Question 2

By the merge_sort() function, an array would have to be split $\log_2(n)$ times.

```python
def merge_sort(arr, low, high):
    if low < high:
        mid = (low + high) // 2
        merge_sort(arr, low, mid)
        merge_sort(arr, mid + 1, high)
        merge(arr, low, mid, high)
```

Question 2

Once one of the sub-arrays has a length of 1, the merge() function will be called. The merge function takes the lower index of the left_arr, the higher index of the right_arr, and the middle index between the two. We then create two new sub-arrays and use two pointers to traverse each array and insert the smallest element of the two back in order into the original array starting at the lower index.

```python
def merge(arr, low, mid, high):
    l_length = mid - low + 1
    r_length = high - mid

    left_arr = [0] * l_length
    right_arr = [0] * r_length

    for i in range(l_length):
        left_arr[i] = arr[low + i]
    for i in range(r_length):
        right_arr[i] = arr[mid + i + 1]

    i = 0
    j = 0
    k = low
    while i < l_length and j < r_length:
        if left_arr[i] <= right_arr[j]:
            arr[k] = left_arr[i]
            i += 1
        else:
            arr[k] = right_arr[j]
            j += 1
        k += 1
```

Question 2

Once we reach the end of one of the arrays, the remaining elements in the other array are inserted. Thus, there will be n comparisons/insertions done $\log_2(n)$ times, meaning that the time-complexity is O(nlog(n)).

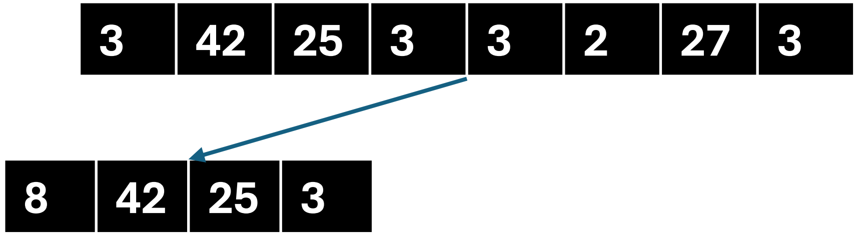
```
while i < l_length:
    arr[k] = left_arr[i]
    i += 1
    k += 1

while j < r_length:
    arr[k] = right_arr[j]
    j += 1
    k += 1
```

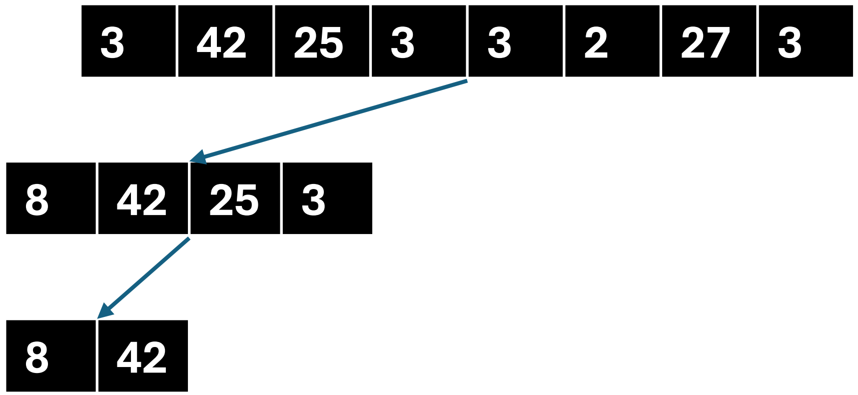
Question 3

3	42	25	3	3	2	27	3

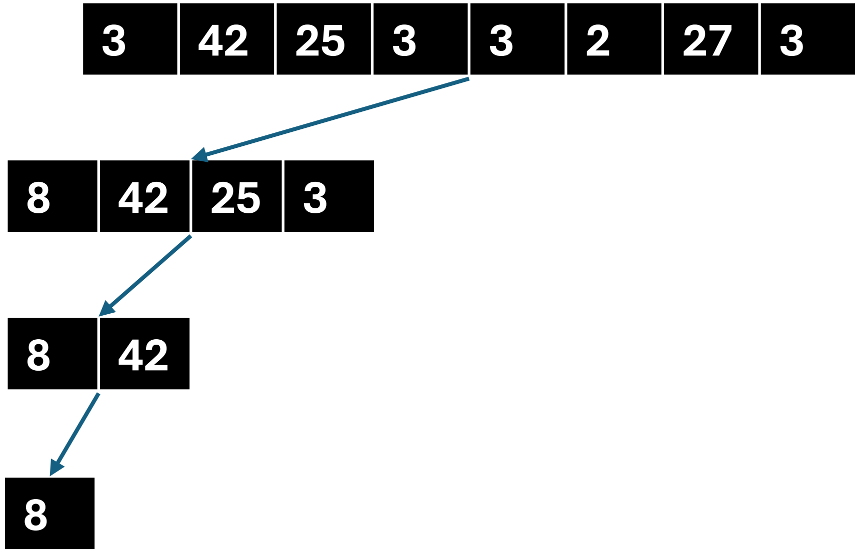
Initial array



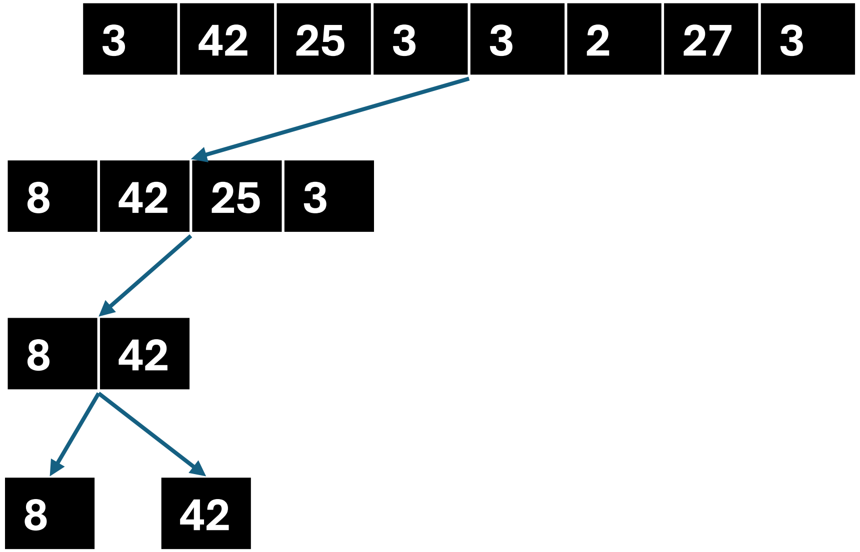
Call merge_sort() on lower half.



