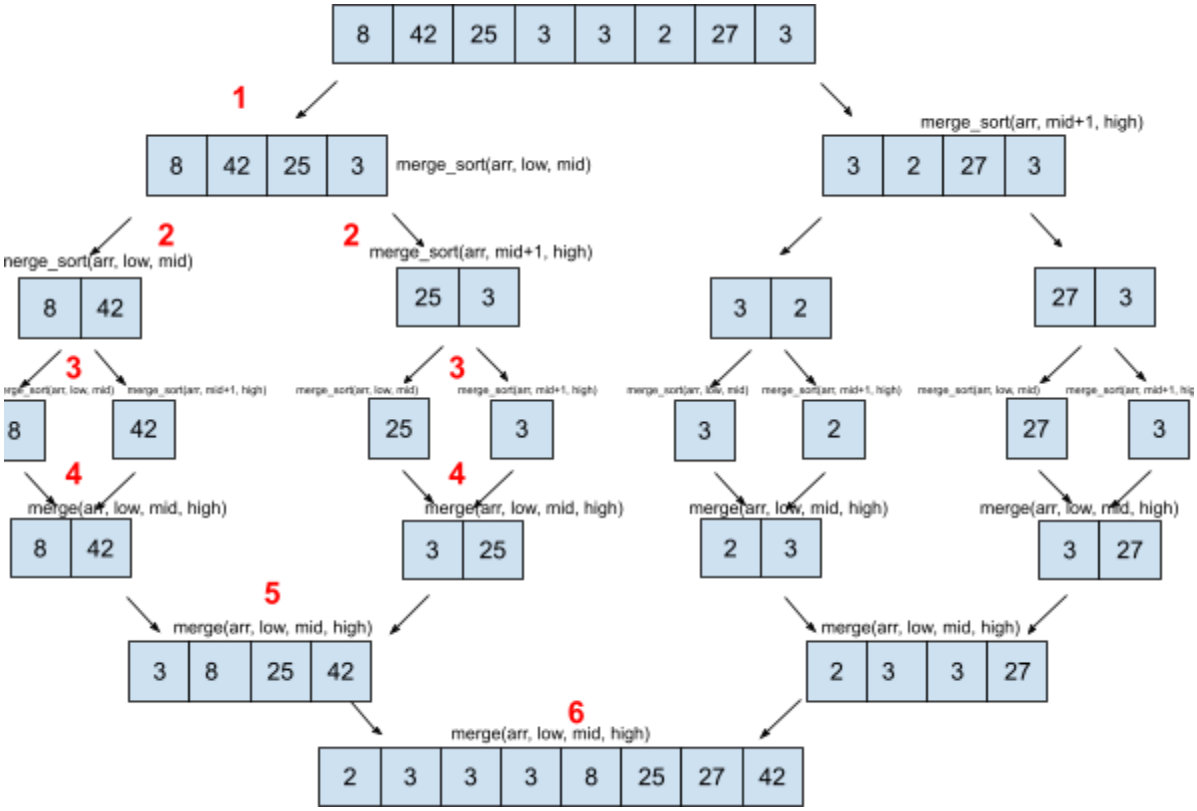
2. The time complexity of the program is PRIMARILY a result of the merge_sort function which contains recursive calls which splits the array in 2 with each call (divide and conquer), until the base case of a single element containing array is reached at which point the merge function is called which has no effect on the single element array, but once an array of two, then three is introduced, it compares current elements at indices i and j in low_half and high_half, and if the element in low_half is less than or equal to the element in high_half, its stored in the the original array (line 18). Otherwise, the element in high_half is stored in that position (line 21) and this continues as long as there are elements in low_half and high_half to compare (line 16).

As this explanation of the merge function illustrates, the execution of this function requires each element to be read and compared to the other elements (using a divide and conquer method of comparison). As demonstrated by the repeated use of the len function, the total number of elements determines how many times the function needs to be called (n). But as each element needs to be compared to every other element before it is stored (line 16-23)the original number of elements has a logarithmic effect on the complexity of the function (n log n).

3.



Step-by-step explanation:

1. First, the merge_sort function takes in an array, establishes a midpoint and splits the array into two new arrays, before calling itself to split itself into two again in step 2. At this point, the two arrays are arrays of length 4 (L containing 8,42,25,3 and R containing 3,2,27,3).

2. Step 2, is conceptually the same as step 1, approaching the same objective of splitting the array into smaller arrays until it is working with arrays of length 1 that can have its elements copied, sorted, and **merged** back into singular ordered arrays. Currently, 4 arrays of length 2 are being worked with (LL containing 8,42, LR containing 25,3, RL containing 3,2, and RR containing 27,3)
3. In step 3, the recursive call has reached the aforementioned base case of single element arrays using the same splitting process in step 1 and 2. Currently, 8 arrays of length 1 are being worked with (LLL: 8, LLR: 42, LRL: 25, LRR: 3, RLL: 3, RLR:2, RRL: 27, RRR:3)
4. At step 4, merge is called where each of the pairs split in step 3 are merged back together into an array of length 2 but ordered.
5. Same as step 4. Because the values of the recursive calls are being returned, multiple calls to merge are occurring and in this step, the elements of two arrays of length two are being compared then placed in a single array of length 4 in order.
6. Same as step 4 and 4. Now, the elements of two arrays of length four are being compared then placed in a single array of length 8 in order. This is the final call to merge and thus where the merge_sort returns a new ordered array then terminates.

4. The number of steps in the merge_sort function are consistent with the complexity analysis. As seen, the function recursively calls itself and each time splits itself in two until each element is in its own array. This part takes O(log n) steps where the original array length is n elements. Then when the call(s) to the merge function occurs another O(n) steps are needed to return each of the elements to an array of length n but ordered. The total number of steps can be acquired by a product of these two phases as a split and merge must occur for each element, thus, resulting in an overall complexity of O(n log n).