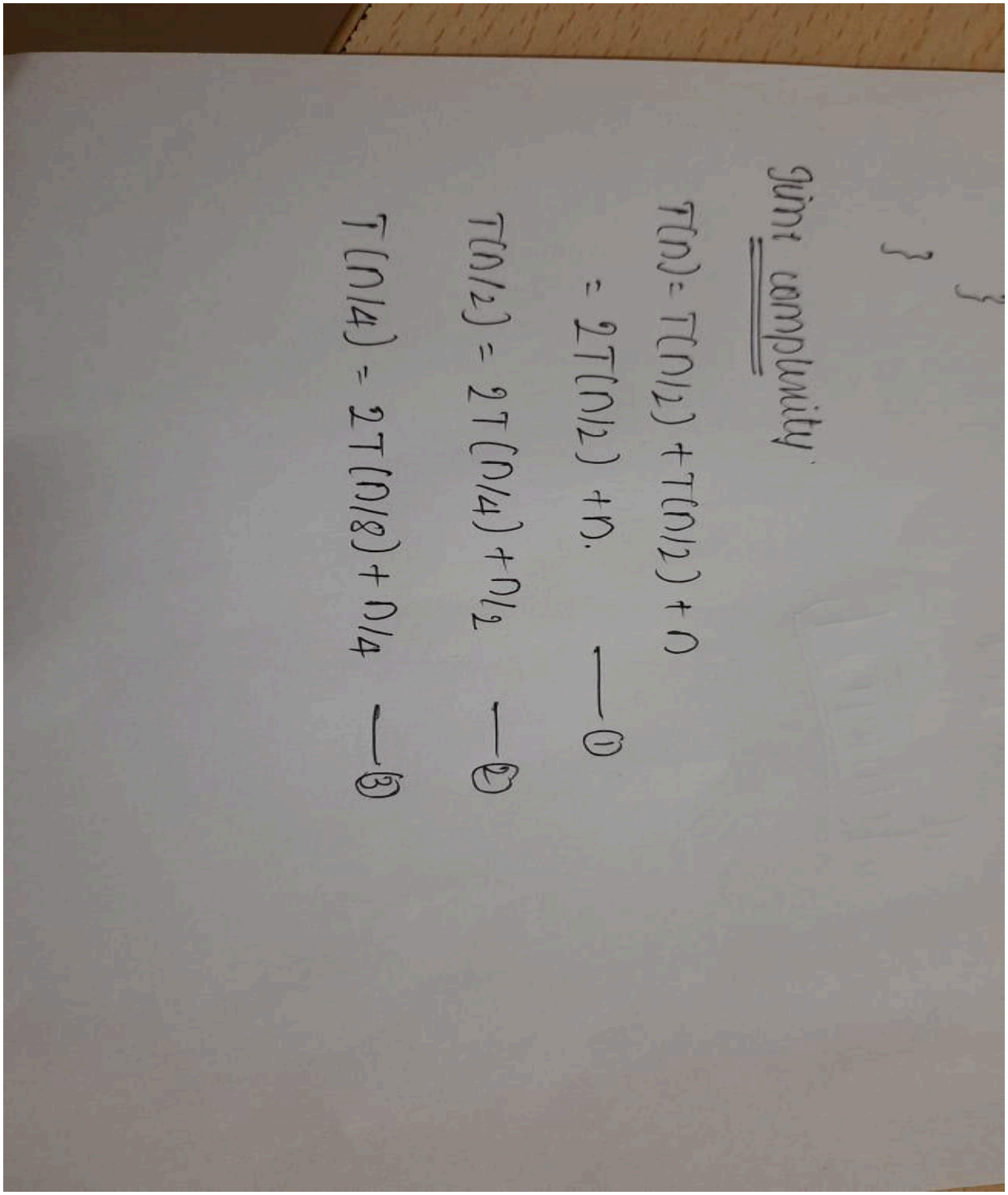
11. Real-Life Analogy

Sorting papers by splitting them into smaller piles, sorting each pile, and merging them back.

12. Short Exam Notes

- Merge sort uses **divide and conquer**.
- Splits recursively until a single element remains.
- Merges sorted halves into a single sorted array.
- Time complexity (all cases) = **O(n log n)**.
- Space complexity = **O(n)**.
- Stable but not in place.
- Efficient for large datasets.

13. Efficiency Analysis



}
}

Time complexity

T(n) = T(n/2) + T(n/2) + n
= 2T(n/2) + n. — (1)

T(n/2) = 2T(n/4) + n/2 — (2)

T(n/4) = 2T(n/8) + n/4 — (3)

$T(n) = 2(2T(n/2) + n/2) + n$

$= 4T(n/4) + 2n$

$= 4(2T(n/8) + n/4) + 2n$

$= 8T(n/8) + 3n$

for k^{th} iteration.

$T(n) = 2^k T(n/2^k) + kn.$

Subst $2^k = n$.

$T(n) = nT(1) + kn$

$= n + kn.$

$2^k = n$

$\log 2^k = \log n.$

$k \log_2 2 = \log_2 n$

$\boxed{k = \log_2 n}$

$\Rightarrow T(n) = n + n(\log_2 n)$

$\therefore$ O(n)

$\boxed{O(n \log n)}$