Tracing Merge Sort

Let's trace the Merge Sort algorithm step by step with a concrete example. We'll use the array:

```
csharp
Copy
[38, 27, 43, 3, 9, 82, 10]
```

Our goal is to sort this array in ascending order using the following two methods:

- **mergeSort(arr, left, right)**
- **merge(arr, left, mid, right)**

We'll assume the array indices go from 0 to 6 (since there are 7 elements). I'll explain every step, including how temporary arrays are created and merged back.

---

1. Initial Call to mergeSort

We call:

```
java
Copy
mergeSort(arr, 0, 6)
```

- **left = 0**, **right = 6**
- **mid = (0 + 6) / 2 = 3**

Now, the array is divided into two halves:

- **Left Half:** indices 0 to 3 → `[38, 27, 43, 3]`
- **Right Half:** indices 4 to 6 → `[9, 82, 10]`

The algorithm recursively sorts each half.

---

2. Sorting the Left Half: mergeSort(arr, 0, 3)

2.1 Call: mergeSort(arr, 0, 3)

- **left = 0**, **right = 3**
- **mid = (0 + 3) / 2 = 1**

Now, split into:

- **mergeSort(arr, 0, 1)**
- **mergeSort(arr, 2, 3)**

2.2 Sorting mergeSort(arr, 0, 1)

2.2.1 Call: mergeSort(arr, 0, 1)

- **left = 0**, **right = 1**
- **mid = (0 + 1) / 2 = 0**

Split into two calls:

- **mergeSort(arr, 0, 0)**
- **mergeSort(arr, 1, 1)**

Both calls hit the base case (when left equals right) and return immediately.
So, we now merge the subarray from indices 0 to 1.

2.2.2 Call: merge(arr, 0, 0, 1)

In merge():

- **left = 0, mid = 0, right = 1**
- Calculate sizes:
  - `n1 = mid - left + 1 = 0 - 0 + 1 = 1`
  - `n2 = right - mid = 1 - 0 = 1`
- Create temporary arrays:
  - `L` of size 1, `R` of size 1.
- Copy data:
  - For L:
    - `L[0] = arr[0] = 38`
  - For R:
    - `R[0] = arr[0+1] = arr[1] = 27`

Merging:

- Initialize indices: `i = 0, j = 0, k = left (0)`
- Compare `L[0] (38)` and `R[0] (27)`:
  - Since 38 > 27, place 27 in `arr[0]`.
  - Increment `j` (now 1) and `k` (now 1).
- Now `j == n2` (1 == 1), so exit the main while loop.
- Copy remaining element from L:
  - `arr[1] = L[0] = 38`

After merging, the subarray at indices 0 to 1 becomes:

```
csharp
Copy
[27, 38]
```

2.3 Sorting mergeSort(arr, 2, 3)

2.3.1 Call: mergeSort(arr, 2, 3)

- **left = 2, right = 3**
- **mid = (2 + 3) / 2 = 2**

Split into:

- **mergeSort(arr, 2, 2)**
- **mergeSort(arr, 3, 3)**

Both are base cases and return immediately.

2.3.2 Call: merge(arr, 2, 2, 3)

In merge():

- **left = 2, mid = 2, right = 3**
- Calculate sizes:
  - `n1 = 2 - 2 + 1 = 1`
  - `n2 = 3 - 2 = 1`
- Create temporary arrays:
  - `L[0] = arr[2] = 43`
  - `R[0] = arr[3] = 3`

Merging:

- Initialize indices: `i = 0, j = 0, k = 2`
- Compare `L[0] (43)` and `R[0] (3)`:
  - 43 > 3, so set `arr[2] = 3`.
  - Increment `j` (now 1) and `k` (now 3).
- Now `j == n2`, exit the loop.
- Copy remaining from L:
  - `arr[3] = L[0] = 43`

After merging, subarray at indices 2 to 3 becomes:

```
csharp
Copy
[3, 43]
```

2.4 Merging the Two Sorted Halves of Left Part

Now we have two sorted parts:

- From indices 0 to 1: `[27, 38]`
- From indices 2 to 3: `[3, 43]`

2.4.1 Call: merge(arr, 0, 1, 3)

In merge():

