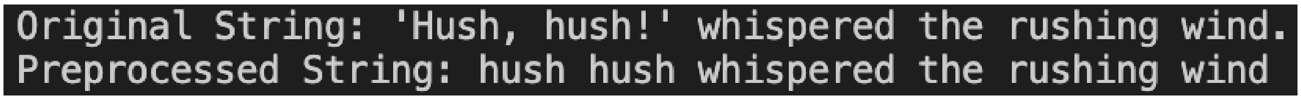
Data Structures and Algorithms

HOMEWORK 7 REPORT

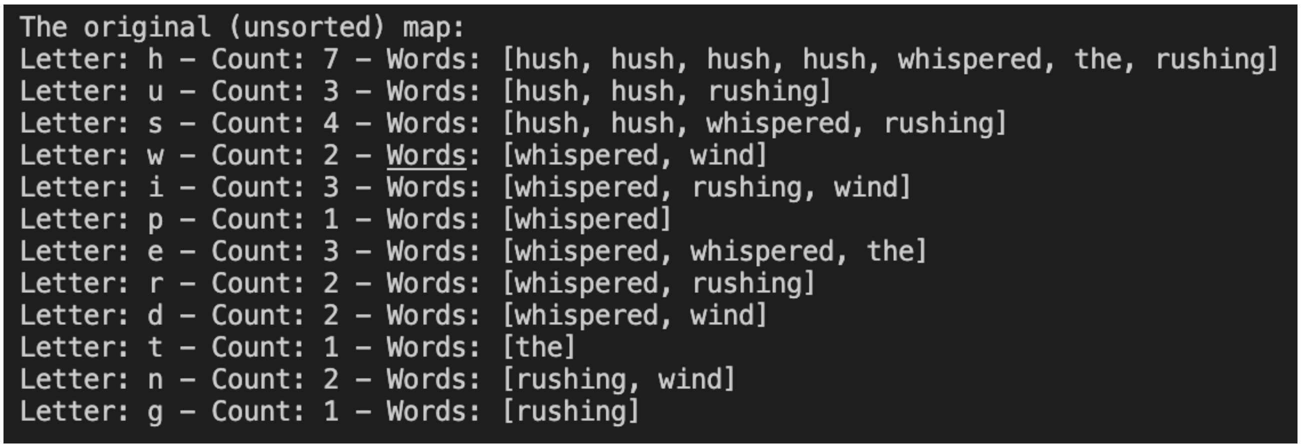
RUNNING COMMANDS AND RESULTS

original and preprocessed input:



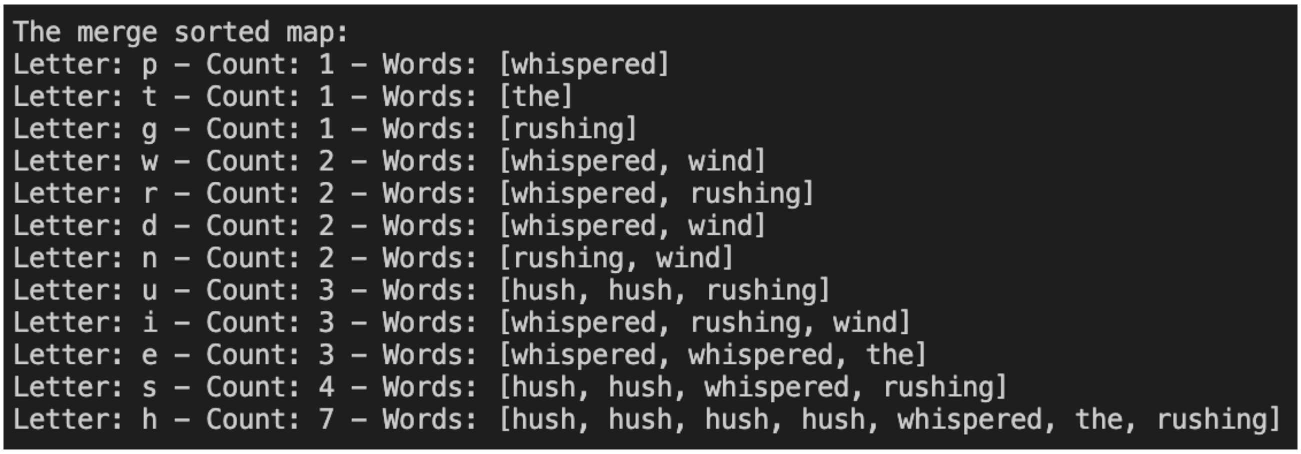
```
Original String: 'Hush, hush!' whispered the rushing wind.
Preprocessed String: hush hush whispered the rushing wind
```

original (unsorted) map:



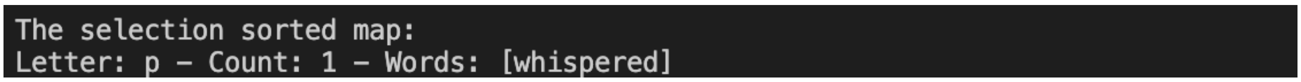
```
The original (unsorted) map:
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: p - Count: 1 - Words: [whispered]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: t - Count: 1 - Words: [the]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: g - Count: 1 - Words: [rushing]
```

merge sorted map:

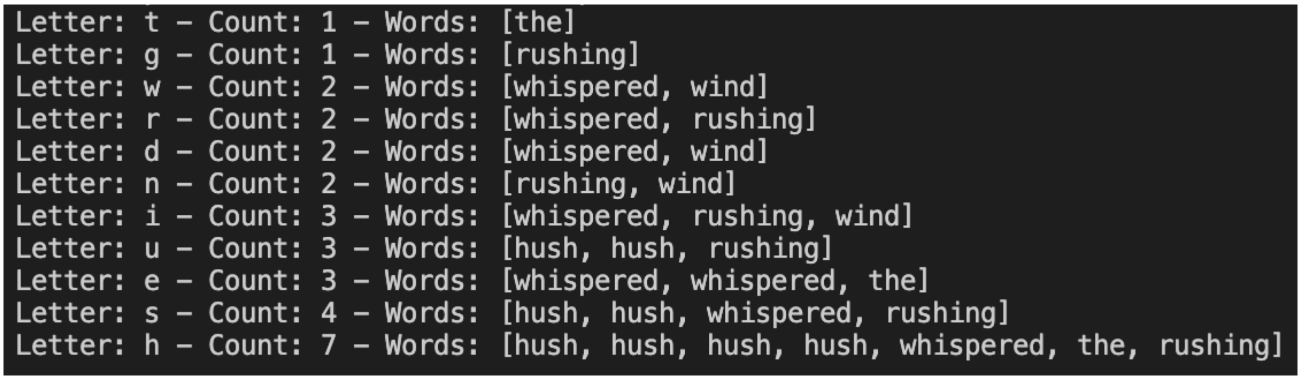


```
The merge sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

selection sorted map:

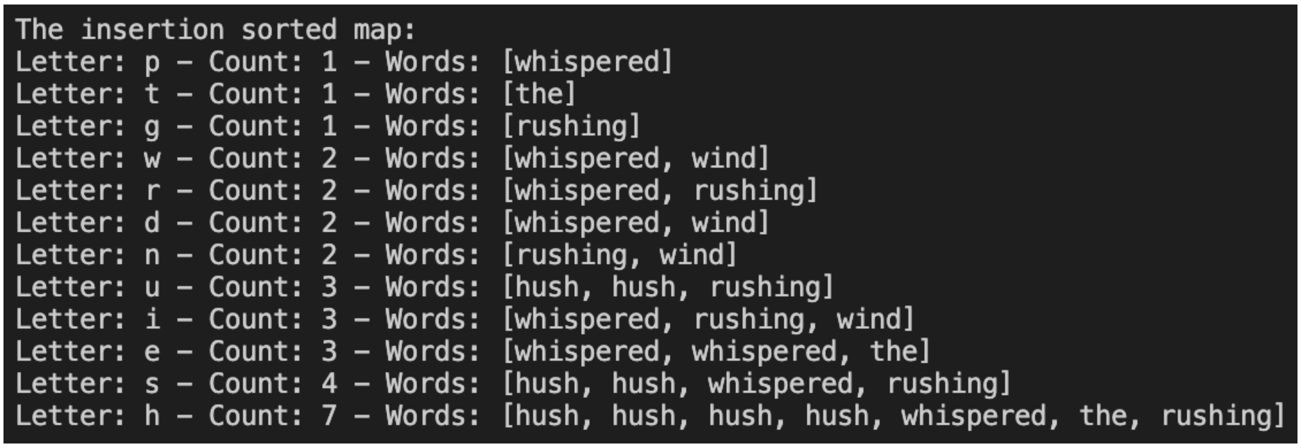


