GTU Department of Computer Engineering

CSE 222/505 - Spring 2023

Homework #7 Report

ÖMER SARIÇAM

200104004009

A)Time Complexity Analysis

	BEST	AVERAGE	WORST
Merge sort	O(n log n)	O(n log n)	O(n log n)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Quick Sort	O(n log n)	O(n log n)	O(n^2)

The time complexity of Merge Sort is O(n log n). This is because the complexity of the merge method is O(n) (The merging process takes linear time since it compares and combines the two halves.), and this method is recursively called by dividing the array in half, which takes O(log n) operations. As a result, the total time complexity becomes O(n log n) since the merge operation is performed log n times on an array of size n. Both the worst case and the best case and average case scenarios have this time complexity.

In each iteration, Selection Sort scans the remaining unsorted portion to find the minimum element and places it in its proper position. This process requires scanning the remaining unsorted portion each time, resulting in nested loops and a quadratic time complexity. No matter how the array is arrayed, it will still be the same time complexity.

While the best-case complexity of the Insertion Sort algorithm can be O(n), the average and worst-case complexities are considered to be O(n^2). This depends on the state of the array to be sorted. If just one comparison is made for every swap, the time complexity becomes O(n) because only one comparison is made for each element in the array. However, in the worst-case scenario, all elements before the pivot element are compared, resulting in a time complexity of O(n^2).

In Bubble Sort, if the elements are already completely sorted, the time complexity is O(n). This is because the array is traversed only once, and since no two elements are swapped, it is evident that the array is already sorted, and the loop is terminated. If the array is sorted in reverse order, then it becomes the worst-case scenario. In this case, the loop runs until the end. If the array is neither completely sorted nor sorted in reverse order, then it becomes the average case scenario.

In the best and average cases, Quick Sort exhibits a time complexity of O(n log n). This occurs when the pivot element is chosen effectively, leading to balanced partitions and efficient sorting. The algorithm divides the array into sub-arrays recursively, with each partition being approximately half the size of the previous partition. This balanced division results in a logarithmic number of divisions and a time complexity of O(n log n). However, in the worst-case scenario, Quick Sort's time complexity can deteriorate to O(n^2). This occurs when the pivot element is consistently chosen as the smallest or largest element, leading to highly imbalanced partitions. In this case, each

partition only reduces the size of the sub-array by one element, resulting in a quadratic time complexity.

I provided a sorted input (with count values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10) as the best-case scenario for Merge Sort, Selection Sort, Insertion Sort, and Bubble Sort. For the worst-case scenario, I provided a reversed input (with count values 10, 9, 8, 7, 6, 5, 4, 3, 2, 1). As for the average-case scenario, I provided a mixed input (with count values 5, 2, 9, 1, 7, 4, 6, 3, 8, 10).

However, since Quick Sort behaves differently from the others, I provided different inputs for its worst and best cases. In Quick Sort, I selected the first element of the array as the pivot. Consequently, if the array is already sorted, it results in the most inefficient scenario due to consistently unbalanced partitioning. Thus, I provided a sorted input as the worst-case scenario (with count values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10). For the best-case scenario, I created a specific order where the first element is always greater than or equal to the second half of the array (with count values 5, 2, 1, 3, 4, 8, 6, 7, 9, 10).

B) Running Time of Each Input

	BEST CASE	AVERAGE	WORST
Merge sort	236824.0	265426.0	332017.0
Selection Sort	206014.0	196390.0	249670.0
Insertion Sort	156076.0	163049.0	201505.0
Bubble Sort	136634.0	131777.0	166992.0
Quick Sort	103234.0	118647.0	156089.0

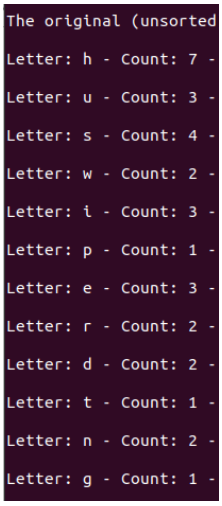
(nanoseconds)

