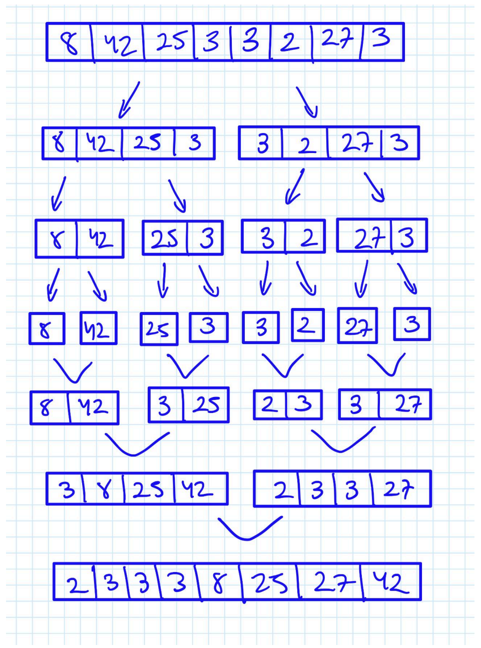
2.
The algorithm has a worst-case complexity of O(nlogn) because the merge_sort() function has a complexity of O(logn) and the merge() function has one of O(n). Since merge_sort divides the subarrays in half recursively using merge_sort(arr, low, mid) and merge_sort(arr, mid+1, high), it results in a time complexity of O(logn). The merge function that is called iterates through the elements of the subarrays once and then merges them. This results in a time complexity of O(n). Since merge is called within each recursive call of merge_sort the overall complexity is the complexity of merge_sort times merge, resulting in O(nlogn).

3.



The merge_sort function first splits the initial array in half, continuing to split the subsequent arrays in half until a single element remains. Then the merge function merges these elements together in order from smallest to greatest. For example, the initial array is split, then the array containing 8,42,25, and 3, then the array with 8 and 42, then we are left with 8 and 42 as single elements so they are merged together, 8 on the left and 42 on the right. We then go back to when the array containing 8,42,25, and 3 was split and repeat the process for 25,3. It is sorted and merged(using merge()), producing 3,8,25,42. This is repeated for the other half of the initial array until everything is merged.

4. Is the number of steps consistent with your complexity analysis?
Justify your answer. [0.2 pts]

Yes, the number of steps is consistent with the time complexity analysis. This is because as the number of elements in the array increases the steps do proportionally to nlogn, n being the number of elements.