Pseudocode:

```
sort_algorithm(array)
        ASSERT array is a list
        SET array_length <- length(array)
        ASSERT array_length > 0
        FOR i <- (array_length - 1, 0, -1)
                SET largest_element <- 0
                FOR j <- (1, i + 1)
                        IF array[j] > array[largest_element]
                                largest_element <- j
                ASSERT largest_element < array_length
                ASSERT i < array_length
                SET array[largest_element], array[i] <- array[i], array[largest_element]
        ASSERT array is sorted correctly
        RETURN array


find_missing_number(sorted_array)
        ASSERT sorted_array is a list
        ASSERT array_length > 1
        ASSERT sorted_array is sorted correctly
        SET array_length <- length(sorted_array)
                FOR i <- (0, array_length - 1)
                        IF sorted_array[i + 1] - sorted_array[i] != 1
                                RETURN sorted_array[i] + 1
        RETURN None


main()
        GET array
```

```
    SET array <- sort_algorithm(array)
    SET missing_number <- find_missing_number(array)
    IF missing_number == None
        PUT "There is no missing number"
    ELSE
        PUT "The missing number is: {missing_number}"

CALL main()
```

Efficiency:

We will start with the sort_algorithm function. The algorithmic efficiency of this function is not great at all, but it gets the job done. The outer loop runs from array_length - 1 to 0, which means it will run n times (n being the length of the array), giving it an efficiency of O(n). The inner loop runs from 1 to i + 1, meaning it will also run n times, which gives the inner loop a time efficiency of O(n). Inside the inner loop there is a swapping statement that is O(1) as it is simply setting some variables equal to values, but the overall efficiency is absolutely dominated by the inner and outer loops. This means that the overall efficiency of the function is O(n) x O(n), which results in a not great O(n^2).

The other part of the pseudocode that needs an efficiency assessment is the find_missing_number function. This function iterates through the array one time in order to check if there is a missing number, so the function has an algorithmic efficiency of O(n).

Understandability:

I would assess my code as being straightforward. Any programmer will be able to look at it and understand what it is doing after just a few moments because everything is pretty clear. Unfortunately, the complexity of the sort_algorithm function makes it so that it isn't immediately obvious to a programmer as to what is happening, even if the find_missing_number function is obvious in what it is doing because it is much more simple. All of the pieces are there and clearly stated and there isn't much room for doubt and confusion, but I definitely wouldn't say that there is no room for doubt or confusion. Thus, I determine this code to be straightforward.

Malleability:

I would say that my program is refactorable. There are no outside configuration files (getting the array from the user doesn't count) and any changes to the logic or functionality of the program would require altering the logic of the functions in major ways.

Trace:

For this trace, let's use the following 4-item array: [4, 1, 3, 5]

Line	array	array_length	largest_element	i	j	missing_number
27	[4, 1, 3, 5]	/	/	/	/	/
3	[4, 1, 3, 5]	4	/	/	/	/
5	[4, 1, 3, 5]	4	/	3	/	/
6	[4, 1, 3, 5]	4	0	3	/	/
7	[4, 1, 3, 5]	4	0	3	1	/
7	[4, 1, 3, 5]	4	0	3	2	/
7	[4, 1, 3, 5]	4	0	3	3	/
9	[4, 1, 3, 5]	4	3	3	3	/
12	[4, 1, 3, 5]	4	3	3	3	/
5	[4, 1, 3, 5]	4	3	2	3	/
6	[4, 1, 3, 5]	4	0	2	3	/
7	[4, 1, 3, 5]	4	0	2	1	/
7	[4, 1, 3, 5]	4	0	2	2	/
12	[1, 3, 4, 5]	4	0	2	1	/
5	[1, 3, 4, 5]	4	0	1	1	/
6	[1, 3, 4, 5]	4	0	1	1	/
7	[1, 3, 4, 5]	4	0	1	1	/
9	[1, 3, 4, 5]	4	1	1	1	/
12	[1, 3, 4, 5]	4	1	1	1	/
28	[1, 3, 4, 5]	4	1	1	1	/
20	[1, 3, 4, 5]	4	1	1	1	/
21	[1, 3, 4, 5]	4	1	0	1	/
29	[1, 3, 4, 5]	4	1	0	1	2

The missing number was 2.