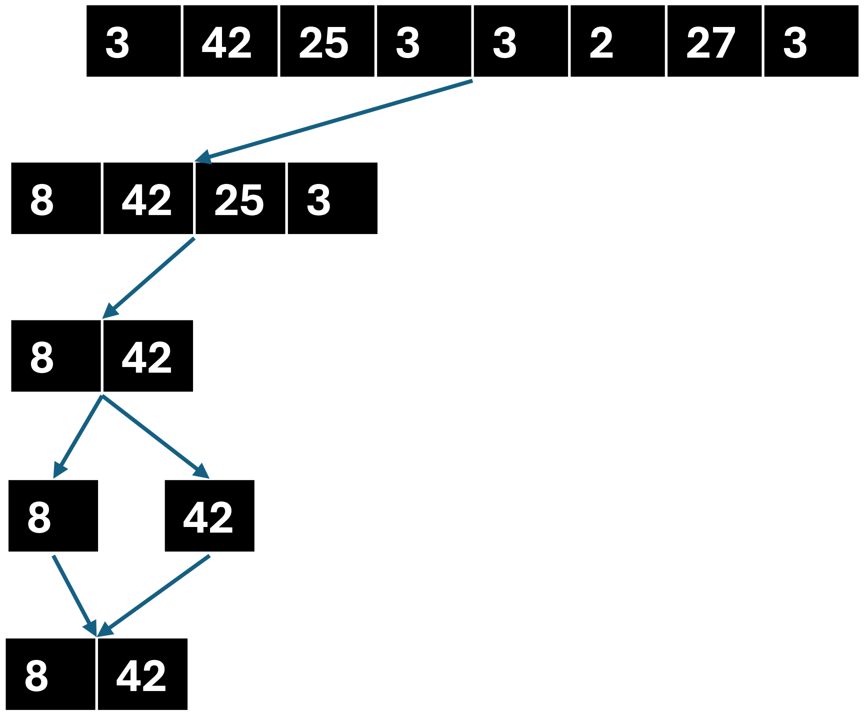
Call merge_sort() on lower half.



Call merge_sort() on lower half.

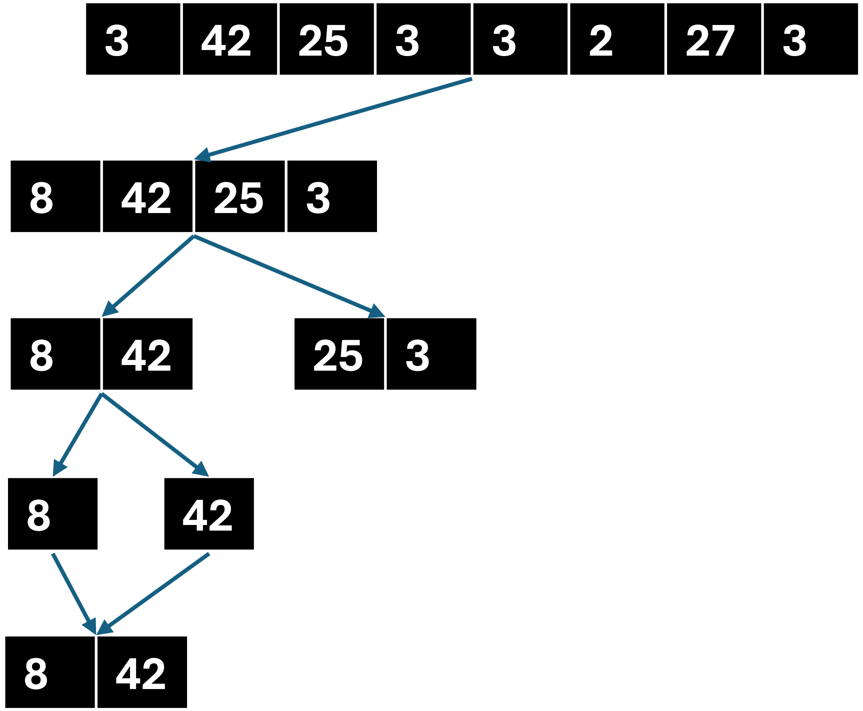


Call merge_sort() on upper half.



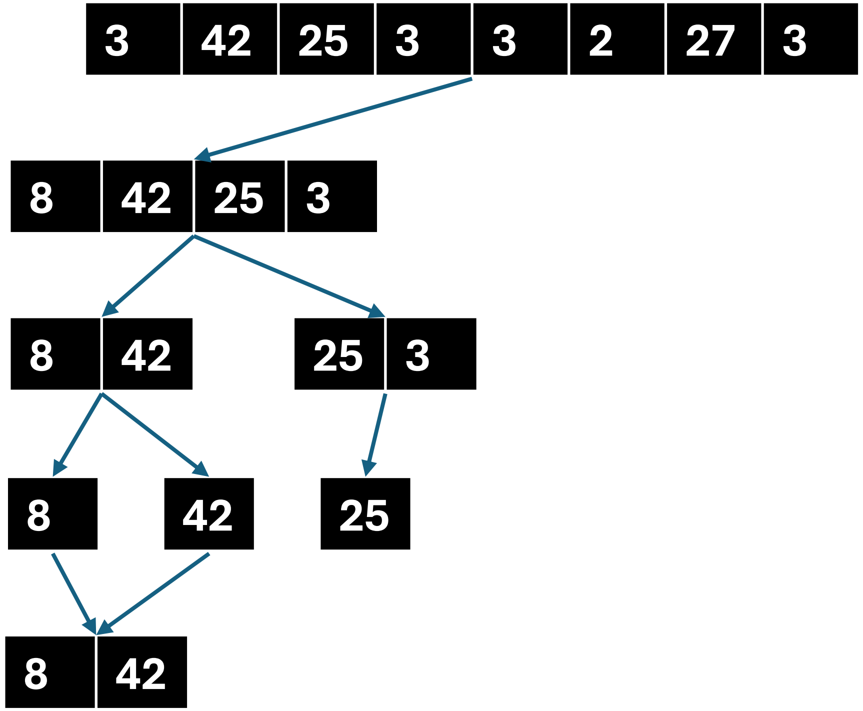
low >= high in previous two merge_sort() calls so merge() is called.

