CSE222 / BİL505
Data Structures and Algorithms
Homework #6 – Report

Elifnur Kabalcı

1) Selection Sort

Time Analysis	O(n^2)- Because of 2 inner for loop Under all circumstances, swap and comparison counter are the same. It performs certain operations in every case, even if the list is ordered.
Space Analysis	O(1) – constant time – There is no new memory area created, There is only min value created 1 time, This means that this is constant. Although time analysis is the worst, space analysis also has the best numerical value.

2) Bubble Sort

Time Analysis	O(n^2) – Because of 2 inner for loop Since it uses a binary for loop, swap and comparison counter values are generally high. However, since I ended the loop with the flag when no swap was made, this was the one that gave the best results in sorted lists.
Space Analysis	O(1) – constant time There is only flag value created 1 time. Every time we call sort, the flag value is kept for use in the method. No other memory space is used. So it has the best numerical space complexity value.

3) Quick Sort

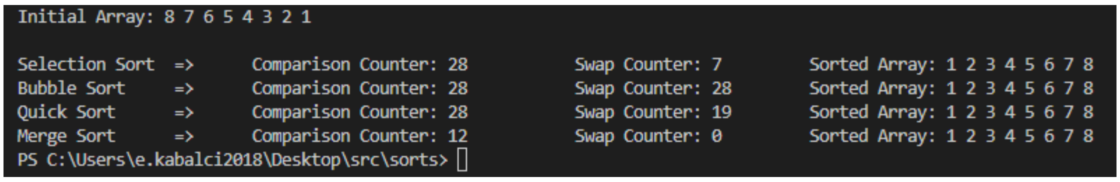
Time Analysis	O(nlogn) – Sort method divide arr into 2 pieces and call again(logn), in every partition calling run for(n). Overall, in every sort calling also call partititon. This means nlogn.
Space Analysis	O(logn) – Memory divides into 2 pieces in every sort calling. With the Partition method, the same memory area is divided into two each time the sort method is called and the method is called again. That's why it has the value logn. In complex lists, the comparison value was better than others, while in others, the results were poor both in comparison and in themselves.

4) Merge Sort

Time Analysis	O(nlogn) – Sort method divide arr into 2 pieces and call again(logn), in merge calling run for(n). Overall nlogn. Since it is based on the divide and conquer method, the comparison counter value is approximately the same for each condition. Since we scanned the array in 2 parts, time complexity gave a better result than n^2.

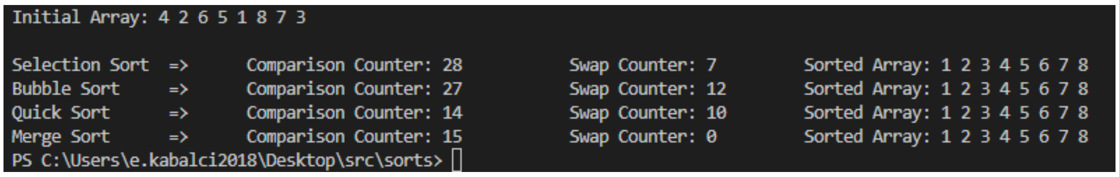
Space Analysis	O(n)Merge method takes 2 new temp memory location in every loop. 2n mean n. Since it does not use the swap method, it is the worst sorting algorithm in terms of space analysis.

General Comparison of the Algorithms



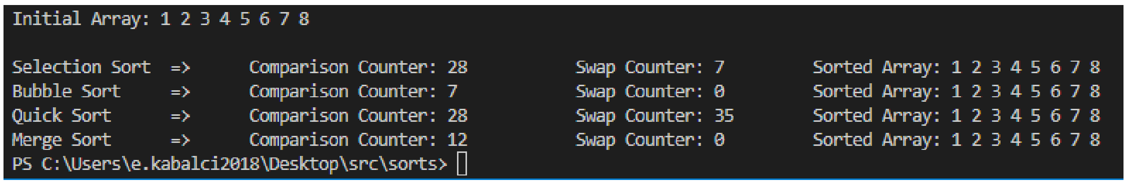
```
Initial Array: 8 7 6 5 4 3 2 1

Selection Sort  =>      Comparison Counter: 28          Swap Counter: 7         Sorted Array: 1 2 3 4 5 6 7 8
Bubble Sort     =>      Comparison Counter: 28          Swap Counter: 28        Sorted Array: 1 2 3 4 5 6 7 8
Quick Sort      =>      Comparison Counter: 28          Swap Counter: 19        Sorted Array: 1 2 3 4 5 6 7 8
Merge Sort      =>      Comparison Counter: 12          Swap Counter: 0         Sorted Array: 1 2 3 4 5 6 7 8
PS C:\Users\e.kabalci2018\Desktop\src\sorts>
```



```
Initial Array: 4 2 6 5 1 8 7 3

Selection Sort  =>      Comparison Counter: 28          Swap Counter: 7         Sorted Array: 1 2 3 4 5 6 7 8
Bubble Sort     =>      Comparison Counter: 27          Swap Counter: 12        Sorted Array: 1 2 3 4 5 6 7 8
Quick Sort      =>      Comparison Counter: 14          Swap Counter: 10        Sorted Array: 1 2 3 4 5 6 7 8
Merge Sort      =>      Comparison Counter: 15          Swap Counter: 0         Sorted Array: 1 2 3 4 5 6 7 8
PS C:\Users\e.kabalci2018\Desktop\src\sorts>
```



```
Initial Array: 1 2 3 4 5 6 7 8

Selection Sort  =>      Comparison Counter: 28          Swap Counter: 7         Sorted Array: 1 2 3 4 5 6 7 8
Bubble Sort     =>      Comparison Counter: 7           Swap Counter: 0         Sorted Array: 1 2 3 4 5 6 7 8
Quick Sort      =>      Comparison Counter: 28          Swap Counter: 35        Sorted Array: 1 2 3 4 5 6 7 8
Merge Sort      =>      Comparison Counter: 12          Swap Counter: 0         Sorted Array: 1 2 3 4 5 6 7 8
PS C:\Users\e.kabalci2018\Desktop\src\sorts>
```

Bubblesort; In reverse sorted, since the first element is at the end, comparison and swap are made in all for loops from beginning to end, it is in the worst case. This is the best case, as Sorted also checks each element once and knows it is in place. However, as can be seen from the time complexity value, the worst case is in the randomized case. It tries to make a swap on each element.

QuickSort: Sorted and revesely sorted also shuffle the list and reorder it. Divide and conquer also cannot understand that the list is in order at the beginning. Therefore, in these two cases, both the comparison counter and the swap are high. It also includes Randomized. Although it is a slightly better algorithm in terms of complexity values, the swap and comparison counter values in random are not good enough to make a difference.

SelectionSort: In Reversely sorted, although the Comparison value is the same as the others, the swap value is low because it scans the entire list and makes a swap after selecting a result. This requires the for loop to work hard in the background. Since it scans the entire loop in the background every time, the comparison counter and swap values are equal in all cases.

MergeSort: Reversely sorted seems to be the best case as there is no swap and the operation is done by dividing the list. It does not use swap in all cases. The comparison counter is also generally good in terms of value, but while doing so, it reduces space efficiency.