Hw7 Report Eren Torlak 210104004090 CSE 222

Best, average, and worst-case time complexities analysis of each sorting algorithm:

1. Merge Sort:
    - Best-case time complexity: O(n log n)
    - Average-case time complexity: O(n log n)
    - Worst-case time complexity: O(n log n)
    - Explanation: Merge Sort has a consistent time complexity of O(n log n) for all cases. It divides the input into smaller subarrays, recursively sorts them, and then merges them back together in sorted order
2. Selection Sort:
    - Best-case time complexity: O(n^2)
    - Average-case time complexity: O(n^2)
    - Worst-case time complexity: O(n^2)
    - Explanation: Selection Sort scans the entire unsorted part of the array in each iteration, selecting the smallest element and swapping it with the current position. İt is always Works same.
3. Insertion Sort:
    - Best-case time complexity: O(n)
    - Average-case time complexity: O(n^2)
    - Worst-case time complexity: O(n^2)
    - Explanation: In the average and worst cases, it requires shifting elements to their correct positions, leading to a quadratic time complexity. When the array is already sorted and we don't need to shift any elements
4. Bubble Sort:
    - Best-case time complexity: O(n)
    - Average-case time complexity: O(n^2)
    - Worst-case time complexity: O(n^2)
    - Explanation: Bubble Sort compares adjacent elements and swaps them if they are in the wrong order, repeatedly iterating through the array until no more swaps are needed. In the best case where the input is already sorted, it has a linear time complexity of O(n)
5. Quick Sort:
    - Best-case time complexity: O(n log n)
    - Average-case time complexity: O(n log n)
    - Worst-case time complexity: O(n^2)
    - Explanation: Quick Sort partitions the array into smaller subarrays based on a pivot element, recursively sorts the subarrays, and combines them to achieve a sorted array. In the worst case where the pivot is consistently chosen as the smallest or largest element, it is O(n^2).

Hw7 Report Eren Torlak 210104004090 CSE 222

- 2.)
- Worst-case Input: dddd ccc bb a
- Preprocessed String: dddd ccc bb a
- Merge Sort:
- Time taken: 514400 nanoseconds
- Selection Sort:
- Time taken: 44700 nanoseconds
- Insertion Sort:
- Time taken: 57000 nanoseconds
- Bubble Sort:
- Time taken: 51300 nanoseconds
- Quick Sort:
- Time taken: 92100 nanoseconds

- All the sorted maps are the same

- Average-case Input: abc abcde asvs
- Preprocessed String: abc abcde asvs
- Merge Sort:
- Time taken: 38400 nanoseconds
- Selection Sort:
- Time taken: 46500 nanoseconds
- Insertion Sort:
- Time taken: 44300 nanoseconds
- Bubble Sort:
- Time taken: 34800 nanoseconds
- Quick Sort:
- Time taken: 29000 nanoseconds

- All the sorted maps are the same

- Best-case Input: ab
- Preprocessed String: ab
- Merge Sort:
- Time taken: 11300 nanoseconds
- Selection Sort:
- Time taken: 19000 nanoseconds
- Insertion Sort:
- Time taken: 12100 nanoseconds
- Bubble Sort:
- Time taken: 21500 nanoseconds
- Quick Sort:
- Time taken: 15100 nanoseconds

- All the sorted maps are the same

Hw7 Report Eren Torlak 210104004090 CSE 222

Comparison of Sorting Algorithms:

- Based on the time complexities, Merge Sort and Quick Sort have the best overall performance with an average time complexity of O(n log n). They are expected to be faster than the other algorithms for large inputs. But some times it does not look like that. There may be problem in timer.
- When first case is happening always it takes longer. Timer is not working good.
- In the worst-case scenario where the frequencies of the letters are varied and the input is more complex, Selection Sort performs the best with the lowest running time.reason for that may be about input is small.

d) SelectionSort preserves input order.

- The idea is to repeatedly find the minimum key in the unsorted portion and swap it with the FİRST unsorted key.

```
 public void sortMap() {
        ArrayList<String> keys = new ArrayList<>(originalMap.getMap().keySet());
        int keysSize = keys.size();
        for (int i = 0; i < keysSize - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < keysSize; j++) {
                if (originalMap.getMap().get(keys.get(j)).getCount() < originalMap.getMap().get(keys.get(minIndex)).getCount()) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                swap(keys, i, minIndex);
            }
        }
        for (String key : keys) {
            sortedMap.getMap().put(key, originalMap.getMap().get(key)); // add the sorted keys to the sorted map
        }
    }
```

Hw7 Report Eren Torlak 210104004090 CSE 222

Insertion Sort: preserves input order:

- The idea is to iteratively insert each element into its correct position in the sorted part of the array.
- İt compares with " > " that makes input order preserved.

```
    public void sortMap() {
        ArrayList<String> keys = new
ArrayList<>(originalMap.getMap().keySet());
        int keysSize = keys.size();

        // Iterate over the keys starting from the second element
        for (int currentIndex = 1; currentIndex < keysSize; currentIndex++) {
            String currentKey = keys.get(currentIndex);
            int previousIndex = currentIndex - 1;

            // Shift elements to the right until the correct position is found
for the
            // current key
            while (previousIndex >= 0
                    && (originalMap.getCount(keys.get(previousIndex)) >
originalMap.getCount(currentKey))) {
                keys.set(previousIndex + 1, keys.get(previousIndex)); // Shift
the element to the right
                previousIndex--; // Move to the previous index
            }
            keys.set(previousIndex + 1, currentKey);
        }

        for (String key : keys) {
            sortedMap.getMap().put(key, originalMap.getMap().get(key)); // Add
the sorted keys to the sorted map
        }
    }
```

Hw7 Report Eren Torlak 210104004090 CSE 222

Bubble sort preserves input order:

The idea is to repeatedly swap adjacent keys if they are in the wrong order until the entire array is sorted.

İt is stable because we are not swapping equal count valued elements

```
public void sortMap() {
        ArrayList<String> keys = new
ArrayList<>(originalMap.getMap().keySet());
        int keySize = keys.size();
        String keyLeft;
        String keyRight;
        for (int i = 0; i < keySize - 1; i++) {
            boolean isSwapped = false; // If no swaps are made in a pass, the
array is already sorted
            for (int j = 0; j < keySize - i - 1; j++) {
                keyLeft = keys.get(j);
                keyRight = keys.get(j + 1);

                if (originalMap.getMap().get(keyLeft).getCount() >
originalMap.getMap().get(keyRight).getCount()) {
                    swap(keys, j, j + 1);
                    isSwapped = true;
                }
            }
            if (!isSwapped) { // this makes time complexity O(n) for an
already sorted array
                break;
            }
        }
        for (String key : keys) {
            sortedMap.getMap().put(key, originalMap.getMap().get(key)); // add
the sorted keys to the sorted map
        }
    }
```

Hw7 Report Eren Torlak 210104004090 CSE 222

My Quick Sort preserves input order:

- Quick Sort partitions the array into smaller subarrays based on a pivot element, recursively sorts the subarrays, and combines them to achieve a sorted array.
- we dont swap equal count elements so it is stable.

```
private int partition(ArrayList<String> keys, int low, int high) {
        String pivot = keys.get(high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            String key = keys.get(j);
            if (originalMap.getMap().get(key).getCount() <
originalMap.getMap().get(pivot).getCount()) {
                i++;
                swap(keys, i, j);
            }
        }
        swap(keys, i + 1, high);
        return i + 1;
    }
```