```
The selection sorted map:
Letter: p - Count: 1 - Words: [whispered]
```



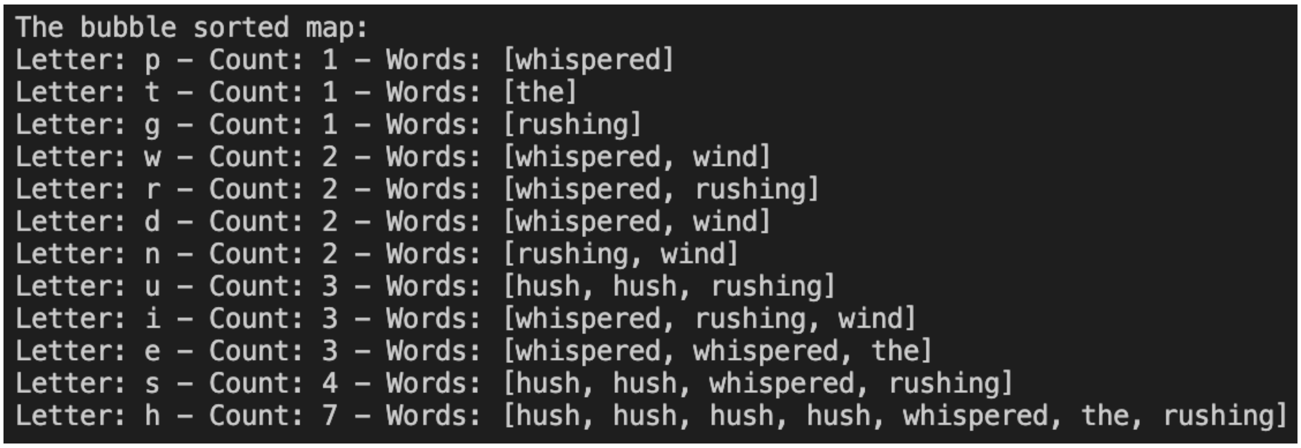
```
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

insertion sorted map:



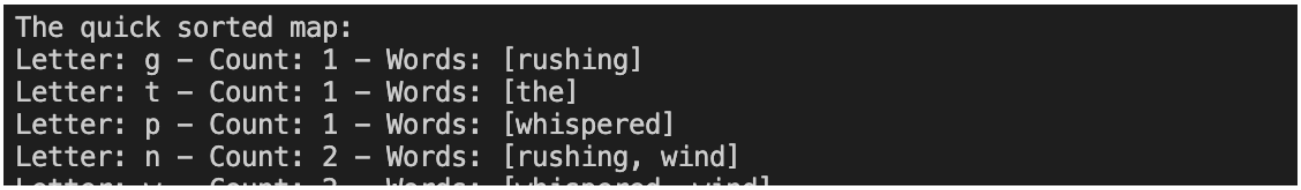
```
The insertion sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

bubble sorted map:

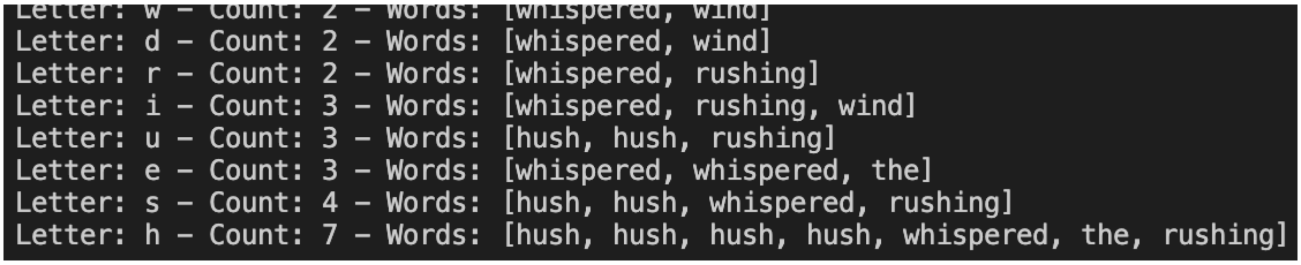