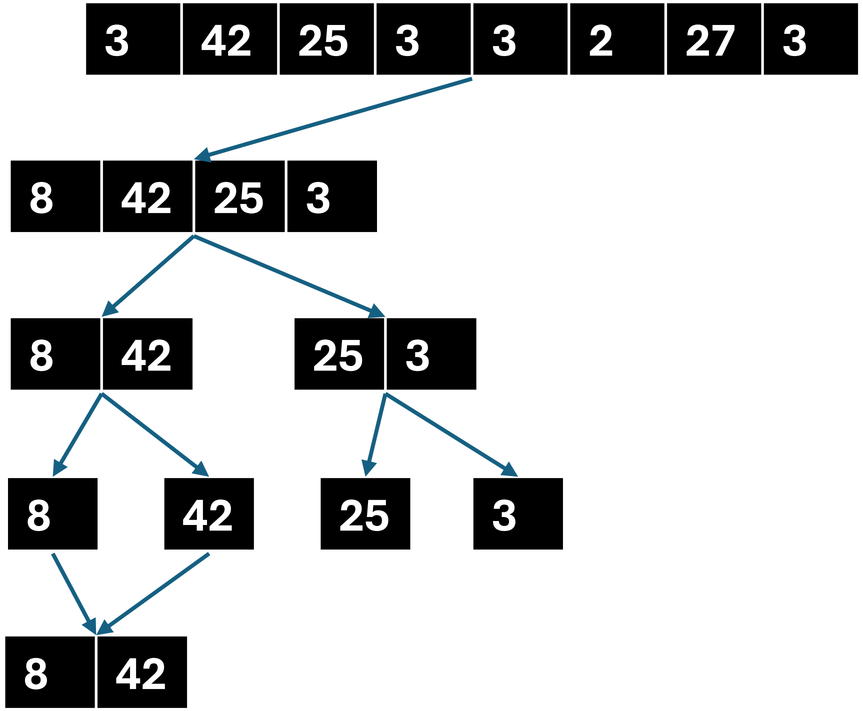
+2 comparisons/insertions (2 total)



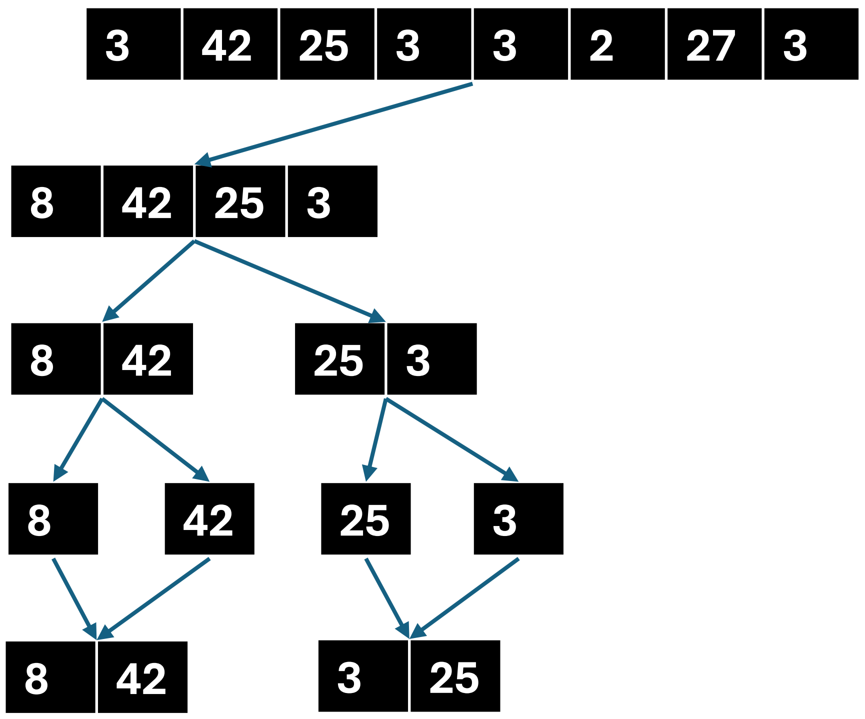
Call merge_sort() on upper half.

Call merge_sort() on lower half.



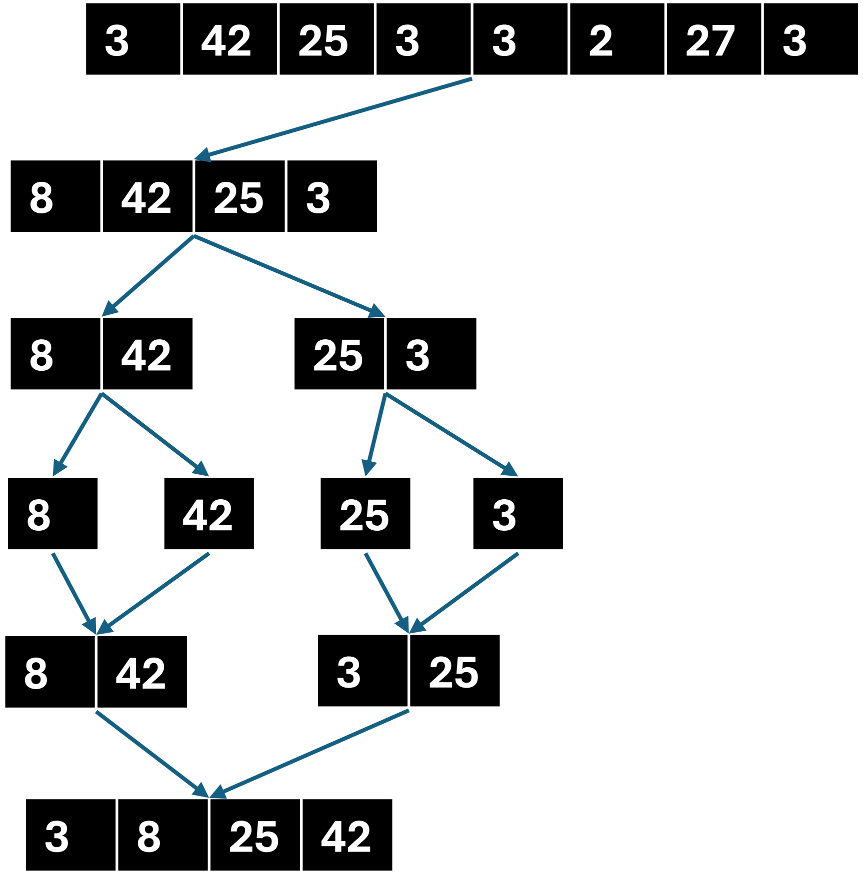


Call merge_sort() on upper half.



