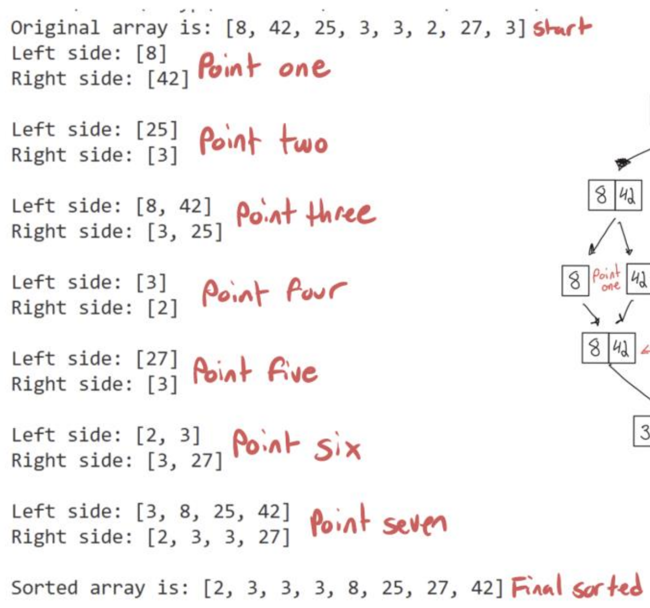
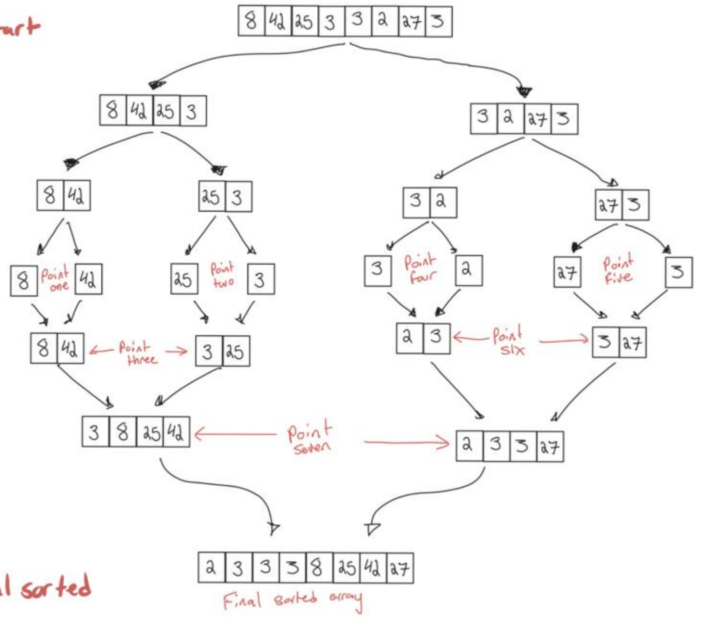
Example 1

2. It is a merge sort because after determining the midpoint of the array and each sub array, it will then sort sub-array which makes the time complexity for each call O(n/2). When the merge is called, it takes O(n) amount of time since it is only called that many times. We take the higher of the two times, which is O(n). When we implement the recursive tree into it, the recursive time complexity is O(log(n)). Once combined, we get a final time complexity of O(n * log(n)).

3.



```
Original array is: [8, 42, 25, 3, 3, 2, 27, 3]
Left side: [8]
Right side: [42]

Left side: [25]
Right side: [3]

Left side: [8, 42]
Right side: [3, 25]

Left side: [3]
Right side: [2]

Left side: [27]
Right side: [3]

Left side: [2, 3]
Right side: [3, 27]

Left side: [3, 8, 25, 42]
Right side: [2, 3, 3, 27]

Sorted array is: [2, 3, 3, 3, 8, 25, 27, 42]
```



The midpoint between the vector is found then split into two halves. Each half is then halved and then halved again. Once the program is down to a single value in the left and right arrays, it compares the two values and orders them from smallest to largest. It then combines that ordered pair to another ordered pair in the initial half, the orders that make an array for 4 ordered values. It repeats the other half, then combines it all in the end to complete the sorted array.

4. Yes, it is consistent. In step one, there is one element. Step two has two, but step three has four. This doubles in step four with eight elements in this step. This shows that it has 2^(n - 1) elements per n, which also tells us it is a divide and conquer sorting algorithm, which has a time complexity of O(n * log(n))