Homework 6: Sorting and Merging Rows of a Matrix

In this assignment, you will implement three sorting algorithms—Bubble Sort, Insertion Sort, and Selection Sort—and apply them to the rows of a matrix (2D list) of size `3 x n` (3 rows and n columns). After sorting each row, you will merge the sorted rows into one sorted 1D list.

Part 1: hw6.py

You have a matrix of size `3 x n`, where each row needs to be sorted using a specific sorting algorithm:

- The **first row** should be sorted using **Bubble Sort**.
- The **second row** should be sorted using **Insertion Sort**.
- The **third row** should be sorted using **Selection Sort**.

You are required to implement three functions as follows:

1. `bubble_sort(matrix) -> sorted_first_row, n_swaps` :

- Takes the matrix as input.
- Sorts the first row using Bubble Sort.
- Returns the **sorted first row list** and the **number of swaps** performed.

2. `insertion_sort(matrix) -> sorted_second_row, n_swaps`:

- Takes the matrix as input.
- Sorts the second row using Insertion Sort.
- Returns the **sorted second row list** and the **number of swaps** performed.

3. `selection_sort(matrix) sorted_third_row, n_swaps`:

- Takes the matrix as input.
- Sorts the third row using Selection Sort.
- Returns the **sorted third row list** and the **number of swaps** performed.
- Aim to minimize the number of swaps performed: don't perform the swap unless you need to.

Your implementation should aim to be as efficient as possible, with no unnecessary comparison steps and no swaps if not needed. Additionally, you should ensure that both bubble sort and insertion sort maintain `O(n)` time complexity in the best case. The number of swaps in the best case for insertion sort should be O(1).

`merge(first_row, second_row, third_row)` is provided to you. The function takes the 3 sorted lists as input, then merges and return the three lists into one sorted 1D list. The merging is done efficiently in `O(n)` time complexity, where `n` is the total number of elements in each row of the matrix.

Example:

```
# Example:
    matrix = [
        [10, 2, 3, 8, 1],    # First row (Bubble Sort)
        [7, 15, 14, 20, 9],  # Second row (Insertion Sort)
        [25, 12, 5, 3, 16]   # Third row (Selection Sort)
```

```
    ]

    # Step 1: Sort the first row using Bubble Sort
    sorted_1_bubble, bubble_swaps = bubble_sort(matrix)
    print("After Bubble Sort:", sorted_1_bubble, "Swaps:", bubble_swaps)

    # Step 2: Sort the second row using Insertion Sort
    sorted_2_insertion, insertion_swaps = insertion_sort(matrix)
    print("After Insertion Sort:", sorted_2_insertion, "Swaps:", insertion_swaps)

    # Step 3: Sort the third row using Selection Sort
    sorted_3_selection, selection_swaps = selection_sort(matrix)
    print("After Selection Sort:", sorted_3_selection, "Swaps:", selection_swaps)

    # Step 4: Merge the sorted rows into one sorted list
    merged_list = merge(sorted_1_bubble, sorted_2_insertion, sorted_3_selection)
    print("Merged List:", merged_list)
```