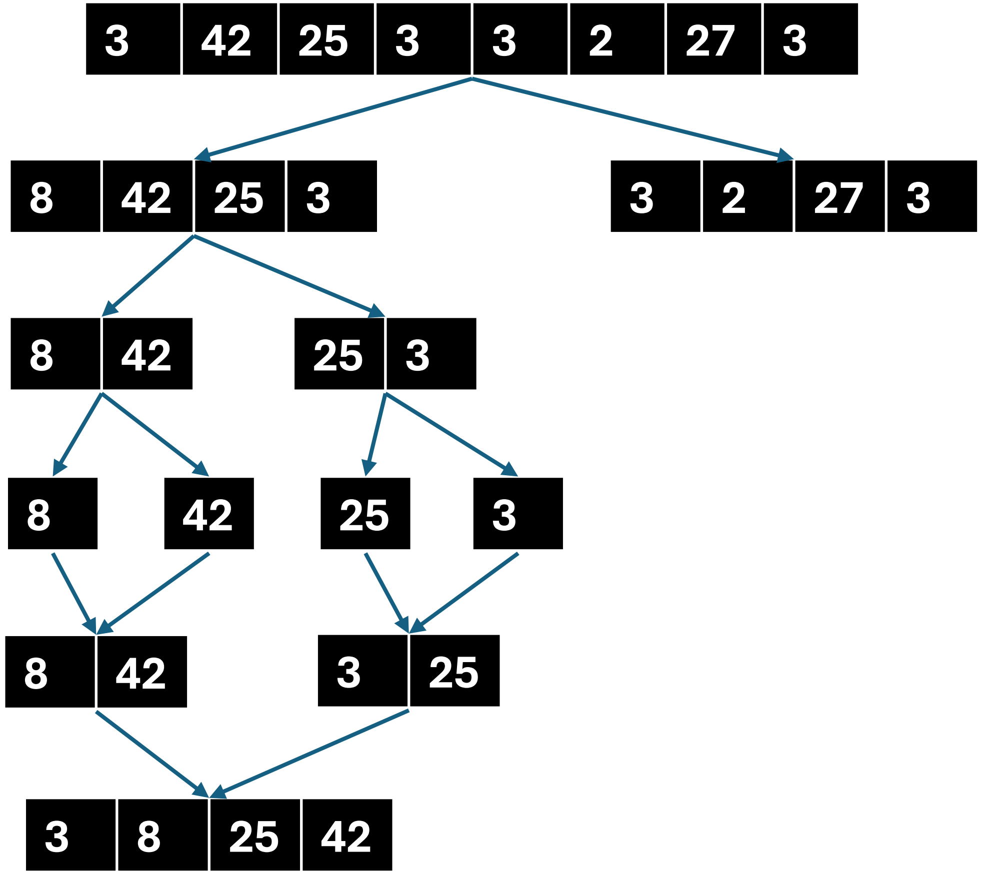
low >= high in previous two merge_sort() calls so merge() is called.

+2 comparisons/insertions (4 total)

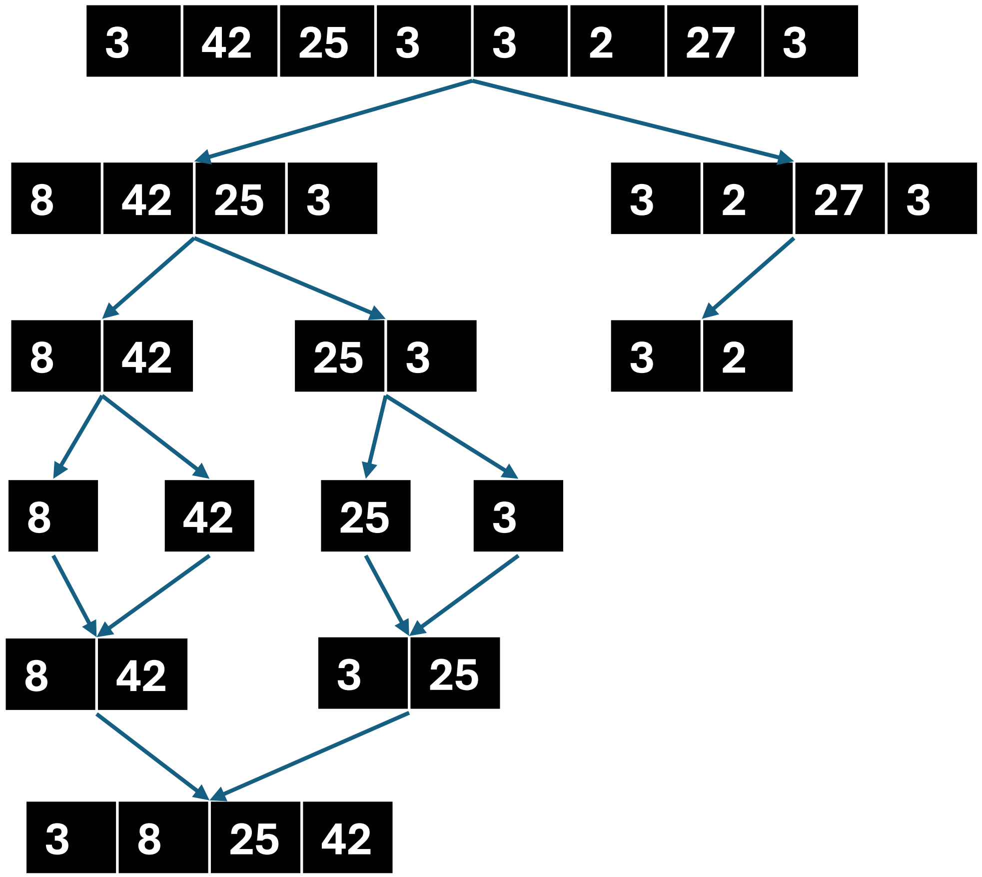


Previous 2 merge_sort() calls returned. merge() called.

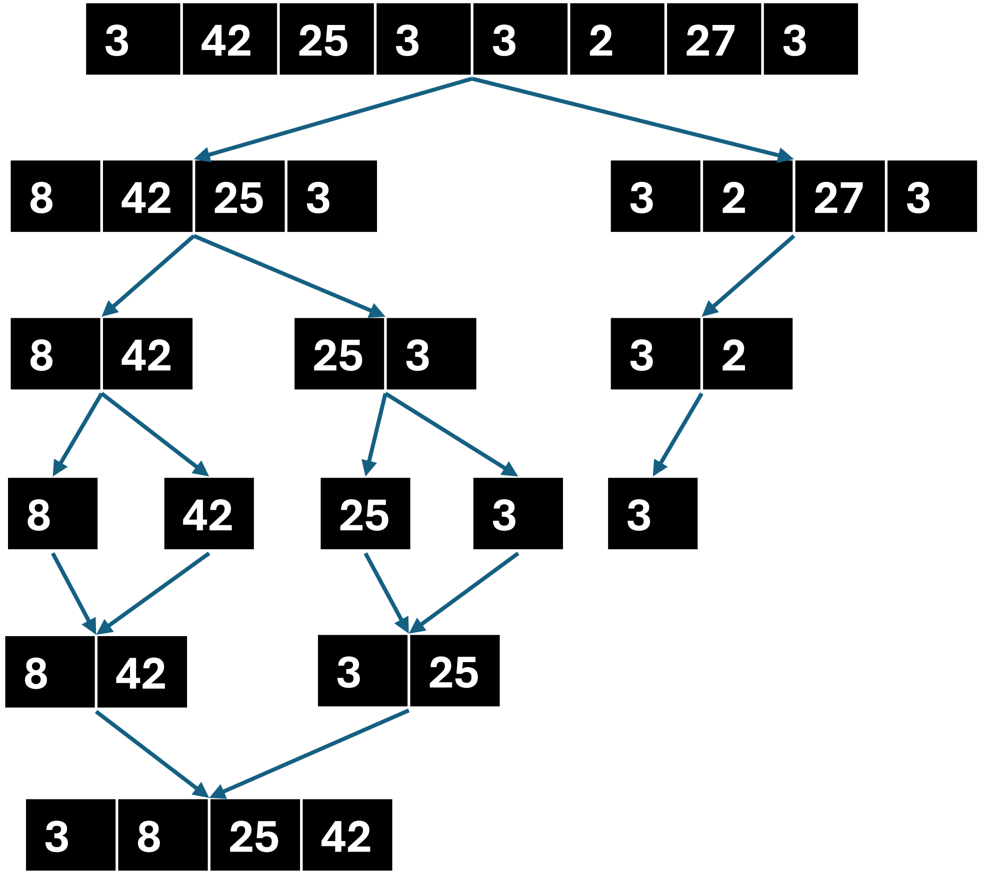
+4 comparisons/insertions (8 total)



