## Question 3:

1. (Submitted on canvas)
2. A screenshot of a computer

   Description automatically generatedA screenshot of a computer

   Description automatically generated
3. There are 2 loops running in the function of critical events. Outer loop is running n times, where n is the size of array and inner loop is also running n times. Hence, the total time complexity of the algorithm is .

## Question 4:

1. A screenshot of a computer

   Description automatically generated

b.

* 1. A screenshot of a computer

     Description automatically generated
  2. For my solution, the operations in the first line are constant-time operations. It works with the while loop employing a binary search strategy, which halves the search space on each iteration. This results in a logarithmic time complexity of O(logn). With each step, half of the remaining elements are removed by adjusting the `low` and/or `high` index based on each comparison between `target` and the middle element of the active search interval. This is an efficient method to narrow down the search range, which is, again the logarithmic style of the algorithm’s complexity. For proof of correctness, this is ensured by my loop invariants maintained throughout the binary search, which are:

1. If `target` value exists in the array, it will always be between the indices `low` and `high`.
2. The `low` index will always point to the first element greater than or equal to `target`, or if `target` is not found, to the insertion point when `high` becomes less than `low`.

So, the loop invariant confirms if `target` is within the array, it will definitely be found. In any other case, `low` indicates the correct insertion position. So the algorithm ensures the correct result in O(logn) time for any input.

* 1. (Submitted on canvas)

## Question 5:

a.

array of { 4, 3, 2, 1, 5 } to demonstrate bubble sort:

**Iteration # 1**

* Compare 4 and 3, swap {3, 4, 2, 1, 5}
* Compare 4 and 2, swap {3, 2, 4, 1, 5}
* Compare 4 and 1, swap {3, 2, 1, 4, 5}
* Compare 1 and 5, no swap {3, 2, 1, 4, 5}

After the first iteration, the largest number (5) is at the end.

**Iteration # 2**

* Compare 3 and 2, swap {2, 3, 1, 4, 5}
* Compare 1 and 3, swap {2, 1, 3, 4, 5}
* Compare 4 and 3, no swap {2, 1, 3, 4, 5}

After the first iteration, the largest number (4, 5) is at the end.

**Iteration # 3**

* Compare 1 and 2, swap {1, 2, 3, 4, 5}
* Compare 3 and 2, no swap {1, 2, 3, 4, 5}

After the first iteration, the largest number (3, 4, 5) is at the end.

**Iteration # 4**

* Compare 1 and 2, no swap {1, 2, 3, 4, 5}

After the first iteration, the largest number (2, 3, 4, 5) is at the end.

**Iteration # 5**

After the first iteration, the largest number (1, 2, 3, 4, 5) is at the end.

Hence, the array is sorted.

b.

**Formula for Comparisons**

* For n elements, n-1 comparisons are required.
* For n-1 elements, n-2comparisons are required.

…

…

…

* For 2 elements, 1 comparison is required.
* For 1 element, 0 comparisons are required.

Hence, we can conclude that total number of comparisons can be found by sum of n-1 numbers.

**Formula for Swaps**

Given that the array is reversed, as this is worse-case scenario; and the formula for it can be calculated same as C(n).

* For n elements, n-1 comparisons are required.
* For n-1 elements, n-2 comparisons are required.

…

…

…

* For 2 elements, 1 comparison is required.
* For 1 element, 0 comparisons are required.

Hence, we can conclude that total number of comparisons can be found by sum of n-1 numbers.

c.

**Initialization**

First, we need to determine whether the invariant is true before the first iteration. With i starting from 0, at the beginning of the 0th iteration, the largest 0 elements of the original list are in the last 0 positions of the list. This is true as it is not promising anything.

**Maintenance**

The largest i elements of the original list occupy the last i positions in the list and are sorted relative to each other. The inner loop executes from index 1 to len(li)- i - 1 (stopping before the last i elements). Look at the first two elements and swap them if they are out of order. The larger of them comes second. Again, with the second and third element, leave the maximum of the first three elements in the third position. This continues until the maximum of the first len(i) - i elements is in the len(li) - ith position, or the len(li) - i - 1 index. The largest i elements occupying the last i positions in sorted order is found, but now we also have the largest of the len(li) - i in position len(li) - i - 1, i.e. just before last i elements. It is not bigger than any of the last i elements, and at least as big as the other len(li) - i - 1 elements. So, the last i+1 elements are sorted with respect to each other, and are the largest i+1 elements.

**Termination**

The outer loop executed len(li) times, but it only did any work len(li) - 1 times. Hence, last len(li) - 1 elements are the largest len(li) - 1 elements and are sorted with respect to each other. That leaves one element left, which is at most the smallest element, and is in position 0. So, the entire list is sorted with respect to itself.

d.

Considering a random permutation, the probability of an element greater than other is ½. Similarly, the probability of swapping is also ½. The probability of swapping I and i+1 elements during a single pass becomes ½ \* ½ = ¼.

If an array has n elements, the probability of swapping each pair in an array will be:

¼ (n-1)

Bubble sort makes n-1 passes to fully sort the array, hence the probability of swapping by n-1 number of passes can be given by:

Hence, this is the formula to find the average number of swaps in bubble sort.