- **left = 0, mid = 1, right = 3**
- Calculate sizes:
  - `n1 = 1 - 0 + 1 = 2`
  - `n2 = 3 - 1 = 2`
- Create temporary arrays:
  - Copy into L:
    - `L[0] = arr[0] = 27`
    - `L[1] = arr[1] = 38`
  - Copy into R:
    - `R[0] = arr[1+1] = arr[2] = 3`
    - `R[1] = arr[1+2] = arr[3] = 43`

Merging:

- Initialize indices: `i = 0, j = 0, k = 0`
- **Iteration 1:**
  - Compare `L[0] = 27` and `R[0] = 3`
  - Since 27 > 3, set `arr[0] = 3`
  - Increment `j` (now 1) and `k` (now 1)
- **Iteration 2:**
  - Compare `L[0] = 27` and `R[1] = 43`
  - Since 27 ≤ 43, set `arr[1] = 27`
  - Increment `i` (now 1) and `k` (now 2)
- **Iteration 3:**
  - Compare `L[1] = 38` and `R[1] = 43`
  - Since 38 ≤ 43, set `arr[2] = 38`
  - Increment `i` (now 2) and `k` (now 3)
- Now `i == n1` (2 == 2), so copy the remaining element from R:
  - `arr[3] = R[1] = 43`

After this merge, the left half (indices 0 to 3) is sorted as:

```
csharp
Copy
```

```
[3, 27, 38, 43]
```

---

3. Sorting the Right Half: mergeSort(arr, 4, 6)

3.1 Call: mergeSort(arr, 4, 6)

- **left = 4**, **right = 6**
- **mid = (4 + 6) / 2 = 5**

Split into:

- **mergeSort(arr, 4, 5)**
- **mergeSort(arr, 6, 6)**

3.2 Sorting mergeSort(arr, 4, 5)

3.2.1 Call: mergeSort(arr, 4, 5)

- **left = 4**, **right = 5**
- **mid = (4 + 5) / 2 = 4**

Split into:

- **mergeSort(arr, 4, 4)** → Base case (single element `[9]`), returns.
- **mergeSort(arr, 5, 5)** → Base case (single element `[82]`), returns.

3.2.2 Call: merge(arr, 4, 4, 5)

In merge():

- **left = 4, mid = 4, right = 5**
- Calculate sizes:
  - `n1 = 4 - 4 + 1 = 1`
  - `n2 = 5 - 4 = 1`
- Create temporary arrays:
  - `L[0] = arr[4] = 9`
  - `R[0] = arr[5] = 82`

Merging:

- Initialize indices: `i = 0, j = 0, k = 4`
- Compare `L[0] (9)` with `R[0] (82)`
  - Since 9 ≤ 82, set `arr[4] = 9`

    - Increment `i` (now 1) and `k` (now 5)
- `i == n1`, so copy remaining from R:
    - `arr[5] = R[0] = 82`

After merging, subarray indices 4 to 5 become:

```
csharp
Copy
[9, 82]
```

3.3 Sorting mergeSort(arr, 6, 6)

- **left = 6**, **right = 6**
- This is a base case; the subarray is just `[10]` and is already sorted.

3.4 Merging the Two Sorted Parts of the Right Half

Now we merge the subarrays:

- Left subarray: indices 4 to 5 → `[9, 82]`
- Right subarray: index 6 → `[10]`

3.4.1 Call: merge(arr, 4, 5, 6)

In merge():

- **left = 4, mid = 5, right = 6**
- Calculate sizes:
    - `n1 = 5 - 4 + 1 = 2`
    - `n2 = 6 - 5 = 1`
- Create temporary arrays:
    - For L:
        - `L[0] = arr[4] = 9`
        - `L[1] = arr[5] = 82`
    - For R:
        - `R[0] = arr[5 + 1] = arr[6] = 10`

Merging:

- Initialize indices: `i = 0, j = 0, k = 4`
- Compare `L[0] (9)` with `R[0] (10)`
    - Since 9 ≤ 10, set `arr[4] = 9`
    - Increment `i` (now 1) and `k` (now 5)
- Compare `L[1] (82)` with `R[0] (10)`