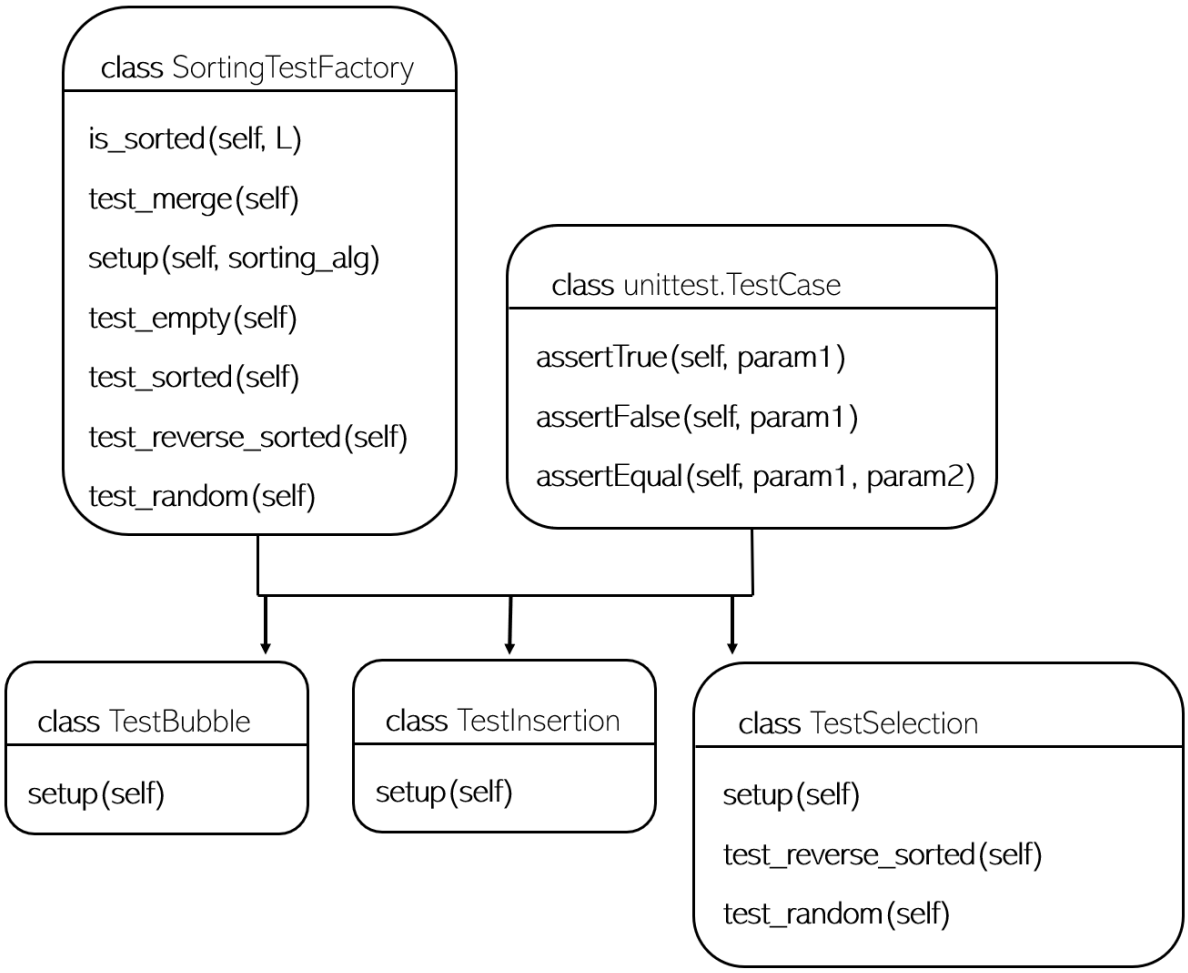
Expected output:

```
After Bubble Sort: [1, 2, 3, 8, 10] Swaps: 7
After Insertion Sort: [7, 9, 14, 15, 20] Swaps: 4
After Selection Sort: [3, 5, 12, 16, 25] Swaps: 3
Merged List: [1, 2, 3, 3, 5, 7, 8, 9, 10, 12, 14, 15, 16, 20, 25]
```

Part 2: Building Test Factory for Sorting Algorithms

In this part of the homework, you will build a test factory to test the sorting algorithms implemented in Part 1. The test factory should include test cases for an empty lists, a sorted lists, a reverse sorted lists, and a randomly shuffled lists. By using a test factory, you can avoid repeating the same tests for different sorting algorithms.

Below is a graphical representation of how to structure the test factory:



A test factory is a design pattern commonly used in software testing to streamline the creation of test cases and promote code reuse. It encapsulates the logic for generating test cases, making it easier to manage and maintain test suites, especially when dealing with multiple test scenarios or similar tests for different functions or classes.

In the context of sorting algorithms, a test factory can be particularly useful for generating test cases to verify the correctness and performance of different sorting implementations. Instead of manually writing test cases for each sorting algorithm, a test factory allows us to define common test scenarios (e.g., empty lists, sorted list, reverse sorted list, random list) once and reuse them across multiple sorting algorithms.

It's important to note that when using a test factory, certain test cases may need to be overridden or modified for specific classes or functions. For example, in the case of sorting algorithms like bubble sort and insertion sort, the test cases may be similar. However, for algorithms like selection sort, which works differently, some test cases in the TestSelection class may need to be overridden or adapted to accommodate these differences. This ensures that the test cases accurately reflect the behavior of each sorting algorithm being tested.

Submission

Submit the following files:

- `hw6.py`
- `test_hw6.py`

Students must submit to Gradescope individually within 24 hours of the due date (homework due dates are typically Tuesday at 11:59 pm EST) to receive credit.