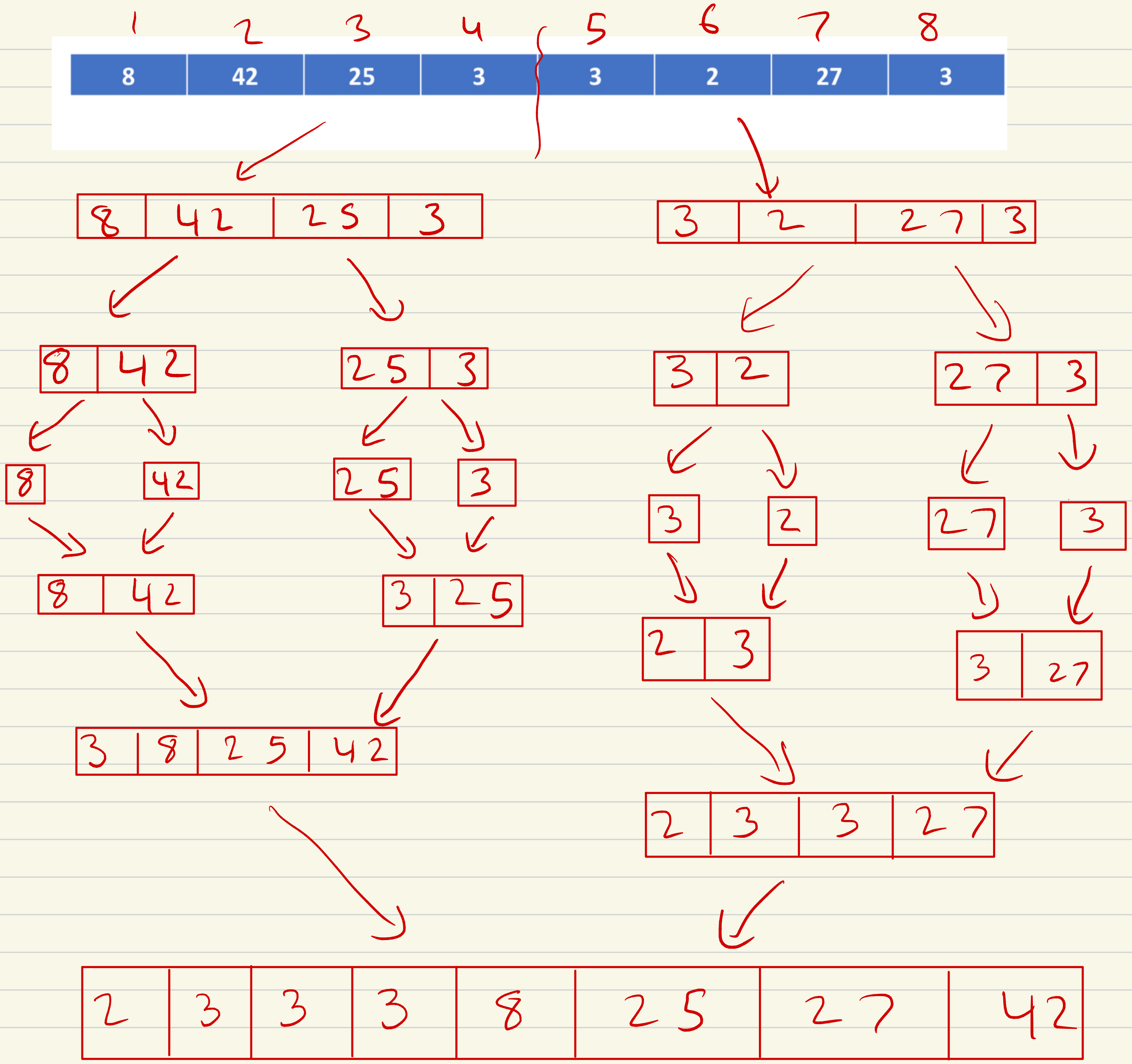
Q2

merge_sort function:
The function divides the array into halves recursively until each subarray contains only one element.
This operation takes O(log n) time because the array is divided into halves in each recursive call until it reaches a size of 1.
Then, the merge operation is called on each pair of subarrays. Since there are log n levels of recursion,
and at each level, each element in the array is merged exactly once, the total time complexity for merging all levels is O(n).

merge function:
The merge function iterates through each element of the two subarrays exactly once, comparing and merging
them into a single sorted array. This process takes O(n) time, where n is the total number of elements in the two subarrays.

3)



The initial array is divided into smaller sub-arrays:

[8 42 25 3] [3 2 27 3]

These are further divided into [8 42] then [8] [42], [25 3] then [25] [3] and

[3 2] then [3] [2] and
[27 3] then [27] [3].
These sub arrays are then sorted individually and merged. After merge they become [3 8 25 42] and [2 3 3 27]. The two sorted sub arrays are then merged this gives us final sorted array [2 3 3 3 8 25 27 42].

Q4)

The divide phase would require at most $\log_2 (8) = 3$ steps to create sub-arrays of size 1 which is consistent with the diagram in Q3. In the worst case scenario, each level of the recursion during the merge phase would require processing all 8 elements at once. Therefore, the total number of steps would be 3 (divide phase) * 8 (merge phase) = 24, which is consistent with the expected time complexity of O (n log n). So, yes, the number of steps aligns with the expected time complexity analysis of Merge Sort. Each element is visited at most log(n) times during the divide phase and n times during the merge phase, leading to an overall time complexity of O (n log n).