2. Argue that the overall algorithm has a worst-case complexity of O(nlogn). Note, your description must specifically refer to the code you wrote, i.e., not just generically talk about mergesort.

The merge function is O(n), since, the way I implemented it, it linearly iterates through the two arrays it is merging without ever repeating an element or skipping over elements. My merge sort implementation splits the array about the midpoint, which will have a recursion depth of $\log_2 n$ for n elements in the array. For example, given an array of 4 elements, they will first be split into 2 | 2, which are further split into 1 | 1 | 1 | 1, resulting in a recursion depth of 2. Whereas, if there are 8 elements, they will be split like so: 4 | 4 => 2 | 2 | 2 | 2 => 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1, which is a recursion depth of 3. When you combine the linear O(n) complexity of the merge function with the logarithmic O(log(n)) complexity of the merge_sort function calls, the overall complexity of the algorithm becomes log-linear O(nlogn).

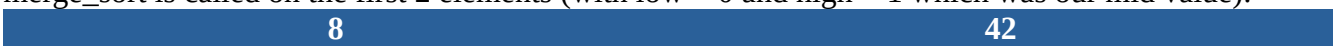
3. Manually apply your algorithm to the input below, showing each step (similar to the example seen in class) until the algorithm completes and the vector is fully sorted. Explanation should include both visuals (vector at each step) and discussion.

The midpoint is calculated: mid = (low + high) // 2 = (0 + 7) // 2 = 3

merge_sort is called on the first 4 elements (with low = 0 and high = 3 which was our mid value):

The midpoint is calculated: mid = (low + high) // 2 = (0 + 3) // 2 = 1

merge_sort is called on the first 2 elements (with low = 0 and high = 1 which was our mid value):



The midpoint is calculated: mid = (low + high) // 2 = (0 + 1) // 2 = 0

merge_sort is called on the first element, but since low == high the function returns
merge_sort is called on the second element, but since low == high the function returns

merge is called on the 2 wide array and reorders it in a temp array (in this case no changes) to:

merge_sort is called on the array:

The midpoint is calculated: mid = (low + high) // 2 = (2 + 3) // 2 = 2 which is the index of 25

merge_sort is called on the first element, but since low == high the function returns
merge_sort is called on the second element, but since low == high the function returns

merge is called on the 2 wide array and reorders it in a temp array, first placing the 3 then the 25 to form:

We go up the call stack and merge is called on

In the temp array, 3 is placed, then 8, then 25, then 42 before being compied to the original array, resulting in:

We move up the call stack back to the second half of the original array. The same steps repeat:
merge_sort is called on:

merge_sort is called on:

merge_sort is called on the first element, but since low == high the function returns
merge_sort is called on the second element, but since low == high the function returns

merge is called on:

2 is placed first into the temp array and then 3, resulting in:

merge_sort is called on:

merge_sort is called on the first element, but since low == high the function returns
merge_sort is called on the second element, but since low == high the function returns

merge is called on:

3 is placed first into the temp array and then 27, resulting in:

We move up the call stack and merge is called on:

2 is placed first, then 3, then 3, then 27, resulting in the same array of:

We finally move up the call stack and call merge on the full array of:

2 is placed from the second half, then 3 and 3 from the second half, then 3 from the first half, then 8, then 25, then 27, then 42, resulting in:

The reason the two 3s from the second array were placed first before the 3 in the first array is because I did the check: if low_element < high_element {place the low element in array}. Since low_element == high_element in this case, it would default to placing the high element into the temp array.

4. Is the number of steps consistent with your complexity analysis?
Justify your answer.

Yes. The merge part of the algorithm did n operations where n was the number of elements in the two subarray it was sorting. The merge_sort calls went to a recursion depth of 3 before returning, which is consistent with O(logn) since $\log_2 8 = 3$. Put together, the complexity is as expected: O(nlogn).