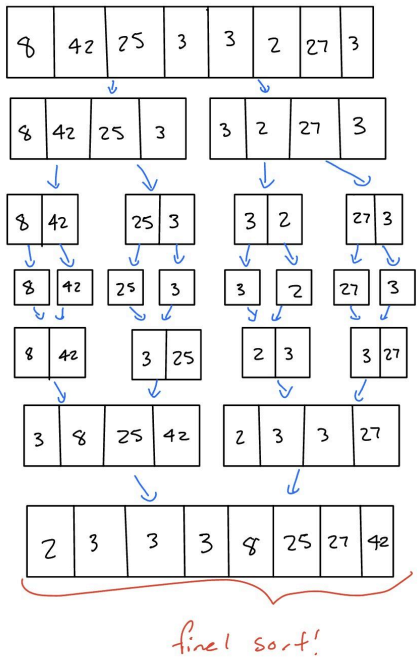
2) O(nlogn) is the time complexity of this entire technique because, in order to finish the sort with the mid = (low+high)//2, the merge_sort function first splits the array in half. Given that these lines, merge_sort(arr, low, mid) and merge_sort(arr, low, mid), are used recursively to perform this division. The temporal complexity that results from these three lines is O(log n). Second, this phase requires an O(n) time complexity since the merge function combines two sorted subarrays; n is the total number of elements in the subarrays. When O(logn) and O(n) are combined, the worst-case result is O(nlogn).

3)



The array is split in half by merge sort until there are only single elements remaining. The algorithm will then join the two parts back together to order the components from biggest to smallest. For instance, 8 and 42 are one-element arrays that are combined into one, with the greatest integer, 42, on the right, and the smallest, 8, to the left. Then, this two-element array and another two-element array will combine to form a four-element array, with the largest element on the right and the smallest on the left.

4) The goal with merge sort is that the number of steps is always the same, not dependent on the best/average/worst case scenario. There the time complexity of merging with all cases is O(nlogn). The merge sort algorithm's time complexity of O(n log n) is consistent with the number of steps it takes. The complexity analysis is verified by the number of steps increasing proportionately to n log n as the number of elements in the array rises. As a result, the merge sort algorithm behaves in a way that makes sense given its temporal complexity analysis.