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

MEHMET MAHIR KAYADELEN

1) Selection Sort

Time Analysis	Best, Average, and Worst Case: O(n^2) This complexity arises because Selection Sort systematically scans through the remaining unsorted part of the array to find the minimum element and swap it into place. Each pass selects the next smallest element, requiring a scan of $n-1$ n−1, $n-2$ n−2, $n-3$ n−3, ..., 1 comparisons, leading to a total of $n(n-1)/2$ comparisons, which simplifies to O(n^2)
Space Analysis	Worst Case: O(1) Selection Sort is an in-place sorting algorithm. It only uses a fixed amount of extra space to store index variables and the temporary variable for swapping. Thus, it has a constant space complexity, making it space-efficient.

2) Bubble Sort

Time Analysis	Best Case: O(n)(when the array is already sorted, and no swaps are needed). Average and Worst Case: O(^2). Each element is compared with its adjacent element, and swaps are made if they are in the wrong order. This process is repeated for each element, leading to nested loops that each run up to n times.
Space Analysis	Worst Case: O(1). Like Selection Sort, Bubble Sort operates in-place. It does not require additional space that grows with the input size, using only extra space for several temporary variables used during swapping and comparison.

3) Quick Sort

Time Analysis	Best and Average Case:O(nlogn). The array is divided into subarrays around a pivot element, where ideally, each pivot divides the list into two equal halves, leading to a logarithmic number of divisions. Worst Case: O(n ^2) (when the smallest or largest element is regularly picked as the pivot). Poor pivot choices can degrade performance, requiring partitioning around each element
Space Analysis	Worst Case: O(n) in the naive version or O(logn) with tail recursion optimization. QuickSort is not a strictly in-place algorithm as it requires stack space to store the bounds of the subarrays in recursive calls. The space needed depends on

	the depth of the recursion tree, which can be O(logn) in the best case or O(n) in the worst case if the pivot splits are unbalanced.

4) Merge Sort

Time Analysis	All Cases (Best, Average, and Worst): O(nlogn). Merge Sort consistently divides the array into two halves and takes linear time to merge two halves. The consistent division leads to a logarithmic number of split levels, and merging each level runs in linear time, hence O(nlogn).
Space Analysis	Worst Case: O(n). Merge Sort requires additional space for the temporary arrays used during the merging process. This additional space is proportional to the size of the array being sorted, hence the O(n) space complexity. Unlike the previously mentioned algorithms, Merge Sort is not in-place as it requires additional memory for merging.

General Comparison of the Algorithms

Selection Sort and **Bubble Sort** are both simple to understand and implement but generally inefficient for large datasets due to their O(n ^2) time complexity in average and worst cases. Bubble Sort has a slight advantage in that it can terminate early if the array becomes sorted before all passes are completed.

QuickSort is significantly faster in practice among comparison-based sorts due to its O(nlogn) average time complexity. However, its performance can degrade to O(n ^2) in the worst case, particularly if poor pivot choices are made. It’s generally preferred for systems where space is less of a concern, given its higher stack space usage in recursion.

Merge Sort offers excellent performance with a guaranteed O(nlogn) time complexity in all cases but at the expense of additional memory usage, as it is not an in-place sorting algorithm. It is very useful in scenarios where data does not fit into memory (like external sorting) or stability is required.