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

ŞEHMUS ACAR

1) Selection Sort

Time Analysis	When I wrote the selection sort algorithm, I used two nested loops. The outer loop runs *n-1* times, where n is the length of the array. The inner loop starts from the current index of the outer loop plus one and goes to the end of the array. As the outer loop progresses, the number of iterations in the inner loop decreases. Regardless of the initial order of the array, each element in the inner loop is compared to find the smallest one, which leads to approximately ***n(n-1) / 2*** comparisons in total. Therefore, the time complexity of the algorithm is considered to be ***O(n²)*** in all cases.
Space Analysis	While writing the algorithm, I performed an in-place sort on the array, using a fixed amount of extra space. The algorithm uses a few auxiliary variables like **n**, **minIndex**, **currentIndex**, but the amount of these variables is independent of the input size. Since the amount of space used for these variables is minimal, the space complexity of the algorithm is expressed as ***O(1).***

2) Bubble Sort

Time Analysis	When I wrote the bubble sort algorithm, I used loops and conditions to swap elements. The outer loop continues as long as there's at least one swap in the pass. Each iteration through the array decreases the array length by one, as the largest element settles at the end and no longer needs to be considered. The inner loop starts from the beginning of the array, swapping elements if the previous one is greater than the current. Each pass reduces the number of comparisons by one. This process results in ***n(n-1) / 2*** comparisons in the worst case, and the time complexity of the algorithm is considered ***O(n²).***
Space Analysis	While writing the algorithm, I performed an in-place sort on the array, using a fixed amount of extra space. I used a few auxiliary variables like **n**, **swap_counter**, **currentIndex**, but the amount of these variables is independent of the input size. Since the amount of space used for these variables is minimal, the space complexity of the algorithm is expressed as ***O(1).***

3) Quick Sort

Time Analysis	When I wrote the quicksort algorithm, I used a recursive approach to partition the array. At each step, a pivot element is chosen and the array is split based on this pivot, with each side sorted separately. In the best and average cases, this approach divides the array into parts with logarithmic depth, and the time complexity of the algorithm is considered to be ***O(n log n).*** However, in the worst case, such as when the array is already sorted, the time complexity can be ***O(n²)*** because the pivot selection can consistently trigger the worst-case scenario
Space Analysis	While writing the quicksort algorithm, I performed an in-place sort on the array without using extra space. The algorithm uses stack space for recursive calls. The depth, and thus the stack space required, varies depending on how the array is partitioned; it is ***O(log n)*** in the average case and ***O(n)*** in the worst case. This determines the space complexity of the algorithm.

4) Merge Sort

Time Analysis	When I wrote the merge sort algorithm, I used a recursive approach to divide the array into two halves, sort each half separately, and then merge them. This process works by dividing the array size in half each time and merging the divided arrays takes ***O(n)*** time. Since this process repeats at a logarithmic number of levels, the time complexity of the algorithm is considered ***O(n log n)*** in both average and best cases. This complexity holds true regardless of the initial order of the array.
Space Analysis	While writing the merge sort algorithm, I created temporary arrays used for sorting. These arrays store the subsets of the original array and new temporary arrays are created with each recursive call. Therefore, the algorithm uses a total of ***O(n)*** additional space. The stack space required for the recursive calls is ***O(log n)*** on average and in the best case, which determines the overall space complexity of the algorithm.

General Comparison of the Algorithms

Selection Sort and **Bubble Sort**: Both algorithms have ***O(n²)*** time complexity, suitable for small data sets but inefficient for large data. They sort in place with ***O(1)*** space complexity.

Quick Sort: Typically operates in ***O(n log n)*** time, ideal for large datasets. However, poor pivot selection can degrade performance to ***O(n²).*** Its space complexity averages ***O(log n)*** depending on the stack.

Merge Sort: Operates consistently in ***O(n log n)*** time and offers stable sorting. Its space complexity is ***O(n)*** due to the use of additional arrays.