Merge Sort Algorithm

A Detailed Analysis of Splitting and Merging Phases

Algorithm Walkthrough

Introduction

Merge Sort is a classic ”Divide and Conquer” sorting algorithm. It works by dividing an unsorted array into smaller subarrays until each subarray contains only a single element (which is inherently sorted). After the splitting is complete, it merges these small arrays back together in sorted order.

As requested, this document traces the algorithm step-by-step.

Part 1: The Splitting Phase Pseudocode

```
Function splitarray(arr, left, right)
    // 1. Base Case: Check if the array can be divided further
    if left >= right
        Return

    // 2. Find the middle index to split the array
    mid = floor( (left + right) / 2 )

    // 3. Recursively split the Left part
    Call splitarray(arr, left, mid)

    // 4. Recursively split the Right part
    Call splitarray(arr, mid + 1, right)

    // 5. Merge the sorted parts (See Part 2)
    Call mergeParts(arr, left, mid, right)

End Function
```

Line-by-Line Explanation of the Pseudocode

Before we trace the array, let’s understand exactly what each line of the algorithm does:

- **`Function splitarray(arr, left, right)`**: This defines our function. It takes three inputs: the array (`arr`), the starting index of the current portion we are looking at (`left`), and the ending index of the current portion (`right`).
- **`if left >= right then Return`**: This is the ”Base Case.” It stops the recursion. If `left` is equal to `right`, it means our current subarray has only 1 element. If `left` is greater than `right`, it has 0 elements. A single-element array cannot be split anymore, so we `Return` (stop splitting and go back).
- **`mid = floor( (left + right) / 2 )`**: This calculates the exact middle point of our current array boundaries. We add the `left` and `right` indices and divide by 2. We use *integer division* (floor), meaning if we get 3.5, we round down to 3.

- **`Call splitarray(arr, left, mid)`**: This is a recursive call. We are telling the computer, ”Now, run this exact same splitting process, but only focus on the left half of the current array (from the `left` index up to the `mid` index).”
- **`Call splitarray(arr, mid + 1, right)`**: This is the second recursive call. After the entire left half is fully broken down, we tell the computer, ”Now, run the splitting process on the right half of the array (from the index just after `mid` up to the `right` index).”

The Problem Setup

We are given an array of 7 elements. The indices (positions) range from 0 to 6.

Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6
8	3	5	2	9	1	7

The algorithm will process these operations in a **Depth-First order**. This means it will keep splitting the left side over and over again until it hits single elements, before it ever looks at the right side! Let's trace this step-by-step.

Step 1: The Initial Call

Action: `splitarray(arr, 0, 6)`
Values: $left = 0$, $right = 6$

- **Condition Check:** Is $0 \geq 6$? No. So, we continue.
- **Calculate Mid:** $mid = (0 + 6)/2 = 3$.
- **What happens next?** The array is logically divided into two parts:
  - Left part (Indices 0 to 3): $[8, 3, 5, 2]$
  - Right part (Indices 4 to 6): $[9, 1, 7]$
- The algorithm now pauses the current step and executes the command to split the left part.

Step 2: Splitting the First Left Part

Action: `splitarray(arr, 0, 3)`
Values: $left = 0$, $right = 3$

- **Condition Check:** Is $0 \geq 3$? No.
- **Calculate Mid:** $mid = (0 + 3)/2 = 1.5$. Rounding down, $mid = 1$.
- **What happens next?** This subarray is divided into two smaller parts:
  - Left part (Indices 0 to 1): $[8, 3]$
  - Right part (Indices 2 to 3): $[5, 2]$
- Again, the algorithm focuses entirely on splitting the newly created left part.

Step 3: Splitting the Left-most Pair

Action: `splitarray(arr, 0, 1)`
Values: $left = 0$, $right = 1$

- **Condition Check:** Is $0 \geq 1$? No.
- **Calculate Mid:** $mid = (0 + 1)/2 = 0.5$. Rounding down, $mid = 0$.
- **What happens next?** This subarray is divided into:
  - Left part (Index 0 to 0): $[8]$
  - Right part (Index 1 to 1): $[3]$
- The algorithm goes to split the left part first.

Step 4: Hitting the Base Cases (Left-most elements)

Action A: `splitarray(arr, 0, 0)`
Values: $left = 0$, $right = 0$

- **Condition Check:** Is $0 \geq 0$? **Yes!** We return immediately. The element $[8]$ is successfully isolated.

Action B: The computer finishes Action A, and now processes the right half of Step 3: `splitarray(arr, 1, 1)`
Values: $left = 1$, $right = 1$

- **Condition Check:** Is $1 \geq 1$? **Yes!** We return immediately. The element $[3]$ is successfully isolated.