    - Since 82 > 10, set `arr[5] = 10`
    - Increment `j` (now 1) and `k` (now 6)
- Now `j == n2` (1 == 1), so copy the remaining element from L:
    - `arr[6] = L[1] = 82`

After merging, the right half (indices 4 to 6) becomes:

```
csharp
Copy
[9, 10, 82]
```

---

4. Final Merge of the Left and Right Halves

At this point, we have two sorted halves:

- **Left half:** indices 0 to 3 → `[3, 27, 38, 43]`
- **Right half:** indices 4 to 6 → `[9, 10, 82]`

4.1 Call: merge(arr, 0, 3, 6)

In merge():

- **left = 0, mid = 3, right = 6**
- Calculate sizes:
    - `n1 = 3 - 0 + 1 = 4`
    - `n2 = 6 - 3 = 3`
- Create temporary arrays:
    - Copy into L:
        - `L[0] = arr[0] = 3`
        - `L[1] = arr[1] = 27`
        - `L[2] = arr[2] = 38`
        - `L[3] = arr[3] = 43`
    - Copy into R:
        - `R[0] = arr[3 + 1] = arr[4] = 9`
        - `R[1] = arr[3 + 2] = arr[5] = 10`
        - `R[2] = arr[3 + 3] = arr[6] = 82`

Merging:

- Initialize indices: `i = 0, j = 0, k = 0`
- **Iteration 1:**
    - Compare L[0] = 3 and R[0] = 9
    - 3 ≤ 9, so set `arr[0] = 3`
    - Increment `i` to 1, `k` to 1.
- **Iteration 2:**
    - Compare L[1] = 27 and R[0] = 9
    - 27 > 9, so set `arr[1] = 9`

    - Increment `j` to 1, `k` to 2.
- **Iteration 3:**
    - Compare L[1] = 27 and R[1] = 10
    - 27 > 10, so set `arr[2] = 10`
    - Increment `j` to 2, `k` to 3.
- **Iteration 4:**
    - Compare L[1] = 27 and R[2] = 82
    - 27 ≤ 82, so set `arr[3] = 27`
    - Increment `i` to 2, `k` to 4.
- **Iteration 5:**
    - Compare L[2] = 38 and R[2] = 82
    - 38 ≤ 82, so set `arr[4] = 38`
    - Increment `i` to 3, `k` to 5.
- **Iteration 6:**
    - Compare L[3] = 43 and R[2] = 82
    - 43 ≤ 82, so set `arr[5] = 43`
    - Increment `i` to 4, `k` to 6.
- Now `i == n1` (4 == 4), so we exit the main loop.
- Copy any remaining elements from R:
    - `arr[6] = R[2] = 82`
    - Increment `j` to 3, `k` to 7.

After this final merge, the entire array becomes:

```
csharp
Copy
[3, 9, 10, 27, 38, 43, 82]
```

Final Outcome

The original array

```
csharp
Copy
[38, 27, 43, 3, 9, 82, 10]
```

has been completely sorted into:

```
csharp
Copy
[3, 9, 10, 27, 38, 43, 82]
```

Tracing QuickSort

Let's trace the Quick Sort program step by step using the example array:

```
csharp
Copy
[10, 7, 8, 9, 1, 5]
```

We'll follow each recursive call and partition call carefully.

Overview

- **Quick Sort** works by choosing a **pivot** and then **partitioning** the array so that:
  - All elements less than the pivot are on its left.
  - All elements greater than the pivot are on its right.
- The algorithm then recursively sorts the subarrays on each side of the pivot.

In our implementation, we choose the **last element** as the pivot.

Initial Call

We start with:

```
java
Copy
quickSort(arr, 0, 5)
```

- **Array:** `[10, 7, 8, 9, 1, 5]`
- **low = 0, high = 5**

Step 1: Partitioning (Call: partition(arr, 0, 5))

1. **Choose Pivot:**
   - Pivot = `arr[5] = 5`
2. **Initialize:**
   - Set `i = low - 1 = -1`
3. **Loop through the array from index `j = 0` to `4`:**
   - **j = 0:**
     - Current element = `arr[0] = 10`
     - Compare: Is `10 < 5`? → **No**
     - So, do nothing (i remains -1).

