📘 Understanding Sorting and MergeSort

🔢 What is Sorting?

Sorting is the process of arranging elements in a specific order—typically **ascending** or **descending**. It is a fundamental operation in computer science, used in searching, data analysis, and optimization problems.

Common Sorting Criteria:

- **Numerical order** (e.g., 1, 2, 3...)
- **Lexicographical order** (e.g., "apple", "banana", "cherry")
- **Custom orderings** (e.g., by date, priority, etc.)

🧠 Why Sorting Matters

Efficient sorting:

- Improves **search performance** (e.g., enables binary search)
- Helps in **data normalization**
- Is often a **preprocessing step** in complex algorithms (e.g., graph algorithms, database queries)

🧮 Types of Sorting Algorithms

Sorting algorithms are generally categorized by:

- **Time complexity** e.g., O(n^2), O(n log n)
- **Space complexity**
- **Stability** (whether equal elements retain their original order)
- **In-place vs out-of-place** (whether extra memory is used)

Examples:

Algorithm	Time Complexity	Stable	In-Place
Bubble Sort	O(n^2)	Yes	Yes
Insertion Sort	O(n^2)	Yes	Yes
MergeSort	O(n log n)	Yes	No
QuickSort	O(n log n)	No	Yes
HeapSort	O(n log n)	No	Yes

🔀 MergeSort: A Divide and Conquer Algorithm

🧩 Concept

MergeSort is a classic example of the **divide and conquer** paradigm. It works by:

1. **Dividing** the array into two halves.
2. **Recursively sorting** each half.
3. **Merging** the two sorted halves into a single sorted array.

🧠 Key Idea

Instead of sorting the entire array at once, MergeSort breaks the problem into smaller subproblems, solves them independently, and then combines the results.

🔍 Step-by-Step Breakdown

Imagine sorting the array: `[38, 27, 43, 3, 9, 82, 10]`

1. **Divide**: Split into halves until each subarray has one element:
   - `[38, 27, 43]` and `[3, 9, 82, 10]`
   - Further split until: `[38]`, `[27]`, `[43]`, `[3]`, `[9]`, `[82]`, `[10]`
2. **Conquer**: Recursively sort each half (trivial for single-element arrays)
3. **Merge**: Combine sorted arrays:
   - Merge `[38]` and `[27]` → `[27, 38]`
   - Merge `[27, 38]` and `[43]` → `[27, 38, 43]`
   - Merge `[3]` and `[9]` → `[3, 9]`, then merge with `[82]` → `[3, 9, 82]`, and so on
4. Final merge gives the fully sorted array: `[3, 9, 10, 27, 38, 43, 82]`

⏱️ Time and Space Complexity

- **Time Complexity**:
  - Best, Average, Worst: O(n log n)
  - Because the array is split log n times and merging takes O(n)
- **Space Complexity**:
  - O(n) due to the temporary arrays used during merging

✅ Advantages of MergeSort

- **Stable**: Maintains the relative order of equal elements
- **Predictable performance**: Always O(n log n)
- **Good for linked lists** and external sorting (e.g., sorting data on disk)

⚠️ Disadvantages

- **Not in-place**: Requires additional memory
- **Slower for small datasets** compared to simpler algorithms like Insertion Sort

---

MergeSort Algorithm Pseudocode

Below are the pseudocodes for the Merge and MergeSort methods that make up the algorithm.

Merge Pseudocode

```
function Merge(left, right) is
    var result := empty vector
    while left is not empty and right is not empty do
    {
         if first(left) ≤ first(right) then
             add first(left) to result
             remove first from left
         else
             add first(right) to result
             remove first from right
    }

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
    {
        add first(left) to result
        remove first from left
    }
    while right is not empty do
    {
        add first(right) to result
        remove first from right
    }
    return result
```

MergeSort Pseudocode

```
function MergeSort(vector m) is
    // Base case. A vector of zero or one elements is sorted, by definition.
    if length of m <= 1 then
        return m

    // Recursive case. First, divide the vector into equal-sized subvectors
    // consisting of the first half and second half of the vector.
    // This assumes vectors start at index 0.
    var left := empty vector
    var right := empty vector
    for i = 0 to length(m) do
        if i < (length of m)/2 then
            add m[i] to left
        else
            add m[i] to right

    // Recursively sort both subvectors.
    left := MergeSort(left)
    right := MergeSort(right)

    // Then Merge the now-sorted subvectors.
    return Merge(left, right)
```

🎯 Quiz!

Here's a short quiz on the topic: quiz

Footer Separator

🔭 Markdown Viewer

How to view the markdown files in a browser...

- Markdown Viewer

🧠 Lecture Practices

Here are the lecture Practices...

- Day 4
- Day 5
- Day 6

🔍 Lecture Quizzes

Here are the lecture quizzes...

- Day 4
- Day 5
- Day 6

Weekly Topics

Here are the topics for the week...

- Recursion
- Pseudocode
- Sorting
- Searching
- Maps
- Time Complexity