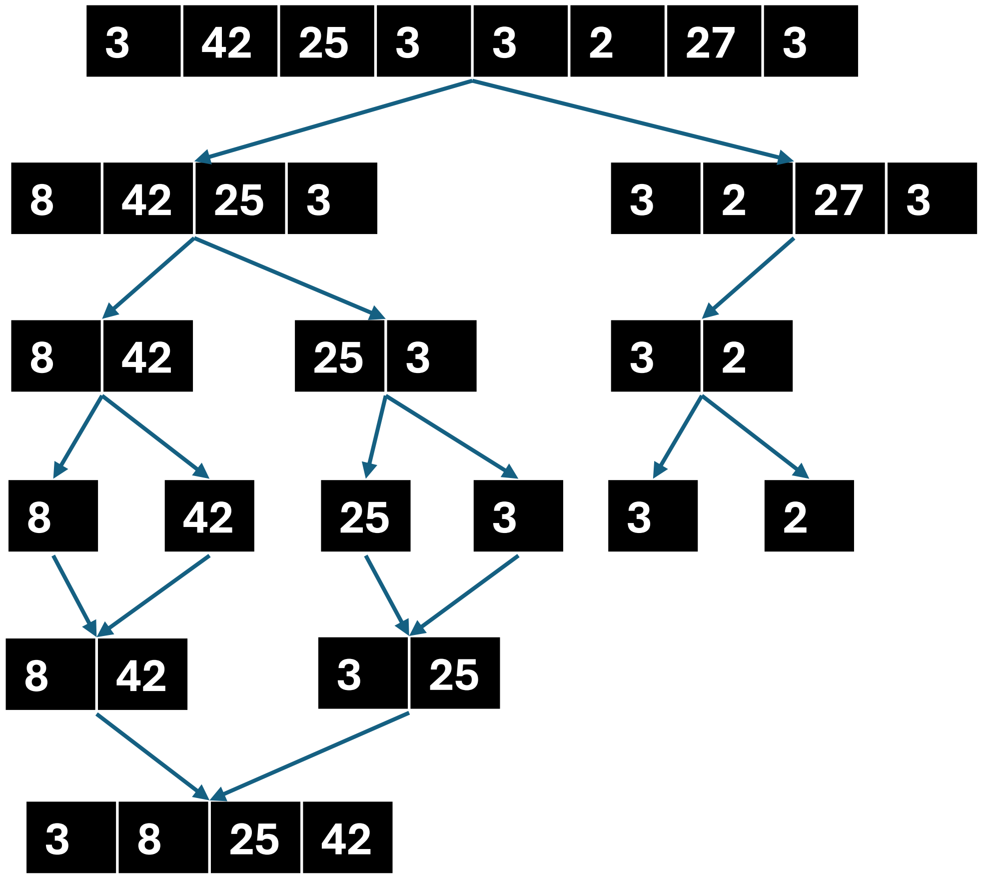
Call merge_sort() on upper half.



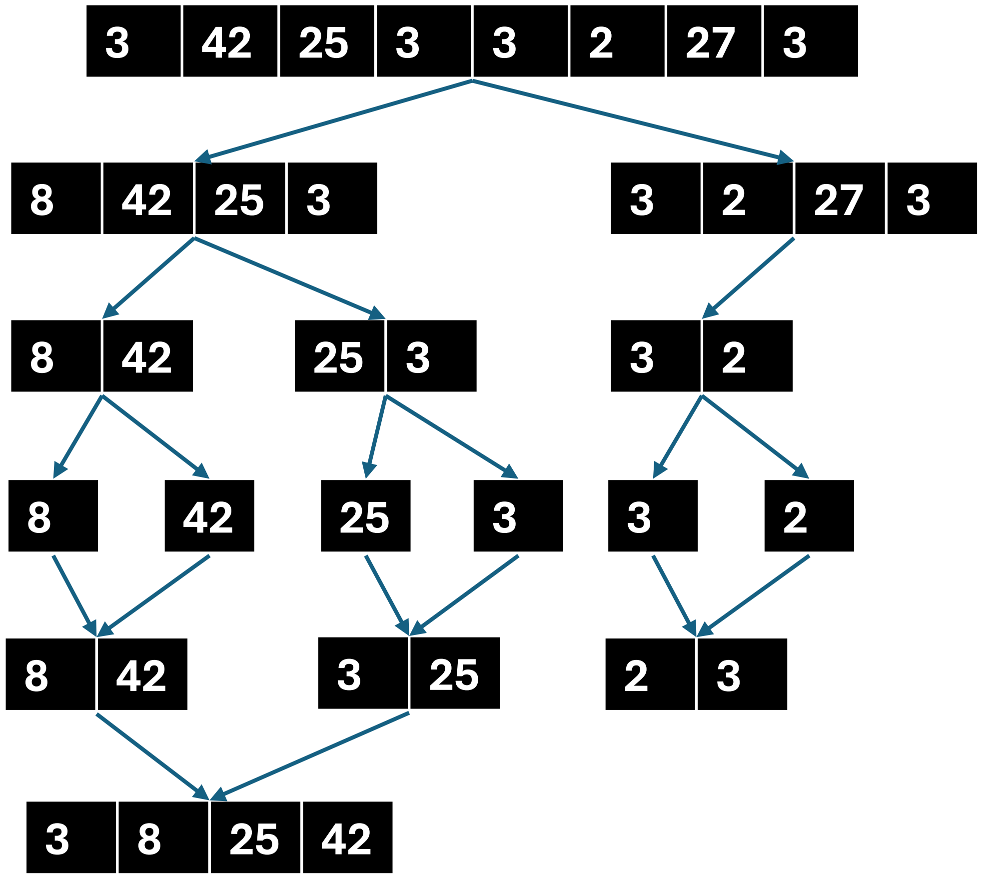
Call merge_sort() on lower half.



Call merge_sort() on lower half.