- **j = 1:**
    - Current element = `arr[1] = 7`
    - Compare: Is `7 < 5`? → **No**
    - Do nothing.
- **j = 2:**
    - Current element = `arr[2] = 8`
    - Compare: Is `8 < 5`? → **No**
    - Do nothing.
- **j = 3:**
    - Current element = `arr[3] = 9`
    - Compare: Is `9 < 5`? → **No**
    - Do nothing.
- **j = 4:**
    - Current element = `arr[4] = 1`
    - Compare: Is `1 < 5`? → **Yes**
    - Increment `i`: now `i = 0`
    - Swap `arr[i]` with `arr[j]`:
        - Swap `arr[0]` (10) with `arr[4]` (1)
        - **Array becomes:** `[1, 7, 8, 9, 10, 5]`

4. **After the loop, place the pivot in its correct position:**
    - The correct pivot position is `i + 1 = 1`
    - Swap `arr[i + 1]` with `arr[high]`:
        - Swap `arr[1]` (7) with `arr[5]` (5)
    - **Array becomes:** `[1, 5, 8, 9, 10, 7]`
5. **Return the pivot index:**
    - Pivot index = `1`

After First Partition

- **Array:** `[1, 5, 8, 9, 10, 7]`
- The pivot (5) is now at index 1.
- **Left subarray:** indices 0 to 0 → `[1]`
- **Right subarray:** indices 2 to 5 → `[8, 9, 10, 7]`

Now we recursively sort the left and right subarrays.

Step 2: Recursive Calls on Subarrays

2.1 Left Subarray: quickSort(arr, 0, 0)

- **Call:** `quickSort(arr, 0, 0)`

- **Condition:** Since `low == high` (0 == 0), the subarray has one element `[1]` and is already sorted.
- **Action:** Return immediately.

2.2 Right Subarray: quickSort(arr, 2, 5)

- **Call:** `quickSort(arr, 2, 5)`
- **Subarray:** `[8, 9, 10, 7]`
- **low = 2, high = 5**

Step 2.2.1: Partition the Right Subarray

Call: `partition(arr, 2, 5)`

1. **Choose Pivot:**
   - Pivot = `arr[5] = 7`
2. **Initialize:**
   - Set `i = low - 1 = 1`
3. **Loop from `j = 2` to `4`:**
   - **j = 2:**
     - `arr[2] = 8`
     - Is `8 < 7`? → **No**
     - Do nothing.
   - **j = 3:**
     - `arr[3] = 9`
     - Is `9 < 7`? → **No**
   - **j = 4:**
     - `arr[4] = 10`
     - Is `10 < 7`? → **No**
4. **No element in this range is less than the pivot.**
   - `i` remains 1.
5. **Place the pivot in its correct position:**
   - Correct position is `i + 1 = 2`
   - Swap `arr[2]` and `arr[5]`:
     - Swap `arr[2]` (8) with `arr[5]` (7)
   - **Array becomes:** `[1, 5, 7, 9, 10, 8]`
6. **Return the pivot index:**
   - Pivot index = `2`

Step 2.2.2: After Partition of Right Subarray

- **Subarray now:** `[7, 9, 10, 8]` with pivot 7 at index 2.
- The subarray is divided into:
  - **Left part:** indices 2 to 1 (empty, since 2 > 1)
  - **Right part:** indices 3 to 5 → `[9, 10, 8]`

Now recursively sort the subarrays of `[7, 9, 10, 8]`.

2.2.3: Sort Left Part of Right Subarray: quickSort(arr, 2, 1)

- **Call:** `quickSort(arr, 2, 1)`
- **Condition:** Since `low > high`, return immediately.