Step 5: Backtracking to the Right Side of Step 2

The computer has finished breaking down $[8, 3]$. It now remembers that in Step 2, it still needs to process the right half, which is the array $[5, 2]$.
Action: `splitarray(arr, 2, 3)`
Values: $left = 2$, $right = 3$

- **Condition Check:** Is $2 \geq 3$? No.
- **Calculate Mid:** $mid = (2 + 3)/2 = 2.5$. Rounding down, $mid = 2$.
- **What happens next?** It divides into:
  - Left part (Index 2 to 2): $[5] \rightarrow$ Hits base case and returns!
  - Right part (Index 3 to 3): $[2] \rightarrow$ Hits base case and returns!
- The entire left half of the original main array is now completely broken down into individual elements: $[8]$, $[3]$, $[5]$, and $[2]$.

Step 6: Processing the Original Right Half

Now that the entire left side of the main array (0 to 3) is processed, the computer finally moves to the right side from Step 1.
Action: `splitarray(arr, 4, 6)`
Values: $left = 4$, $right = 6$

- **Condition Check:** Is $4 \geq 6$? No.
- **Calculate Mid:** $mid = (4 + 6)/2 = 5$.
- **What happens next?** The array $[9, 1, 7]$ is divided into:
  - Left part (Indices 4 to 5): $[9, 1]$
  - Right part (Indices 6 to 6): $[7]$

Step 7: Splitting the Final Pair

Action: `splitarray(arr, 4, 5)`
Values: $left = 4$, $right = 5$

- **Condition Check:** Is $4 \geq 5$? No.
- **Calculate Mid:** $mid = (4+5)/2 = 4.5$. Rounding down, $mid = 4$.
- **What happens next?** It divides into:
  - Left part (Index 4 to 4): $[9] \rightarrow$ Hits base case and returns!
  - Right part (Index 5 to 5): $[1] \rightarrow$ Hits base case and returns!

Step 8: The Final Base Case

The computer finishes Step 7 and looks at the right part from Step 6.
Action: `splitarray(arr, 6, 6)`
Values: $left = 6$, $right = 6$

- **Condition Check:** Is $6 \geq 6$? **Yes!** We return immediately. The element $[7]$ is successfully isolated.

Divide Phase Complete!

All elements have now successfully triggered the `left >= right` base case. The array has been entirely shattered into single, 1-element subarrays.

Visual Summary of the Splitting Tree

While the computer processes the split in a depth-first manner (as detailed in the steps above), mathematically, the splitting tree looks like this level by level:

Level 0: The Full Array

8	3	5	2	9	1	7

$\Downarrow$ *Split at mid = 3* $\Downarrow$

Level 1: Two Halves

$\boxed{8\ \vert\ 3\ \vert\ 5\ \vert\ 2} \qquad \boxed{9\ \vert\ 1\ \vert\ 7}$

$\Downarrow$ *Split at mid = 1 and mid = 5* $\Downarrow$

Level 2: Pairs and Triplets broken down

$\boxed{8\ \vert\ 3} \quad \boxed{5\ \vert\ 2} \qquad \boxed{9\ \vert\ 1} \quad \boxed{7}$

$\Downarrow$ *Split at mid = 0, mid = 2, mid = 4* $\Downarrow$

Level 3: Single Elements (Base Cases Reached)

$\boxed{8} \ \boxed{3} \quad \boxed{5} \ \boxed{2} \qquad \boxed{9} \ \boxed{1} \quad \boxed{7}$

Part 2: The Merge Phase (Conquering)

Once the recursive calls hit the base cases (single elements), they return to where they were called. As they return, they execute the next line of code: `Call mergeParts`. This is where the sorting actually happens.

Pseudocode for Merging

```
Function mergeParts(arr, left, mid, right)
    // 1. Calculate sizes of two subarrays to be merged
    n1 = mid - left + 1    // Size of Left Subarray
    n2 = right - mid       // Size of Right Subarray

    // 2. Create temporary arrays
    Create Array L[n1]
    Create Array R[n2]

    // 3. Copy data to temporary arrays
    For i = 0 to n1-1:
        L[i] = arr[left + i]
    For j = 0 to n2-1:
        R[j] = arr[mid + 1 + j]

    // 4. Merge the temp arrays back into arr[left..right]
    i = 0   // Initial index of first subarray (L)
    j = 0   // Initial index of second subarray (R)
    k = left // Initial index of merged subarray (arr)

    While i < n1 AND j < n2
        If L[i] <= R[j]
            arr[k] = L[i]
            i = i + 1
        Else
            arr[k] = R[j]
            j = j + 1
        k = k + 1

    // 5. Copy the remaining elements of L[], if any
    While i < n1
        arr[k] = L[i]
        i = i + 1
        k = k + 1

    // 6. Copy the remaining elements of R[], if any
    While j < n2
        arr[k] = R[j]
        j = j + 1
        k = k + 1
End Function
```

Line-by-Line Explanation of the Merge Logic

- **Calculate Sizes (Step 1):** We determine how big the two parts are. `n1` is the size of the left part (from `left` to `mid`), and `n2` is the size of the right part (from `mid+1` to `right`).
- **Create Temp Arrays (Step 2 & 3):** We create two temporary ”buckets” named `L` (Left) and `R` (Right). We copy the values from the main `arr` into these buckets. `L` gets the values from the left side, and `R` gets the values from the right side.
- **Compare and Merge (Step 4):** This is the heart of the sort. We use two pointers, `i` for bucket `L` and `j` for bucket `R`.
  - We compare `L[i]` with `R[j]`.
  - If `L[i]` is smaller (or equal), we place `L[i]` into the main array and move the `i` pointer forward.
  - If `R[j]` is smaller, we place `R[j]` into the main array and move the `j` pointer forward.
  - We repeat this until one of the buckets is empty.
- **Copy Remaining Elements (Steps 5 & 6):** Once one bucket is empty, the other bucket might still have elements left. Since the subarrays were already sorted individually, we can safely copy any remaining elements directly into the main array.

