Introduction to Merge Sort

Merge Sort is a widely used comparison-based sorting algorithm that follows the *divide-and-conquer* paradigm. It is **stable**, has a consistent time complexity of $\mathcal{O}(n \log n)$, and is especially suitable for:

- Sorting linked lists,
- Sorting large datasets,
- Use cases where consistent performance matters more than in-place memory use.

How Merge Sort Works

The core idea behind Merge Sort is simple:

1. **Divide:** Split the array into two equal halves.
2. **Conquer:** Sort both halves recursively (Top-Down) or iteratively (Bottom-Up).
3. **Combine:** Merge the two sorted halves into one sorted array.

The merging step is key—it merges two sorted lists into a larger sorted list in linear time.

Top-down Merge Sort

Top-down merge sort is the traditional, recursive approach where the array is repeatedly divided into smaller subarrays until each subarray has only one element. Then, the algorithm merges these subarrays in a sorted manner as it returns from each recursive call.

Algorithm

```
void mergeSort(int arr[], int left, int right) {
    if (left >= right)
        return;
    int mid = (left + right) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Example

Sort the array [38, 27, 43, 3, 9, 82, 10] using Top-Down Merge Sort:

1. Divide: [38, 27, 43, 3, 9, 82, 10] → [38, 27, 43] and [3, 9, 82, 10]
2. Divide: [38, 27, 43] → [38], [27], [43]
3. Merge: [38] and [27] → [27, 38]
   Merge: [27, 38] and [43] → [27, 38, 43]
4. Divide: [3, 9, 82, 10] → [3], [9], [82], [10]
5. Merge: [3] and [9] → [3, 9]
   Merge: [82] and [10] → [10, 82]
   Merge: [3, 9] and [10, 82] → [3, 9, 10, 82]
6. Final Merge: [27, 38, 43] and [3, 9, 10, 82] → [3, 9, 10, 27, 38, 43, 82]

Time Complexity

Case	Time Complexity	Explanation
Best Case	$O(n \log n)$	Always divides into halves and merges, even if already sorted.
Average Case	$O(n \log n)$	Performs consistently regardless of initial order.
Worst Case	$O(n \log n)$	Maximum recursive depth and merging still yields same behavior.

Space Complexity

Component	Space Usage	Explanation
Auxiliary Array	$O(n)$	Needed to merge sorted subarrays.
Recursion Stack	$O(\log n)$	Due to recursive function calls.
Total	$O(n + \log n)$	Combined space complexity.

Bottom-Up Merge Sort

Bottom-up merge sort is a non-recursive approach to merge sort. Instead of recursively splitting the array, it starts by merging individual elements (subarrays of size 1), then merges pairs of size 2, then 4, and so on, doubling the size each time until the array is completely sorted.

Algorithm (Bottom-Up)

:

```
mergeSort(arr):
    n = arr.length
    for size = 1 to n step size *= 2:
        for left = 0 to n - size step size *= 2:
            mid = left + size - 1
            right = min(left + 2*size - 1, n - 1)
            merge(arr, left, mid, right)
```

Example

Sort the array [38, 27, 43, 3] using Bottom-Up Merge Sort:

- Step 1 (size = 1): Merge [38] and [27] → [27, 38], Merge [43] and [3] → [3, 43]
- Step 2 (size = 2): Merge [27, 38] and [3, 43] → [3, 27, 38, 43]

Time Complexity – Bottom-Up Merge Sort

Case	Time Complexity	Explanation
Best Case	$O(n \log n)$	Always merges $\log n$ times, regardless of the initial order.
Average Case	$O(n \log n)$	Deterministic iteration and merging without branching.
Worst Case	$O(n \log n)$	All cases behave similarly due to uniform iterative structure.

Space Complexity – Bottom-Up Merge Sort

Component	Space Usage	Explanation
Auxiliary Array	$O(n)$	Required to hold temporary merged results during merge steps.
Recursion Stack	$O(1)$	No recursive function calls are made in this approach.
Total	$O(n)$	Only the merge buffer contributes to the overall space.

Applications of Merge Sort

- **Sorting linked lists:** Efficient due to non-contiguous memory access; merge sort does not require random access.
- **External sorting:** Suitable for sorting large datasets stored on disk, as it minimizes data movement.
- **Counting inversions in arrays:** Can be modified to count the number of inversions efficiently in $O(n \log n)$ time.
- **Stable sorting for large datasets:** Preferred when a stable sort is required for large or complex data structures.
- **Merge phase in database operations:** Useful in implementing database joins and multi-way merging in data pipelines.

Lab exercises

Exercise 1

Sorting Weather Data for Climate Analysis: A climate research team collects daily temperature readings (in °C) from multiple weather stations. Due to sensor malfunctions, the data arrives out of order. Your task is to sort the temperatures in **ascending order** to identify trends like heatwaves or cold spells.

Input:

- A list/array of floating-point numbers representing temperatures.
- Example: `{23.5, 18.2, 25.1, 20.7, 19.4}`

Output:

- A sorted list of temperatures in ascending order.
- Example Output: `18.2 19.4 20.7 23.5 25.1`

Instructions:

1. Use the **Top-Down Merge Sort** approach (recursive).
2. Break down the problem into smaller subarrays until each subarray has a single element.
3. Merge the subarrays while comparing temperatures to ensure ascending order.
4. Ensure your solution handles duplicate temperatures correctly.

Why Merge Sort?

- Stable sorting (preserves order of equal elements).
- Efficient for large datasets (O(n log n) time complexity).

Exercise 2

Alphabetizing a Library's Book Inventory: A librarian needs to sort book titles alphabetically for a new digital catalog. The titles are currently in random order, and some start with numbers (e.g., "1984").

Input:

- A list/array of book titles (strings)
- Example: {"The Hobbit", "1984", "To Kill a Mockingbird", "Brave New World"}

Output:

- A sorted list of titles in **lexicographical order** (numbers first, then letters)
- Example Output: 1984 Brave New World The Hobbit To Kill a Mockingbird

Instructions:

1. Use the **Bottom-Up Merge Sort** approach (iterative)
2. Compare strings using standard lexicographical order (case-sensitive)
3. Test edge cases:
   - Empty strings
   - Titles with special characters (e.g., "C# Programming")

Why Merge Sort?

- Consistent O(n log n) performance
- Works well with strings (no overhead from pivot selection like in QuickSort)

Exercise 3

Prioritizing High-Value Sales Transactions: An e-commerce platform wants to analyze its highest-revenue transactions first. The sales data is unsorted, and each transaction is a floating-point value (in USD).

Input:

- A list/array of transaction amounts
- Example: {150.0, 99.99, 499.95, 75.5, 200.0}

Output:

- A sorted list of transactions in **descending order**
- Example Output: 499.95 200.0 150.0 99.99 75.5

Instructions:

1. Modify the **Top-Down Merge Sort** to sort in descending order
2. The merge step should place larger numbers first
3. Discuss how this helps the business (e.g., identifying top customers)

Why Merge Sort?

- Stable sorting ensures identical transactions retain their original order
- Predictable performance for financial data

Exercise 4

Sorting Employee Names by Badge Printing Priority: An HR department needs to print employee badges in order of name length (shortest first) to optimize space. Ties (names of equal length) should be broken alphabetically.

Input:

- A list/array of employee names (strings)
- Example: {"Alice", "Bob", "Charlie", "Dave"}

Output:

- Names sorted by:
  1. Length (ascending order)
  2. Alphabetical order (for names of equal length)
- Example Output: Bob Dave Alice Charlie

Instructions:

1. Implement using **Bottom-Up Merge Sort** (iterative approach)
2. Primary sorting key: name length (|name|)
3. Secondary sorting key: lexicographical order

Why Merge Sort?

- **Stability**: Preserves alphabetical order for equal-length names
- **Predictable performance**: Guaranteed $O(n \log n)$ time complexity
- **No worst-case** $O(n^2)$ scenarios (unlike QuickSort)
- **Efficient** for medium-sized datasets typical in HR systems

Technical Considerations:

- Handle edge cases:
  - Empty names
  - Names with special characters
  - Large datasets (discuss scalability)
- The comparison function should first check length, then use string comparison if lengths are equal

Exercise 5

Compare Sorted Halves Using Merge Sort: You are given a string S of length $2N$, consisting only of digits from 0 to 9. Your task is to divide this string into two halves:

- The **first** N **digits** form the first half.
- The **last** N **digits** form the second half.

You must perform the following steps:

1. Convert each half into an array of digits.
2. Sort both halves using **Merge Sort**. You may choose either:
   - Top-Down Recursive Merge Sort
   - Bottom-Up Iterative Merge Sort
3. After sorting, compute the **sum of digits** in each half.
4. Output:
   - The sorted digits in each half.
   - The sum of each half.
   - A message indicating which half has the greater sum or whether the sums are equal.

Function Signature

```
void solveTestCase(int N, const string& S);
```

Input Format

- The first line contains an integer T — the number of test cases.
- For each test case:
  - The first line contains an integer N.
  - The second line contains a string S of length $2N$.

Output Format

For each test case, print:

- The sorted digits in both halves on separate lines.
- The result message:
  - "`Result: First half has greater sum`"
  - "`Result: Second half has greater sum`"
  - "`Result: Equal sums`"

Constraints

- $1 \leq T \leq 10$
- $1 \leq N \leq 10^4$
- $|S| = 2N$
- S contains only digits `0` to `9`

Sample Input

```
2
3
912345
2
2211
```

Sample Output

Note

In the first test case:

- $S =$ `"912345"`, split as $S_1 =$ `"912"`, $S_2 =$ `"345"`
- After sorting: $S_1 = [1, 2, 9]$, $S_2 = [3, 4, 5]$
- Sums: 12 and 12 $\Rightarrow$ Result: Equal sums

In the second test case:

- $S =$ `"2211"`, split as $S_1 =$ `"22"`, $S_2 =$ `"11"`
- After sorting: $S_1 = [2, 2]$, $S_2 = [1, 1]$
- Sums: 4 and 2 $\Rightarrow$Result: First half has greater sum