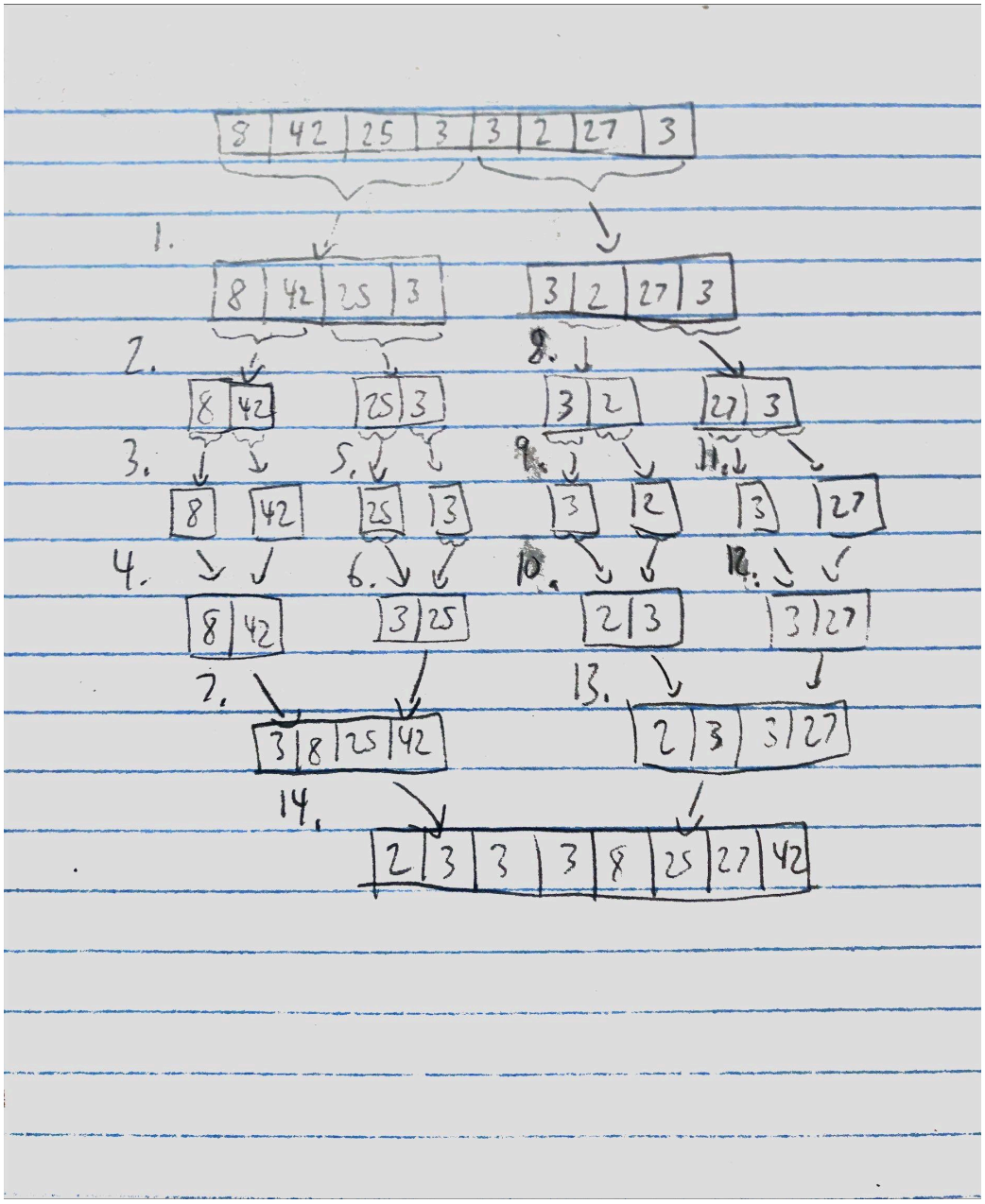
ENSF 338 - Lab 3, Exercise 1

Question 1.2 Answer:
First, we partition the original array into two subarrays: one subarray for the bottom half and one subarray for the top half of the original array. Since we need to copy each element in the array split into a bottom and top half subarray, the time complexity for each half is O(n/2) = O(n).

Next, we have the three while loops. The first while loop picks the larger of the elements among leftArr and rightArr; this is clearly O(n) since we're essentially just going through the entire array at once (also no nested loops!). As for the last two while loops, they both check for any remaining elements of half the entire array (bottom and top) and then copy them into the original array; again this time complexity will be O(n/2)=O(n) for these sections of code. Therefore, the time complexity for the merge() function is O(n).

Finally, we have the merge_sort() function. Every time merge_sort(arr, low, mid) and merge_sort(arr, mid+1, high) are called, we are splitting the arr into two (bottom and top half) which has complexity O(log n). This splitting happens n times, which gives us log-linear complexity i.e O(n log(n)). The last step is the merging using the merge() function which is O(n) as discussed above. Therefore, since O(n log(n)) >> O(n) for large inputs, when we combine the two complexities O(n log(n)) dominates O(n) so the overall complexity of the Merge Sort Algorithm is O(n log(n)).

Question 1.3 Answer:



Question 1.4 Answer:

Yes, the number of steps is consistent with my complexity analysis. We can see that the vector is halved at each step and takes the left half each time. Then after sorting each left half array we merge back with the right half of the vector from the bottom up. Then we repeat the same process but this time on the right half of the vector.