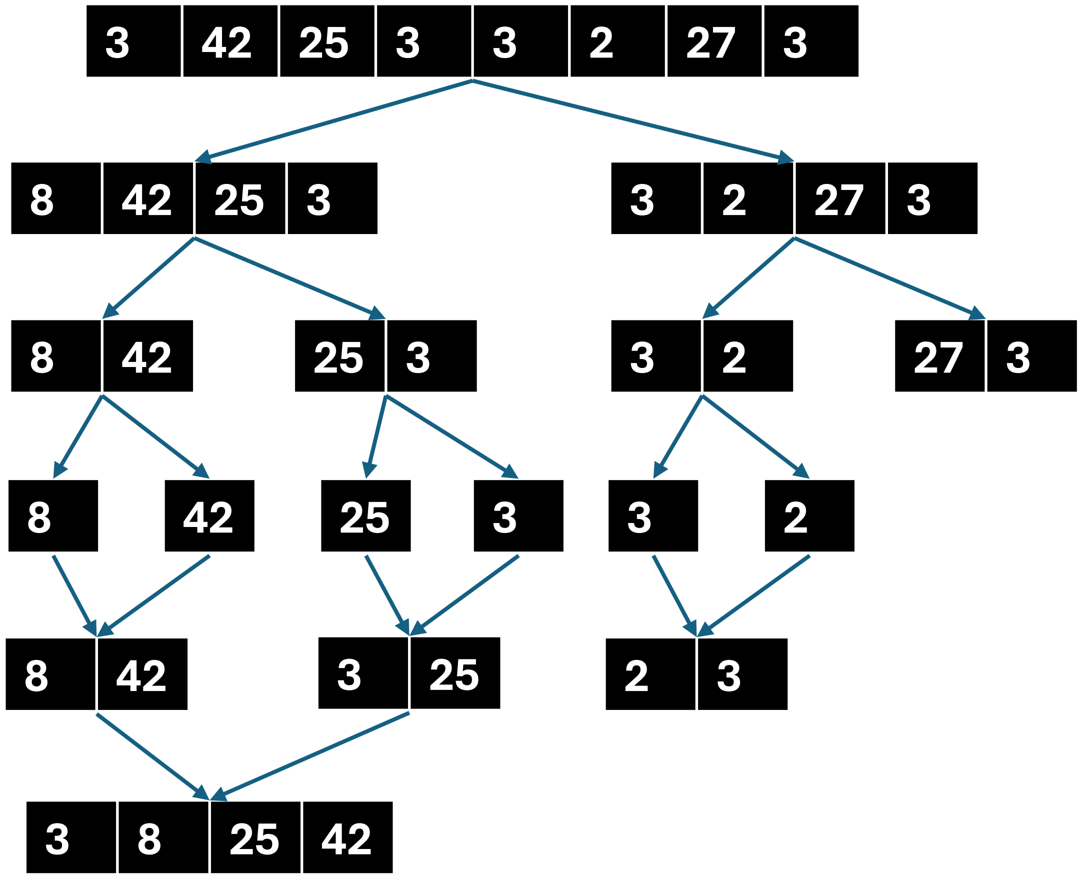


Call merge_sort() on upper half.

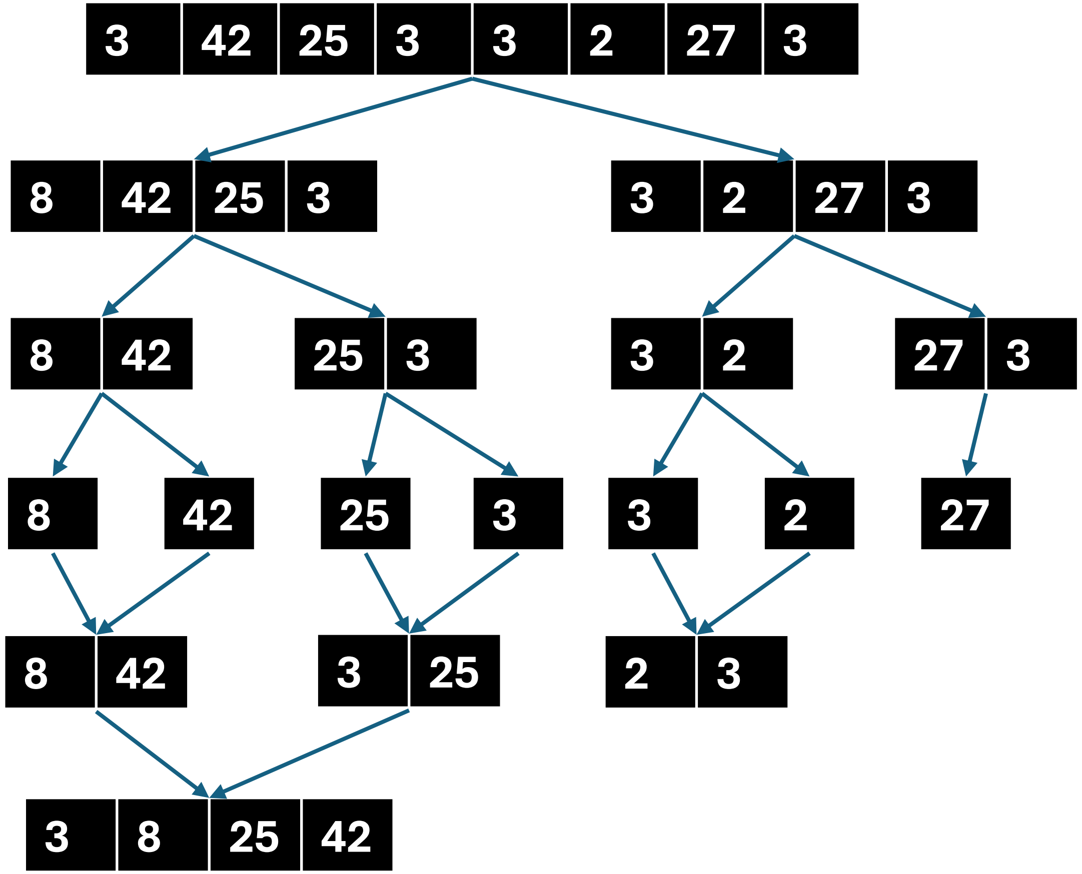


low >= high in previous two merge_sort() calls so merge() is called.

