Homework Assignment #3
Due: January 31, 2025 at 5:00 p.m.

The same rules apply as for Assignment 1. (In particular, you can work in pairs, where each pair submits just one paper.)

1. Recall that merge sort divides a sequence of n elements to sort into two halves, sorts the two halves recursively and then merges the two halves to get a fully sorted array. (For this question, assume the length of the array is a power of 2 for simplicity, so that we do not have to worry about rounding when we divide n by 2.)

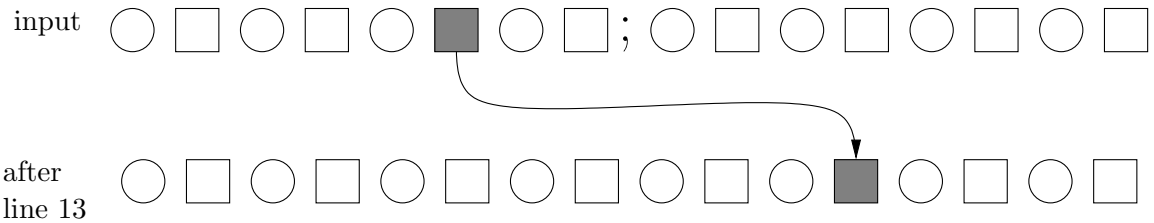
```
1: MergeSort(x_1, x_2, ..., x_n)
2:     if n > 1 then
3:         MergeSort(x_1, x_2, ..., x_{n/2})
4:         MergeSort(x_{n/2+1}, x_{n/2+2}, ..., x_n)
5:         Merge(x_1, x_2, ..., x_{n/2}; x_{n/2+1}, x_{n/2+2}, ..., x_n)
6:     Postcondition: x_1 ≤ x_2 ≤ ··· ≤ x_n
```

One disadvantage of this sorting algorithm is that the standard Merge algorithm requires using additional space. In this question we will consider a different implementation of the Merge algorithm that can be done without using any extra storage space. The new Merge algorithm uses a divide-and-conquer approach.

```
7: Merge(x_1, x_2, ..., x_{n/2}; x_{n/2+1}, x_{n/2+2}, ..., x_n)
8:     Precondition: x_1 ≤ x_2 ≤ ··· ≤ x_{n/2} and x_{n/2+1} ≤ x_{n/2+2} ≤ ... ≤ x_n
9:     if n = 2 and x_1 > x_2 then
10:        swap x_1 and x_2
11:    else if n > 2 then
12:        Merge(x_1, x_3, ..., x_{n/2-1}; x_{n/2+1}, x_{n/2+3}, ..., x_{n-1})   ▷ merge odd positions
13:        Merge(x_2, x_4, ..., x_{n/2}; x_{n/2+2}, x_{n/2+4}, ..., x_n)         ▷ merge even positions
14:        for i ← 1..n/2 − 1 do
15:            if x_{2i} > x_{2i+1} then
16:                swap x_{2i} and x_{2i+1}
17:    Postcondition: x_1 ≤ x_2 ≤ ··· ≤ x_n
```

[2] **(a)** Explain why the element in the leftmost position at the end of the Merge routine is less than or equal to every input value.

[5] **(b)** The following diagram shows the inputs to a call to Merge and the sequence after lines 12 and 13 have completed. In this example, $n = 16$. Odd and even positions are indicated by squares and circles, respectively.



Suppose the element in the shaded position of the top diagram was moved to the shaded position in the bottom diagram by the recursive call on line 13.

Redraw this diagram, adding one of the following symbols inside each of the circles and squares:

- $\leq$ if the element must be less than or equal to the shaded square,
- $\geq$ if the element must be greater than or equal to the shaded square,
- $=$ if the element must be equal to the shaded square, or
- ? if there is no way to know how the element compares to the shaded square.

Write a brief justification explaining why your answer is correct.

Hint: first think about exactly which squares in the input must be less than or equal to the shaded square.

[2] **(c)** After line 13, the only further adjustment to the shaded square of part (b) during the loop on lines 14–16 is that it might be swapped with the element to its right, if they are out of order. Explain why this results in the shaded element ending up in exactly the right place to satisfy the postcondition.

Remark: you could make a similar argument to show every *element ends up in the correct position at the end of* MERGE*, but I am not asking you to do that.*

[2] **(d)** Let $T(n)$ be the worst-case running time of MERGE on inputs of size n. Give a recurrence for $T(n)$ and use it to give good asymptotic bounds on $T(n)$. [1]

[2] **(e)** Let $S(n)$ be the worst-case running time of MERGESORT on inputs of size n if it uses the MERGE algorithm given above. Give a recurrence for $S(n)$ and use it to give good asymptotic bounds on $S(n)$. How does this compare to MERGESORT using the standard linear-time algorithm for performing MERGE?

[1] The way the pseudocode is written, it looks like you would have to pass many array elements as arguments to the functions. But if you actually implemented it, you would not do this. When sorting an array A, the elements passed to any function call would be of the form $A[f], A[f+d], A[f+2d], A[f+3d], \ldots, A[f+(n-1)d]$ so you would really only have to pass three integers f, d and n, together with a pointer to array A, as arguments. It is easy to compute these three integers for each recursive call to MERGESORT or MERGE in constant time. When computing worst-case times in parts (d) and (e), assume that the implementation of the algorithm uses this technique.