Merge Sort

Josh Wilcox (jw14g24@soton.ac.uk)

March 8, 2025

Contents

1 Recursion Algorithm

```
static void MergeSort(array, start, end) {
    if (start < end){
        mid = floor((start + end) / 2) // Middle element index
        MergeSort(array, start, mid) // Left half
        MergeSort(array, mid+1, end) // Right half
        Merge(array, start, mid, end) // Merge the split up arrays
    }
}
```

2 Merging Sub Arrays

- The merge operation takes two sorted sub-arrays and combines them into a single sorted array.
- Assume the two sub-arrays are `left` and `right`.
- Initialize three pointers: `i` for the `left` sub-array, `j` for the `right` sub-array, and `k` for the position in the merged array.
- Compare the elements at `left[i]` and `right[j]`:
  - If `left[i]` is smaller, place `left[i]` in the merged array at position `k` and increment `i` and `k`.
  - Otherwise, place `right[j]` in the merged array at position `k` and increment `j` and `k`.
- Repeat the comparison until one of the sub-arrays is exhausted.
- Copy any remaining elements from the non-exhausted sub-array into the merged array.

```
static void merge(int array[], int startIndex, int middleIndex, int endIndex)
{
    // Step 1: Calculate sizes of the two subarrays to be merged
    int leftSize = middleIndex - startIndex + 1;  // Size of left subarray
    int rightSize = endIndex - middleIndex;       // Size of right subarray

    // Step 2: Create temporary arrays to hold the subarrays
    int leftArray[] = new int[leftSize];
    int rightArray[] = new int[rightSize];

    // Step 3: Copy data from original array to the temporary arrays
    for (int i = 0; i < leftSize; ++i)
        leftArray[i] = array[startIndex + i];     // Copy to left array
    for (int j = 0; j < rightSize; ++j)
        rightArray[j] = array[middleIndex + 1 + j];  // Copy to right array

    // Step 4: Merge the two subarrays back into the original array

    // Initialize pointers for traversal
    int leftIndex = 0;      // Current position in leftArray
    int rightIndex = 0;     // Current position in rightArray
    int mergedIndex = startIndex;  // Current position in merged array
```

```
    // Step 5: Compare and merge elements from both arrays in sorted order
    while (leftIndex < leftSize && rightIndex < rightSize) {
        // Compare current elements from both arrays and place smaller one first
        if (leftArray[leftIndex] <= rightArray[rightIndex]) {
            // Left element is smaller or equal, so place it first
            array[mergedIndex] = leftArray[leftIndex];
            leftIndex++;  // Move to next element in left array
        }
        else {
            // Right element is smaller, so place it first
            array[mergedIndex] = rightArray[rightIndex];
            rightIndex++;  // Move to next element in right array
        }
        mergedIndex++;  // Move to next position in merged array
    }

    // Step 6: Copy any remaining elements from left array (if any)
    while (leftIndex < leftSize) {
        array[mergedIndex] = leftArray[leftIndex];
        leftIndex++;
        mergedIndex++;
    }

    // Step 7: Copy any remaining elements from right array (if any)
    while (rightIndex < rightSize) {
        array[mergedIndex] = rightArray[rightIndex];
        rightIndex++;
        mergedIndex++;
    }
    // At this point, the subarrays are fully merged in sorted order
}
```

3 Properties of Merge Sort

- It is **stable**
- It is **not** *in-place*
- Merging is quick
  - At most $n - 1$ comparisons to merge two subarrays
- Recurrence Relation:
$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$$
  - Therefore **worst case** complexity: $\mathcal{O}(nlog(n))$

4 Improving Merge sort with Insertion Sort

- Insertion sort is very efficient for short arrays

- Therefore, if a sub-array size falls below a certain threshold, it is better to switch to insertion sort to sort the array instead of splitting even more
- These can then be merged in the same ray