2.2.4: Sort Right Part of Right Subarray: quickSort(arr, 3, 5)

- **Call:** `quickSort(arr, 3, 5)`
- **Subarray:** `[9, 10, 8]`
- **low = 3, high = 5**

Partition this Subarray (Call: partition(arr, 3, 5))

1. **Choose Pivot:**
   - Pivot = `arr[5] = 8`
2. **Initialize:**
   - Set `i = low - 1 = 2`
3. **Loop from `j = 3` to `4`:**
   - **j = 3:**
     - `arr[3] = 9`
     - Is `9 < 8`? → **No**
   - **j = 4:**
     - `arr[4] = 10`
     - Is `10 < 8`? → **No**
4. **No elements less than pivot; `i` remains 2.**
5. **Place the pivot in correct position:**
   - Correct position is `i + 1 = 3`
   - Swap `arr[3]` and `arr[5]`:
     - Swap `arr[3]` (9) with `arr[5]` (8)
   - **Array becomes:** `[1, 5, 7, 8, 10, 9]`
6. **Return pivot index:**
   - Pivot index = `3`

After Partition of [9, 10, 8]

- **Subarray now:** `[8, 10, 9]` with pivot 8 at index 3.
- Dividing it:
  - **Left part:** indices 3 to 2 (empty)
  - **Right part:** indices 4 to 5 → `[10, 9]`

Recursive Calls on [8, 10, 9]

- **Sort Left Part:**
  - `quickSort(arr, 3, 2)` → returns immediately.
- **Sort Right Part:**
  - `quickSort(arr, 4, 5)`

Sorting Subarray [10, 9]: quickSort(arr, 4, 5)

1. **Call:** `quickSort(arr, 4, 5)`
   - **low = 4, high = 5**
   - **mid = (4 + 5) / 2 = 4**
2. **Partition (Call: partition(arr, 4, 5)):**
   - **Pivot:** `arr[5] = 9`
   - **Initialize:** `i = low - 1 = 3`
   - Loop from `j = 4` to `4`:
     - **j = 4:**
       - `arr[4] = 10`
       - Is `10 < 9`? → **No**
   - End loop, `i` is still 3.
   - **Place pivot:**
     - Correct position = `i + 1 = 4`
     - Swap `arr[4]` and `arr[5]`:
       - Swap `arr[4]` (10) with `arr[5]` (9)
     - **Array becomes:** `[1, 5, 7, 8, 9, 10]`
   - Return pivot index = `4`
3. **After Partition [10, 9] becomes [9, 10]**
4. **Recursive calls:**
   - `quickSort(arr, 4, 3)` → returns immediately
   - `quickSort(arr, 5, 5)` → returns immediately

Putting It All Together

Now, let's see the full progression of the array after each major partition:

1. **Initial Array:**
   `[10, 7, 8, 9, 1, 5]`
2. **After First Partition (with pivot 5):**
   `[1, 5, 8, 9, 10, 7]`
   Pivot 5 is at index 1.
3. **Sorting the Right Subarray ([8, 9, 10, 7]):**
   - After partition with pivot 7, subarray becomes:
     `[7, 9, 10, 8]` with pivot 7 at index 2.
4. **Sorting subarray [9, 10, 8] (right part of the above):**
   - After partition with pivot 8, subarray becomes:
     `[8, 10, 9]` with pivot 8 at index 3.
5. **Sorting subarray [10, 9]:**
   - After partition with pivot 9, subarray becomes:
     `[9, 10]` with pivot 9 at index 4.
6. **Final Array:**
   `[1, 5, 7, 8, 9, 10]`

Summary of Recursive Calls

1. **quickSort(arr, 0, 5):**
   - Partition → pivot index = 1 (pivot 5).
   - Left side: quickSort(arr, 0, 0) → already sorted.
   - Right side: quickSort(arr, 2, 5).
2. **quickSort(arr, 2, 5):**
   - Partition → pivot index = 2 (pivot 7).
   - Left side: quickSort(arr, 2, 1) → empty, returns.
   - Right side: quickSort(arr, 3, 5).
3. **quickSort(arr, 3, 5):**
   - Partition → pivot index = 3 (pivot 8).
   - Left side: quickSort(arr, 3, 2) → empty.
   - Right side: quickSort(arr, 4, 5).
4. **quickSort(arr, 4, 5):**
   - Partition → pivot index = 4 (pivot 9).
   - Left side: quickSort(arr, 4, 3) → empty.
   - Right side: quickSort(arr, 5, 5) → single element.

Each partition moves the chosen pivot to its correct sorted position, and the recursive calls sort the subarrays to the left and right of that pivot.

Final Sorted Array

After all the recursive calls and partitions complete, the sorted array is:

```
csharp
Copy
[1, 5, 7, 8, 9, 10]
```