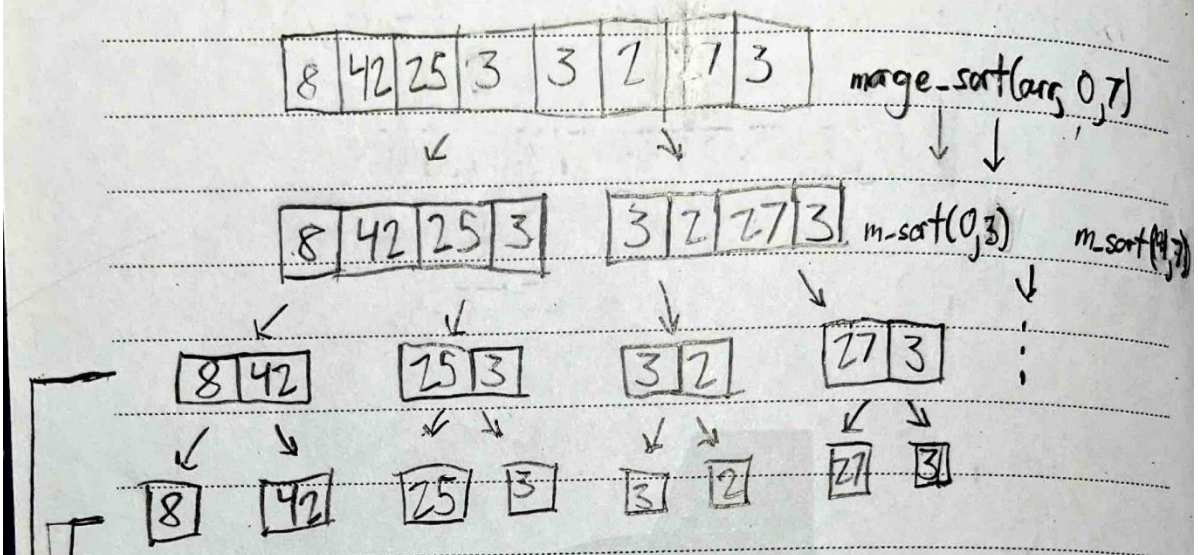
Exercise 1 Questions

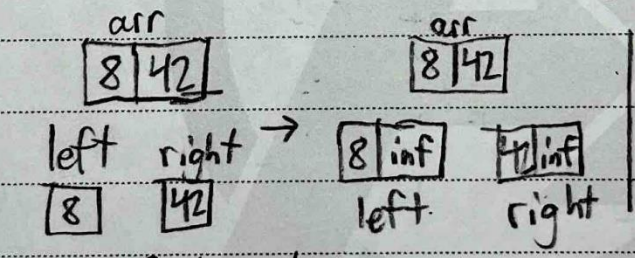
2) Argue that the overall algorithm has a worst-case complexity of O(nlogn). Note, your description must **specifically refer to the code you wrote**, i.e., not just generically talk about mergesort.

The worst-case scenario for mergesort is the same for both its average and best-case scenario. That is, the merge_sort() implementation will always divide the array into halves until it reaches arrays of size 1. As such, the complexity is consistent regardless of different organizations/cases. The complexity of the algorithm itself depends on its two parts division and merging. In the division portion (the merge_sort()) the function recursively calls itself to divide the array into halves until it reaches subarrays of size one. On the first division, two subarrays are made of size n/2, then four subarrays of size n/4, and so on. Thus, the number of divisions is logarithmic with respect to the array size; O(log n). The merge() function iterates over each element in the input arrays exactly once. Since the number of comparisons and movements of elements is directly proportional to the total number of elements being merged (i.e., the sum of the lengths of the left and right arrays), the time complexity of the merge() function is O(n). In addition, the merge() function does not involve nested loops nor recursive calls that would change the complexity of the function beyond a linear scan of the elements. Therefore, the overall algorithm has a complexity of O(nlog n) as this combines the complexity of the merge_sort() and merge() algorithms.

3) Manually apply your algorithm to the input below, showing each step (like the example seen in class) until the algorithm completes and the vector is fully sorted. Explanation should include both visuals (vector at each step) and discussion.