```
The bubble sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

quick sorted map:



```
The quick sorted map:
Letter: g - Count: 1 - Words: [rushing]
Letter: t - Count: 1 - Words: [the]
Letter: p - Count: 1 - Words: [whispered]
Letter: n - Count: 2 - Words: [rushing, wind]
```



```
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

TIME COMPLEXITY ANALYSIS

Merge Sort

Best-case: The best-case scenario occurs when the input array is already sorted. So, the best-case time complexity is **O(n log n)**, where n is the size of the input array.

Average-case: The average-case time complexity of this algorithm is **O(n log n)**, where n is the size of the input array. The recursive sort function divides the array into two parts repeatedly until each subarray has only one element. The merge function then merges the sorted subarrays, which takes O(n) time in the worst case.

Worst-case: The worst-case time complexity of this algorithm is also **O(n log n)**. It occurs when the input array is in reverse sorted order.

Selection Sort

Best-case: The best-case scenario occurs when the input array is already sorted. So, the best-case time complexity is **O(n)**, where n is the size of the input array.

Average-case: The average-case time complexity of this algorithm is **O(n^2)**, where n is the size of the input array. The algorithm consists of two nested loops. On average, the inner loop performs approximately n/2 iterations. As a result, the average number of comparisons and swaps is roughly (n^2)/2.

Worst-case: The worst-case time complexity of this algorithm is also **O(n^2)**. It occurs when the input array is in reverse sorted order.

Insertion Sort

Best-case: The best-case scenario occurs when the input array is already sorted. So, the best-case time complexity is **O(n)**, where n is the size of the input array.

Average-case: The average-case time complexity of this algorithm is **O(n^2)**, where n is the size of the input array. On average, the inner loop performs approximately (nextPos - 1) / 2 iterations. By summing up the iterations over all the outer loop iterations, we get the average time complexity of O(n^2).

Worst-case: The worst-case time complexity of this algorithm is **O(n^2)**. It occurs when the input array is in reverse sorted order.

Bubble Sort

Best-case: The best-case scenario occurs when the input array is already sorted. So, the best-case time complexity is **O(n)**, where n is the size of the input array.

Average-case: The average-case time complexity of this algorithm is **O(n^2)**, where n is the size of the input array. On average, the inner loop performs approximately (n - i - 1) / 2 iterations. By summing up the iterations over all the outer loop iterations, we get the average time complexity of O(n^2).

Worst-case: The worst-case time complexity of this algorithm is **O(n^2)**. It occurs when the input array is in reverse sorted order.

Quick Sort

Best-case: The scenario happens when the pivot selected during each partitioning step divides the array into two roughly equal parts. So, the algorithm performs well and achieves a balanced partition at each level of recursion. The best-case time complexity is **O(n log n)**, where n is the size of the input array. This occurs when the pivot element consistently divides the array into two equal-sized subarrays.

Average-case: The average-case time complexity of this algorithm is **O(n log n)**, where n is the size of the input array. On average, the partitioning step divides the array into two subarrays of roughly equal size, resulting in a balanced partition at each level of recursion.

Worst-case: The worst-case time complexity of this algorithm is **O(n^2)**, where n is the size of the input array. The worst-case occurs when the pivot selection consistently leads to highly unbalanced partitions, such as when the pivot is always the smallest or largest element. This can lead to a worst-case time complexity of O(n^2).

RUNNING TIME OF EACH SORTING ALGORITHM

sorts \ cases	**best-case**	**average-case**	**worst-case**
merge sort	0.004285875 ms	0.004302958 ms	0.004598292 ms
selection sort	0.003862209 ms	0.004025917 ms	0.004313709 ms
insertion sort	0.003563041 ms	0.004249208 ms	0.004972209 ms
bubble sort	0.003987208 ms	0.004084583 ms	0.004186375 ms
quick sort	0.003622458 ms	0.003634542 ms	0.003665875 ms

COMPARISON OF SORTING ALGORITHMS

According to given inputs, for small data size selection, insertion and bubble sort is more efficient than quick sort and merge sort. Quick sort and merge sort's best case complexities not suitable for given inputs but for worst-case scenario more suitable than other sort algorithms.

ANALYZING OF INPUT ORDERING AND CODE SNIPPET

Merge, insertion and bubble sorts preserve input ordering. Because these algorithms are stable. They maintain the relative order of equal elements.

On the contrary, selection and quick sorts don't preserve input ordering. Selection sort is an in-place comparison-based sorting algorithm that repeatedly selects the minimum element and places it at the beginning of the unsorted portion of the list. Also , quick sort is an in-place sorting algorithm that relies on the selection of a pivot element and partitioning of elements around it.

Merge Sort achieves the preservation of input ordering by recursively dividing the array and merging the sorted sub-arrays. If two elements have the same value, the algorithm ensures that the element from the left sub-array takes precedence, maintaining the original input ordering.

The swapping step in the Selection Sort algorithm, when finding the minimum element in the unsorted portion, it is swapped with the element at the current iteration index. This swapping disrupts the original ordering of equal elements.

Insertion Sort works by iteratively inserting each element into its correct position in the sorted portion of the list. The algorithm compares each element with the elements before it and shifts them if necessary to make space for the current element. This shifting ensures that equal elements maintain their relative order and the original input ordering is preserved.

Bubble Sort compares contiguous elements and swaps if necessary. If two contiguous elements are equal, the swap operation can change their relative order. However, subsequent passes of Bubble Sort will restore the original input ordering.

In quick sort, depending on the choice of the pivot, the partitioning process can reorder elements. Elements equal to the pivot can end up on either side of the partition, disrupting the original input ordering.