Lab #3

Exercise 01

1. See code for implementation

2. A worst-case complexity of O(nlogn) implies that there is a linear logarithmic growth of the mergesort's complexity, where n represents the input size of the array. In fact, even if the array is sorted, it would have a best-case complexity of O(nlogn).

   Looking at the code, the reason why is because the algorithm will always recursively divide the array into subarrays until the subarray has only one value. Afterwards, the subarrays then begin the merging process, through the means of creating temporary arrays and storing the values of said subarrays. Before the subarrays are merged, the algorithm compares the values between the two temp arrays, replacing the values of the original array in the correct order. If one temp array still has some values left over (due to a difference is array size), add any leftover values. All of these processes have a complexity of O(n), as they often times go through the full input. Given that none of them are nested, the total complexity of the merge step is O(n). This comparison occurs across each and every subarray that was halved; and the number of comparisons as a result is equivalent to the length of the sub-arrays being merged.

   In any instance, the main array will always be divided into multiple subarrays of two halves recursively, and said subarrays will always be merged via the means of temp arrays and comparisons, even if the original array is already ordered. The recursive splitting takes O(logn) time, and the time across all the comparisons across each merge is represented by O(n). Thus, the worst-case complexity is O(nlogn).

3. We have the following vector:

8	42	25	3	3	2	27	3

   First, we split the array into the two halves, applying the merge_sort() recursively for them:

8	42	25	3

3	2	27	3

   We split the subarrays again, utilizing merge_sort() once again recursively:

8	42

25	3

3	2

27	3

   Repeating the process once more...

8	42	25	3	3	2	27	3

The array has now been split into the smallest subarrays possible. merge() is now called upon the corresponding subarrays. The values of said subarrays are compared amongst each other, and are then properly ordered within their scope. The sub-arrays that are to be merged with each other are color-coded, as seen above:

8	42	**3**	**25**	**2**	**3**	**3**	**27**

The values above which are bolded are the ones that had their values swapped. The merged subarrays will then be merged once again, following the color code above:

3	8	25	42	2	3	3	27

Lastly, the subarrays are merged back together to form the original array, this time in a sorted manner:

2	3	3	3	8	25	27	42

4. Well, given O(nlogn), where n = 8 [As the size of the array is 8], 8log(8) gives us 24. This adds up with the number of steps associated with splitting and merging the array. The array can only be split log(n) times [3 times in this instance], and following the splits it is then merged a total of n times [8 times]. The number of steps is consistent with the complexity analysis.