C) Comparison of The Sorting Algorithms

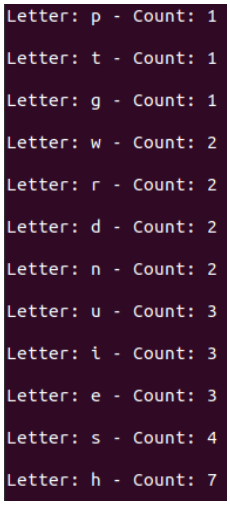
Quick Sort proved its worth and demonstrated that it is faster in all scenarios. Bubble Sort showed relative efficiency compared to Selection Sort and Insertion Sort, particularly with small inputs. However, I was surprised to see that Merge Sort consistently lagged behind the others in all cases. Nevertheless, as the number of inputs increases, Merge Sort will become more efficient. In this scenario, Merge Sort becomes an inefficient sorting algorithm for a small number of inputs due to its constant function calls and array creation.

D) Which Algorithms Keep The Input Ordering

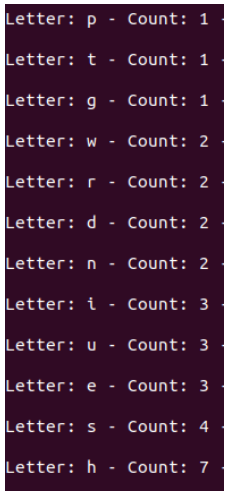
1)Order of insertion of elements:
2) merge sort
3) selection sort
4) insertion sort
5) bubble sort
6) quick sort



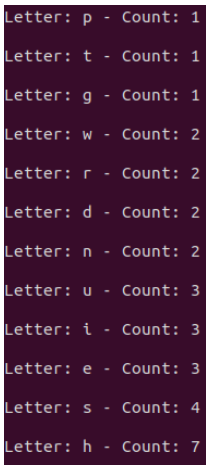
1



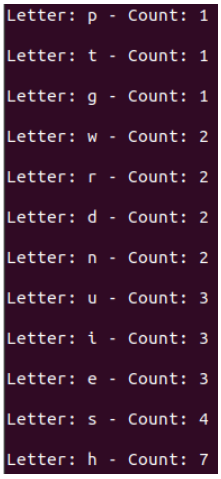
2



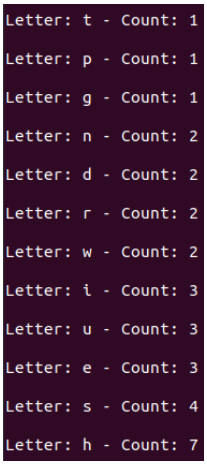
3



4



4



6

- As we can see from the outputs, in Merge Sort, Insertion Sort, and Bubble Sort, the input order is preserved. However, in Selection Sort and Quick Sort, the order of inputs can change during the sorting process.

- In Selection Sort, only the order of the letters 'i' and 'u' is not preserved. The reason for this is as follows: In Selection Sort, starting from the beginning of the array, each element swaps with the smallest element that comes after it. The order of the counts was as follows: 734231322121, corresponding to the letter array: "huswiperdtng". First, we start with the number 7, and since the smallest number in the remaining part is 1, 1 and 7 swap places, resulting in the array: 134237322121, "puswiherdtng". Then we move to index 1, which is the value 3. Again, the smallest number in the remaining part is 1, so 1 and 3 swap places, resulting in the array: 114237322321, "ptswiherdung". As we can see, here the letter 'u' started to come after the letter 'i'. However, the letter 'u' had already been added to the map earlier. Therefore, the input order got disrupted. In other words, there is no rule that the input order will be preserved in Selection Sort.

- If we look at the 6th image, we can see that there is more disruption in the input order in Quick Sort. The reason for this is as follows: In Quick Sort, the first element is chosen as the pivot, and the array is rearranged so that elements smaller or equal to the pivot are placed to the right, and larger elements are placed to the left. Then, the elements to the right and left are recursively subjected to the same process. This means that elements are constantly swapping positions, and the input order is not preserved. For example, the initial state of the array was: 734231322121, and the corresponding letter array was: "huswiperdtng". When we apply Quick Sort, we choose the first element as the pivot, which is 7. Since all the remaining elements are smaller than 7, they are placed at the end of the array, swapping positions with the element at the end, resulting in: 134231322127,

"guswiperdtnh". As we can see, the letter 'g' moved to the beginning of the array in the first operation, and this process continues until the end. However, 'g' had been added to the map after letters 'p' and 't' with the same count value. But it ended up at the beginning of the new map, so the input order was not preserved.