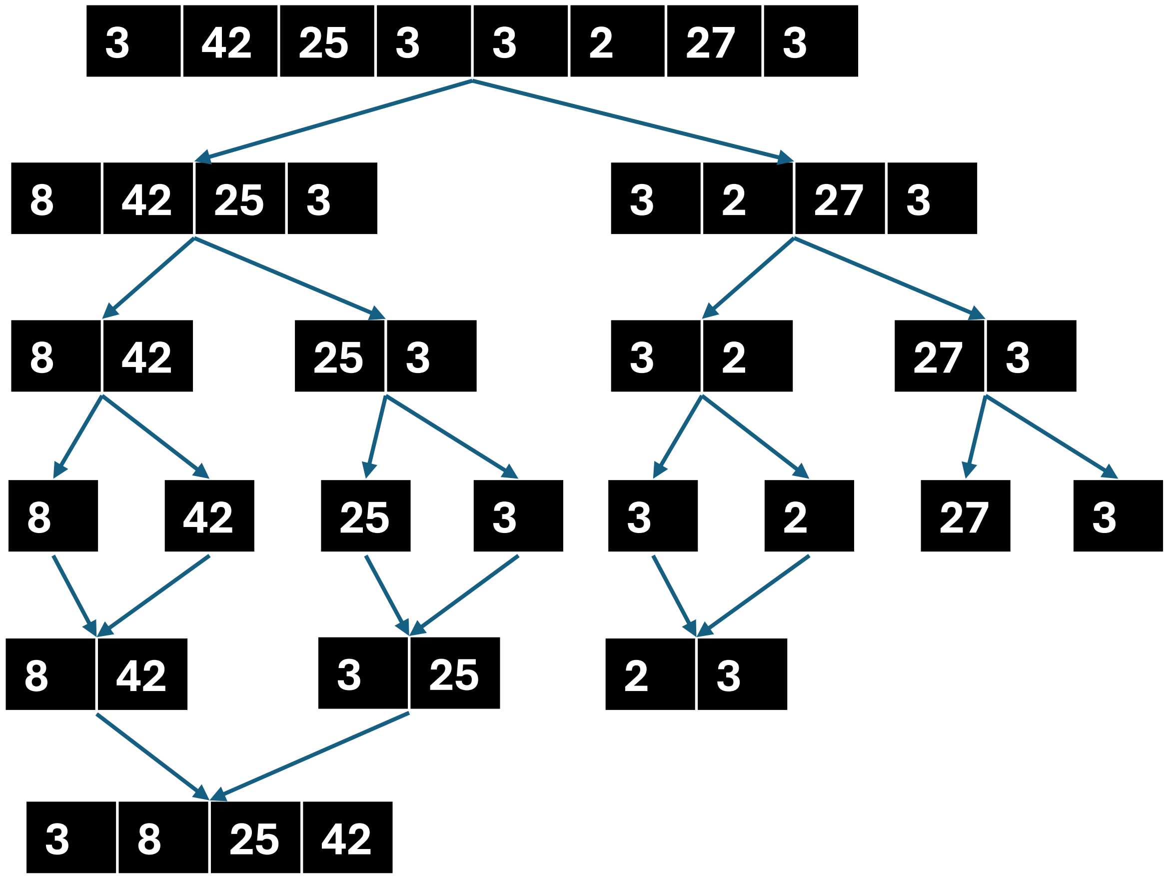
+2 comparisons/insertions (10 total)



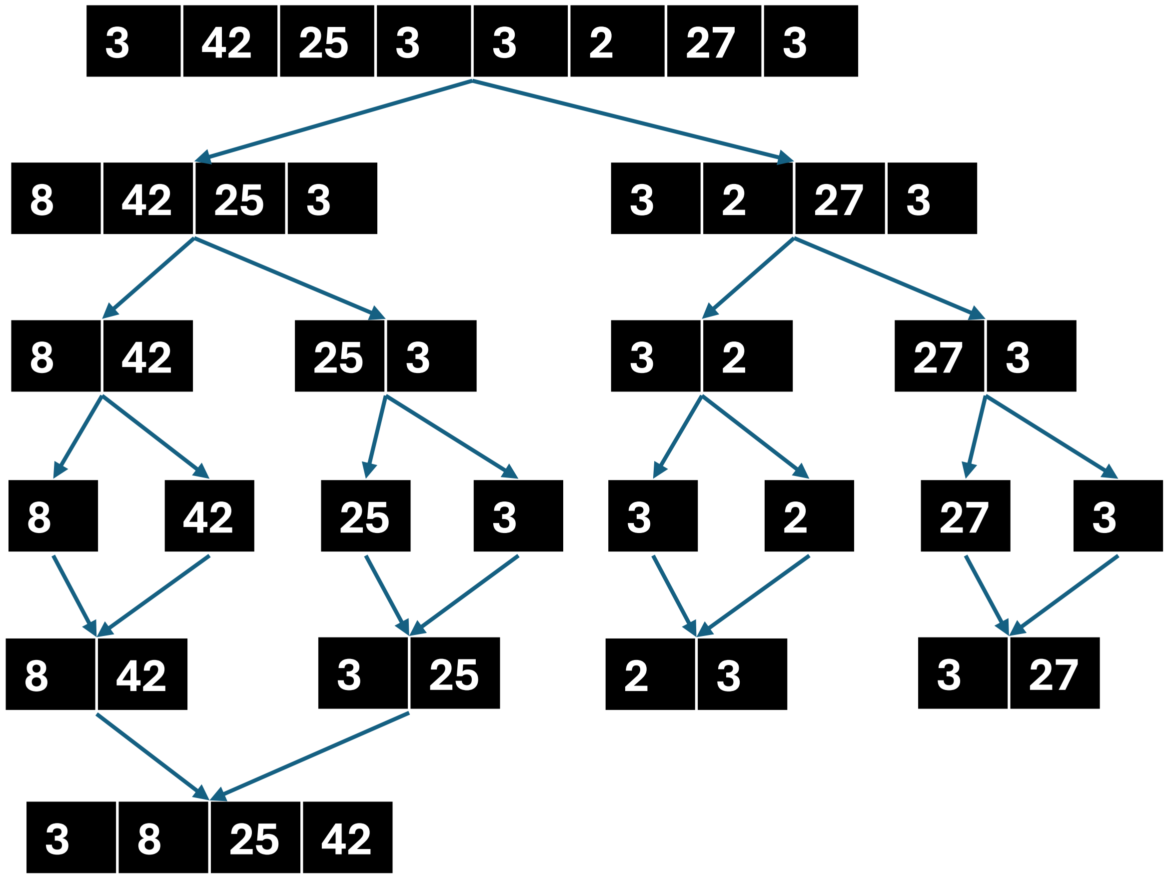
Call merge_sort() on upper half.



Call merge_sort() on lower half.

