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

SEMİH SARIKOCA

1) Selection Sort

Time Analysis	Selection Sort has a time complexity of O(n^2) in all cases. This is because it iterates through the array multiple times, each time finding the minimum (or maximum) element and swapping it with the element in its proper position. Its nested loops make Selection Sort highly inefficient, especially for large datasets, as it needs to compare each element with every other element in the unsorted part of the array in all cases.
Space Analysis	Selection Sort has a space complexity of O(1), meaning it only requires a constant amount of extra space for temporary variables, regardless of the input size. Thin making it memory-efficient.

2) Bubble Sort

Time Analysis	Bubble Sort has a time complexity of O(n^2) in all cases. This is because it iterates through the array multiple times, comparing adjacent elements and swapping them if they are in the wrong order. Its nested loops make Bubble Sort highly inefficient, especially for large datasets, as it needs to compare each element with every other element in the worst-case scenario.
Space Analysis	Bubble Sort has a space complexity of O(1), meaning it doesn't use any additional space proportional to the size of the input array, making it memory-efficient similar to Selection Sort.

3) Quick Sort

Time Analysis	Quick Sort has an average-case time complexity of O(n log n). This is because it partitions the array into smaller sections based on a pivot element and then recursively sorts those sections. In each iteration, it divides the array into two partitions, which halves the size of the problem, leading to a logarithmic behavior. The key to Quick Sort's efficiency is its ability to quickly divide the problem into smaller sub-problems, making it highly efficient on average for large datasets.

Space Analysis	Quick Sort has a average space complexity of O(n) due to its recursive nature. Each recursive call consumes stack space proportional to the logarithm of the input size.

4) Merge Sort

Time Analysis	Merge Sort has a consistent time complexity of O(n log n) in all cases. It achieves this by recursively dividing the array into smaller halves until each sub-array contains only one element, then merging them in sorted order. The key to Merge Sort's efficiency is its ability to divide the problem into smaller sub-problems and then combine them in a sorted manner, which avoids the inefficient comparisons of other algorithms like Bubble Sort and Selection Sort.
Space Analysis	Merge Sort always requires O(n) extra space for merging. This is because it needs additional memory to hold the merged result of sub-arrays during the merging phase.

General Comparison of the Algorithms

Quick Sort and Merge Sort are efficient for large datasets, with Quick Sort being faster on average but Merge Sort being more stable.

Quick Sort typically uses O(log n) extra space for recursion but can degrade to O(n) in the worst case. Merge Sort requires O(n) extra space for merging, making it less memory-efficient compared to Quick Sort in terms of space usage.

Bubble Sort and Selection Sort are simple and easy to understand but highly inefficient for large datasets, making them unsuitable for practical use except for small datasets or educational purposes.

Both Bubble Sort and Selection Sort typically use O(1) extra space, making them memory-efficient but not suitable for large datasets due to their quadratic time complexity.