1. Completed in ex1.py
2. The definition of an O(nlogn) complexity is that it divides the set of data in half then operates on the subsets of data which merge sort does do. In the merge_sort() function

```
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)
```

We can see here that it recursively splits the array into subarrays until there are 1 element per subarray

```
merge_sort(arr,low,mid)
merge_sort(arr,mid+1, high)
```

Then the merge() function is called and operates on those subarrays

```
n1 = mid - low + 1
n2 = high - mid

L = [0] * n1
H = [0] * n2

for i in range(n1):
    L[i] = arr[low + i]
for j in range(n2):
    H[j] = arr[mid + 1 + j]
```

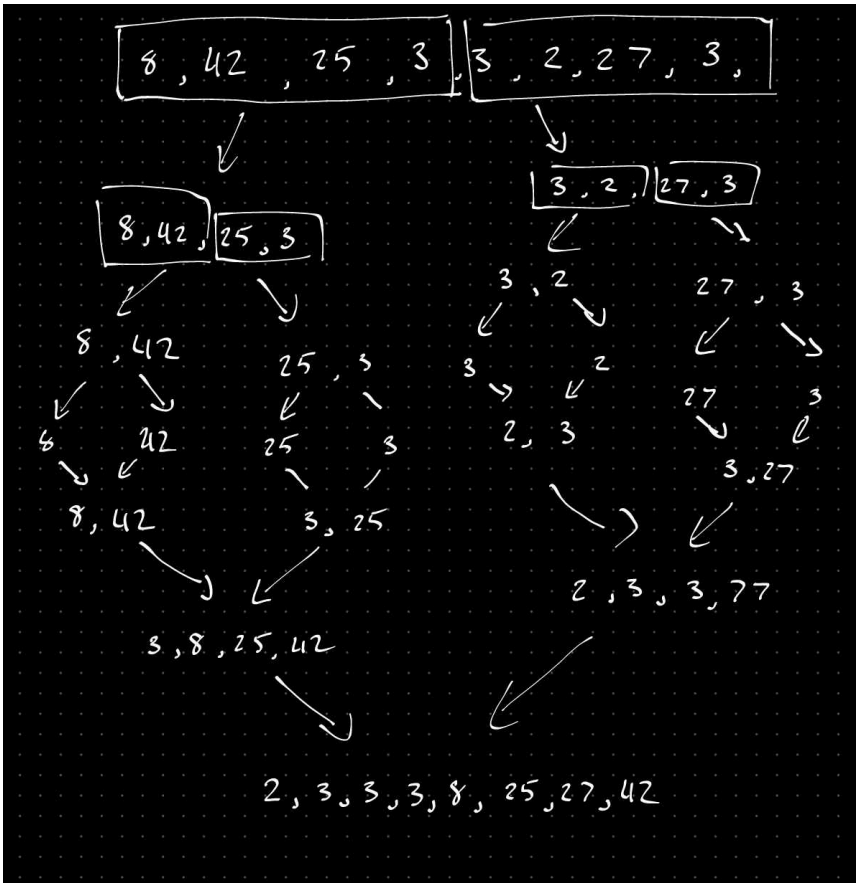
This will first copy the elements into left and right into the Low and High arrays respectively

```
i = 0 # temp index
j = 0 # temp index
k = low # main index

while i < n1 and j < n2:
    if L[i] <= H[j]:
        arr[k] = L[i]
        i += 1

    else:
        arr[k] = H[j]
        j += 1
    k += 1
```

Then it will compare values in an O(n) complexity and insert them back to the main array in a sorted manner

```
while i < n1:
      arr[k] = L[i]
      i += 1
      k += 1

while j < n2:
      arr[k] = H[j]
      j += 1
      k += 1
```

If any elements are left it will be taken care of by this segment of code

3. 

4. Yes the number of steps follows the O(nlogn) complexity. With each level of recursion the complexity is O(logn) and each time an element is processed and sorted it is O(n) complexity