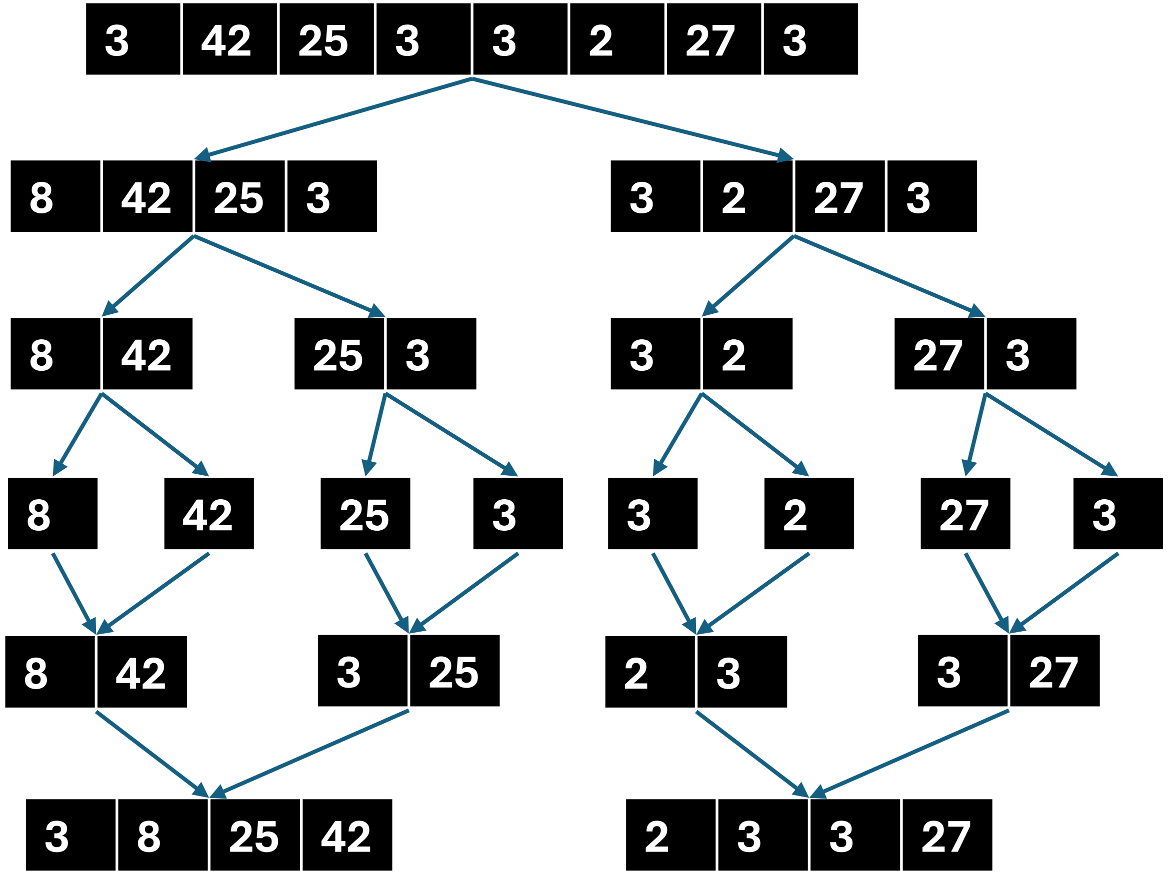


Call merge_sort() on upper half.



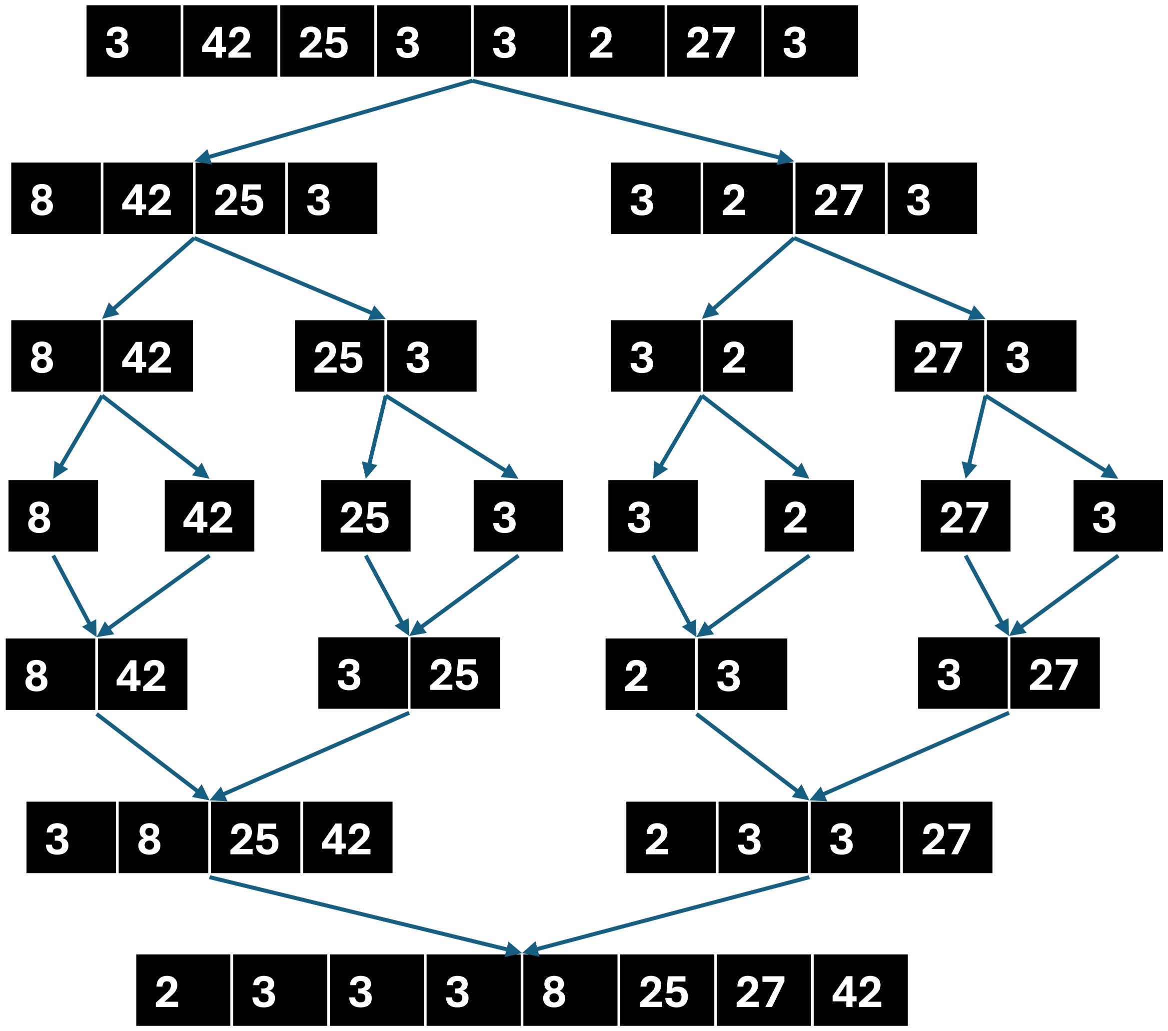
low >= high in previous two merge_sort() calls so merge() is called.

+2 comparisons/insertions (12 total)



Previous 2 merge_sort() calls returned. merge() called.

+4 comparisons/insertions (16 total)



Previous 2 merge_sort() calls returned. merge() called.

+8 comparisons/insertions (24 total)

$24 = 8 * 3 = n * \log_2(n)$

Question 4

This turns out to be consistent with my previous analysis, requiring n comparisons/insertions done $\log_2(n)$ times.