2. In the ‘merge_sort()’ function, the array is divided two halves recursively until each sub-array contains a single element. This division follows a binary tree structure, where each level is a halving of the array. Since at each level the size of the array is halved, the depth of this tree is *logn*. Therefore, the total number of recursions is *logn*. The ‘merge()’ function takes two sorted lists and merges them into a single sorted list. It does this by iterating through both lists once, comparing the smallest unmerged elements from each list and adding the smallest one to the merged list. In the worst case, this function iterates through every element in both lists exactly once. Since the lists being merged at each step of the algorithm contain *n* elements, the times complexity of the ‘merge()’ function at each level of recursion is O(*n*). Since there are *logn* levels of recursion (from the divide phase) and at each level, the merging operation takes O(*n*) time, the worst-case time complexity is O(*nlogn*).
3. **Initial Array:**

   [8, 42, 25, 3, 3, 2, 27, 3]

   Step 1: Divide

   First, we divide the array recursively until we have sub-arrays that can not be divided further.

   - **[8, 42, 25, 3]** and **[3, 2, 27, 3]**

   Further dividing these arrays:

   - **[8, 42]** and **[25, 3]**
   - **[3, 2]** and **[27, 3]**

   Then divide further until each sub-array has only one element:

   - **[8] [42]** and **[25] [3]**
   - **[3] [2]** and **[27] [3]**

   Step 2: Conquer and Merge

   Now, we start merging these sub-arrays back together in a sorted order.

   1. Merge **[8]** and **[42]**
   - **[8, 42]**
   2. Merge **[25]** and **[3]**
   - **[3, 25]**
   3. Merge **[3]** and **[2]**
   - **[2, 3]**
   4. Merge **[27]** and **[3]**
   - **[3, 27]**

After merging these sub-arrays, we get:

- **[8, 42]** and **[3, 25]**
- **[2, 3]** and **[3, 27]**

Next, we merge these:

1. Merge **[8, 42]** and **[3, 25]**

- Compare **8** and **3**: **3** is smaller
- Compare **8** and **25**: **8** is smaller
- **42** and **25** are left, add them in order: **[3, 8, 25, 42]**

2. Merge **[2, 3]** and **[3, 27]**

- Compare **2** and **3**: **2** is smaller
- Compare **3** and **3**: **8** both are equal, add both
- **27** is left, add them in order: **[2, 3, 3, 27]**

Now, we have two halves:

- **[3, 8, 25, 42]** and **[2, 3, 3, 27]**

Step 3: Final Merge

Finally, we merge these two halves:

- Compare **3** and **2**: **2** is smaller
- Compare **3** and **3**: both are equal, add both
- Compare **8** and **3**: **3** is smaller
- Now, **8** and **27** are compared: **8** is smaller
- **25** is compared with **27**: **25** is smaller
- **42** and **27** are left, add them in order: **[2, 3, 3, 3, 8, 25, 27, 42]**

Final Sorted Array:

- **[2, 3, 3, 3, 8, 25, 27, 42]**

4. Yes, the number of steps illustrated above and the process described previously are consistent with the complexity analysis of O(*nlogn*) for the merge sort algorithm. The visual and step-by-step explanation showcases the division and merging process in detail, aligning with the O(*nlogn*) complexity by illustrating the depth of recursion (*logn*) and the linear work done at each level of merging (*n*).