Step-by-Step Explanation of Merge Algorithm

The given algorithm is an implementation of the Merge step in Merge Sort. Let's break it down line by line for a clear understanding.

🔹 Purpose of the Algorithm

This algorithm merges two sorted subarrays A[p:q]A[p:q] and A[q+1:r]A[q+1:r] into a single sorted subarray.

🔹 **Understanding the Parameters**

- **AA → The original array.**
- **pp → Starting index of the first subarray.**
- **qq → Ending index of the first subarray.**
- **rr → Ending index of the second subarray.**

👉 **Goal: Merge the two sorted subarrays:**

- **First subarray: A[p],A[p+1],...,A[q]A[p], A[p+1], ..., A[q]**
- **Second subarray: A[q+1],A[q+2],...,A[r]A[q+1], A[q+2], ..., A[r]**

🔹 Step-by-Step Breakdown of the Algorithm

Step 1: Calculate the Sizes of Two Subarrays

1 n1 = q - p + 1

2 n2 = r - q

- **n1n1 is the number of elements in the first subarray.**
- **n2n2 is the number of elements in the second subarray.**

🔹 **Why these formulas?**

- **The first subarray starts at pp and ends at qq, so its size is:**
  n1=q−p+1n1 = q - p + 1
- **The second subarray starts at q+1q+1 and ends at rr, so its size is:**
  n2=r−qn2 = r - q

Step 2: Create Temporary Arrays

3 let L[1..n1+1] and R[1..n2+1] be new arrays

- **We create two temporary arrays:**
  - **L (Left Array) → stores elements from A[p] to A[q]**
  - **R (Right Array) → stores elements from A[q+1] to A[r]**

🔹 **Why size (n1+1) and (n2+1)?**

- **We add an extra space for a sentinel value (∞), which helps in merging.**

Step 3: Copy Data into Temporary Arrays

```
for i = 1 to n1
    L[i] = A[p + i - 1]
```

- **This copies elements from A[p]A[p] to A[q]A[q] into array L.**

```
for j = 1 to n2
    R[j] = A[q + j]
```

- **This copies elements from A[q+1]A[q+1] to A[r]A[r] into array R.**

🔹 **Why use p+i−1p + i - 1?**

- **Since indexing starts at p, we adjust the index to properly copy elements.**

Step 4: Add Sentinel Values (∞)

```
L[n1+1] = ∞
R[n2+1] = ∞
```

- **We set the last element in both arrays to ∞ (infinity).**
- **This ensures that when merging, we don't go out of bounds.**

🔹 **Why do we use ∞∞?**

- **When one array is fully processed, the other can still contribute.**
- **∞∞ ensures the remaining elements are always smaller.**

Step 5: Merge the Two Arrays Back

```
i = 1
j = 1
```

- **Initialize two pointers:**
  - **i → Tracks position in L.**
  - **j → Tracks position in R.**

```
for k = p to r
```

- **Iterate over the range pp to rr in the original array AA.**

Step 6: Compare and Merge

```
if L[i] ≤ R[j]
    A[k] = L[i]
    i = i + 1
```

- **If the current element in L is smaller, we take it.**
- **Move the pointer i in L.**

```
else
    A[k] = R[j]
    j = j + 1
```

- **If the current element in R is smaller, we take it.**
- **Move the pointer j in R.**

🔹 **Why use ≤ instead of <?**

- **This ensures stability, meaning equal elements keep their order.**

🔹 Example Walkthrough

Let's consider an array:

A=[3,5,8,4,7,9]A = [3, 5, 8, 4, 7, 9]

We want to merge A[1:3] and A[4:6].

Left Array (L)

L = [3, 5, 8, ∞]

1.

Right Array (R)

R = [4, 7, 9, ∞]

2.

Merging Process

k	L[i]	R[j]	A[k]	i or j updated
1	3	4	3	i = 2
2	5	4	4	j = 2
3	5	7	5	i = 3
4	8	7	7	j = 3
5	8	9	8	i = 4
6	∞	9	9	j = 4

Final merged array:

A=[3,4,5,7,8,9]A = [3, 4, 5, 7, 8, 9]

🔹 Complexity Analysis

Time Complexity

- **Copying elements into L and R → O(n)O(n)**

- **Merging elements back into A → O(n)O(n)**
- **Total → O(n)O(n)**

Space Complexity

- **We use extra O(n) space for L and R.**

🔹 Final Notes

- **Merge sort is stable because equal elements maintain their order.**
- **Time complexity is O(n log n) because we repeatedly merge subarrays.**

🚀 Would you like a Python or C implementation? 🚀