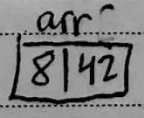
- Recursive array splitting via merge sort until low < high condition is no longer true
  - → merge_sort() does not do anything at this level; low = high
  - → merge_sort calls merge(arr, 0, 1, 1) at this subarray; (low, mid, high)



for loop starts

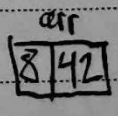
8 ↓ 42

left[i] < right[j] so, arr[k] = left[i]



i=0+1
j=0
k=0+1

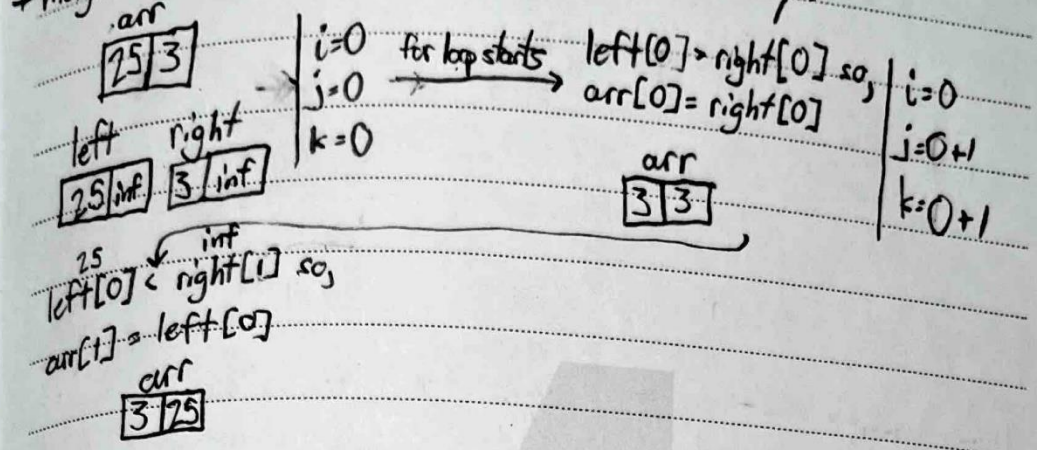
left[i] ≮ right[j]
inf ≮ 42
so,



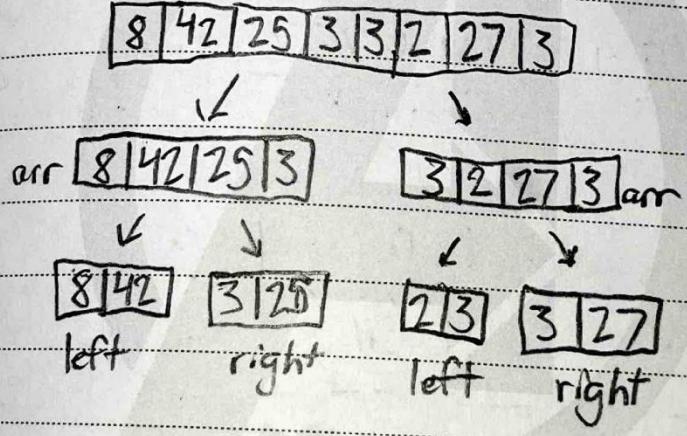
Done →

- Note: inf appending simplifies boundary checking.

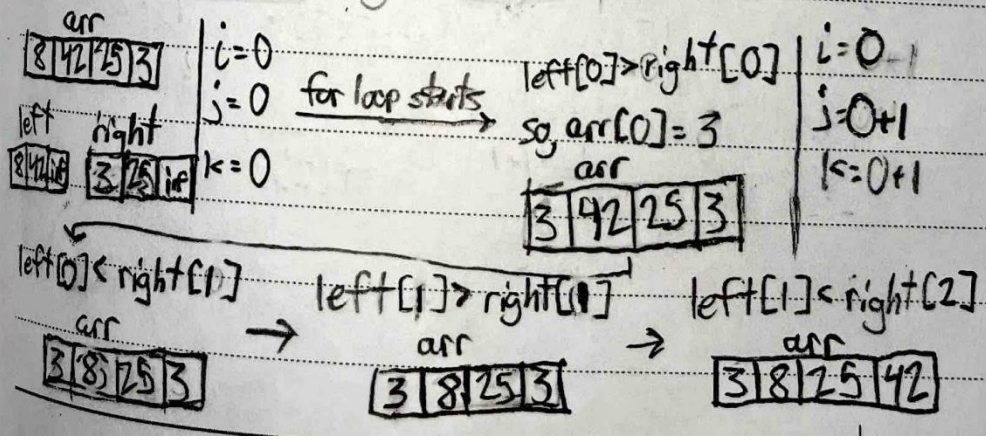
→ merge_sort calls merge for [25|3] subarray:



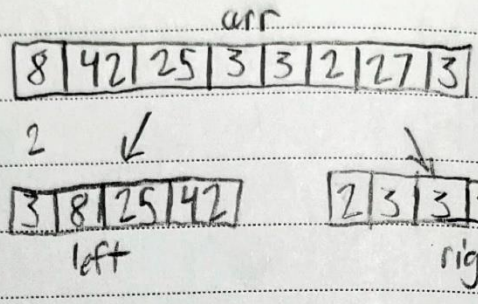
• This process repeats for the other two subarrays, now our call stack looks like:



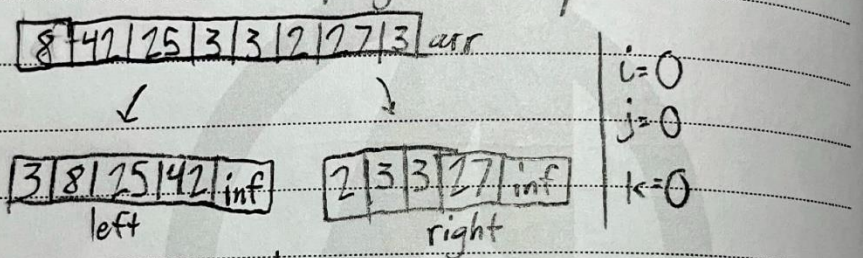
→ merge_sort calls merge for [8|42|25|3]:



- Same process for other half, now the stack looks like:



→ now, merge_sort calls merge for the whole array with sorted left & right subarrays:



↓ for loop starts

left[0] > right[0] | i=0, j=0+1, k=0+1
arr[0] = 2

left[0] ≤ right[1] | i=0+1, j=1, k=1+1
arr[1] = 3

left[1] > right[1] | i=1, j=1+1, k=2+1
arr[2] = 3

left[1] > right[2] | i=1, j=2+1, k=3+1
arr[3] = 3

left[1] < right[3] | i=1+1, j=3, k=4+1
arr[4] = 8

left[2] < right[3] | i=2+1, j=3, k=5+1
arr[5] = 25

↳

→ left[3] > right[3] | i=3, j=3+1, k=6+1 → left[3] < right[4] (inf)

arr[6] = 27 | arr[7] = 42

- merge() ends and the array is sorted:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
2	3	3	3	8	25	27	42

Sorted array.

4) Is the number of steps consistent with your complexity analysis? Justify your answer.

The number of steps is in fact consistent with the complexity analysis; that the steps for division and merge follow the O(log n), and O(n) complexities respectively. In the division portion (recursive merge_sort() call), the example array has size 8 (n=8) and the number of divisions needed follows. For the given array size of 8, first division divides 8 elements into two halves of four, and so on making for 3 total divisions (or levels). Note that the number of steps required to reach sub-arrays of size one is indeed $\log_2(8) = 3$, thus proving that the complexity of the division portion is logarithmic relative to array size. In the sorting and merging (merge() function calls), this process is undoubtedly O(n) because of how it interacts with varying input size. Whenever called, the only non-constant portion of code complexity-wise is the “for k in range(low, high+1)“ loop, and we can see from the manual application that the number of iterations depended directly on the array size; sorting an array of 3 elements has the “for” loop iterate 3 times, over each element once. As such, increasing the number of elements of the array by for example one element, would increase the total number of loop iterations in an amount directly proportional to the increase in array size since the “for” loop iterates one extra time for each subproblem.