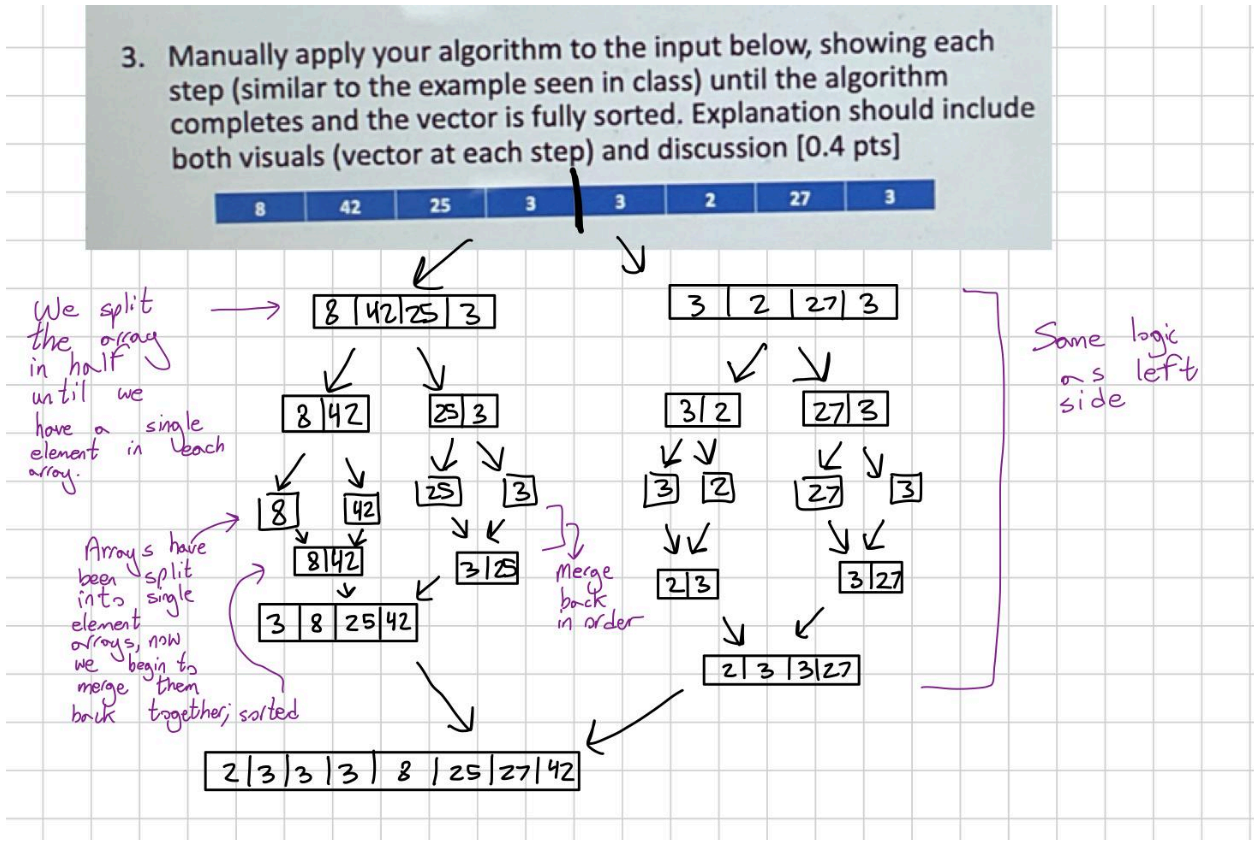
2. There are two parts to the merge sort algorithm that cause it to have a complexity of O(nlog(n)). The merge_sort function in the program recursively splits the array in half at each step until there each array has only one element left regardless of how the array is organized since the base case can only be contradicted when low < high, which cannot be achieved until they both have the same value. By low and high arguments being used as indexes, we created a mid variable by dividing the two, and then we used all three of those arguments along with an array to split the array and recursively call the merge function. Since we are splitting the array by half here each time, the complexity is there by O(log(n)) for the merge_sort function. Now for the merge function that is being recursively called, we find that the function splits the incoming array in two. The merge function is required to iterate throughout both the arrays to create temporary arrays, and then iterate with while loops later on to merge the arrays back in the correct order from low to high. Since we are required to iterate through all the elements in the array regardless of how the array was originally sorted for the sake of comparison, it is clear that the complexity of the merge function is O(n).

Since the merge_sort function recursively calls the merge function with a complexity of O(log(n)) times and each merge function call itself has a complexity of O(n), then the overall complexity becomes:

O(n) x O(log(n) = O(nlog(n))

3.



3. Manually apply your algorithm to the input below, showing each step (similar to the example seen in class) until the algorithm completes and the vector is fully sorted. Explanation should include both visuals (vector at each step) and discussion [0.4 pts]

| 8 | 42 | 25 | 3 | 3 | 2 | 27 | 3 |
|---|---|---|---|---|---|---|---|

We split the array in half until we have a single element in each array.
Same logic as left side
Arrays have been split into single element arrays, now we begin to merge them back together, sorted
Merge back in order
8 | 42 | 25 | 3
3 | 2 | 27 | 3
8 | 42
25 | 3
3 | 2
27 | 3
3 | 8 | 25 | 42
2 | 3 | 3 | 27
2 | 3 | 3 | 3 | 8 | 25 | 27 | 42

4. log2(n=8) = 3. n= 8.
nlog(n) = 8 x 3 = 24. Complexity is O(24)

^^The above is the mathematical number of steps that it would take to completely sort the array using the merge sort algorithm. Counting each step in the above image, we have 1 operation in the first split, 2 operations once we split the already split arrays, 4 operations once we split each of those arrays. Now for merging, we have 4 operations in the first round of merging. In the second round of merging, it begins to be a little more complicated as there are now going to be 3 comparisons and thereby 3 operations for each merged array, resulting in 6 operations. In the final merging increment in the above image, in order to correctly sort everything and merge, we'd have to make 7 comparisons, and thereby 7 operations. In a total count at each step, we find that we have 1 + 2 + 4 + 4 + 6 + 7 = 24 operations, which is consistent with the determined complexity we have calculated.