Step-by-Step Execution of the Merge Phase

The merging happens as we ”return” from the splits. We will merge in the exact reverse order of the splits (backtracking up the tree).

Step 9: Merging the First Pair (Indices 0 and 1)

We are returning from splitting indices 0 and 1 (from Step 3).
Action: `mergeParts(arr, left=0, mid=0, right=1)`

- **Temp Arrays:** $L = [8]$, $R = [3]$
- **Comparison:** Compare 8 and 3. Is $8 \leq 3$? No.
- **Action:** Take 3 from R. Place in `arr[0]`. Increment j.
- **Remaining:** R is empty. Take 8 from L. Place in `arr[1]`.
- **Result:** Subarray 0-1 is now sorted: $[3, 8]$.

Step 10: Merging the Second Pair (Indices 2 and 3)

We are returning from splitting indices 2 and 3 (from Step 5).
Action: `mergeParts(arr, left=2, mid=2, right=3)`

- **Temp Arrays:** $L = [5]$, $R = [2]$
- **Comparison:** Compare 5 and 2. Is $5 \leq 2$? No.
- **Action:** Take 2 from R. Place in `arr[2]`. Increment j.
- **Remaining:** R is empty. Take 5 from L. Place in `arr[3]`.
- **Result:** Subarray 2-3 is now sorted: $[2, 5]$.

Step 11: Merging the Left Half (Indices 0 to 3)

Now that pairs $[3, 8]$ and $[2, 5]$ are sorted, we merge them together (Backtracking Step 2).
Action: `mergeParts(arr, left=0, mid=1, right=3)`

- **Temp Arrays:** $L = [3, 8]$, $R = [2, 5]$
- **Pass 1:** Compare L’s 3 vs R’s 2. ($2 < 3$). Place 2 at `arr[0]`. (R moves to 5)
- **Pass 2:** Compare L’s 3 vs R’s 5. ($3 < 5$). Place 3 at `arr[1]`. (L moves to 8)
- **Pass 3:** Compare L’s 8 vs R’s 5. ($5 < 8$). Place 5 at `arr[2]`. (R is empty)
- **Remaining:** R is empty. Copy remaining 8 from L to `arr[3]`.
- **Result:** Indices 0-3 are now sorted: $[2, 3, 5, 8]$.

Step 12: Merging the Right Pair (Indices 4 and 5)

We switch to the right side of the main array (from Step 7).
Action: `mergeParts(arr, left=4, mid=4, right=5)`

- **Temp Arrays:** $L = [9]$, $R = [1]$
- **Comparison:** 1 is smaller than 9.
- **Result:** Subarray 4-5 is now sorted: $[1, 9]$.

Step 13: Merging the Right Half (Indices 4 to 6)

We merge the pair $[1, 9]$ with the single element $[7]$ (Backtracking Step 6).
Action: `mergeParts(arr, left=4, mid=5, right=6)`

- **Temp Arrays:** $L = [1, 9]$, $R = [7]$
- **Pass 1:** Compare 1 vs 7. ($1 < 7$). Place 1 at `arr[4]`. (L moves to 9)
- **Pass 2:** Compare 9 vs 7. ($7 < 9$). Place 7 at `arr[5]`. (R is empty)
- **Pass 3:** R is empty. Copy remaining 9 from L to `arr[6]`.
- **Result:** Indices 4-6 are now sorted: $[1, 7, 9]$.

Step 14: The Final Merge (Indices 0 to 6)

Finally, we merge the sorted left half $[2, 3, 5, 8]$ with the sorted right half $[1, 7, 9]$ (Backtracking Step 1).
Action: `mergeParts(arr, left=0, mid=3, right=6)`

- **Temp Arrays:** $L = [2, 3, 5, 8]$, $R = [1, 7, 9]$
- **Comparison 1:** 2 vs $1 \rightarrow$ Take 1 (from R).
- **Comparison 2:** 2 vs $7 \rightarrow$ Take 2 (from L).
- **Comparison 3:** 3 vs $7 \rightarrow$ Take 3 (from L).
- **Comparison 4:** 5 vs $7 \rightarrow$ Take 5 (from L).
- **Comparison 5:** 8 vs $7 \rightarrow$ Take 7 (from R).
- **Comparison 6:** 8 vs $9 \rightarrow$ Take 8 (from L).
- **Remaining:** L is empty. Copy remaining 9 from R to the end.

Final Sorted Array:

[1, 2, 3, 5, 7, 8, 9]