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

MERT EMİR ŞEKER 200104004085

1) Selection Sort

Time Analysis	**Why Slow:** Selection Sort is inherently slow due to its O(n^2) complexity, even though it minimizes the number of swaps. This is because each element is compared to all other unsorted elements to find the minimum. Double for loop structure ensures that every element is considered, leading to high overall comparisons. **Why Fast:** It has no best-case scenario optimizations; the algorithm always performs the full O(n^2) comparisons regardless of the initial order of the array.
Space Analysis	**Space Usage:** Selection Sort is an in-place algorithm with O(1) space complexity beyond the input array. It only requires a minimal amount of additional space for variables such as the index of the minimum element (min_index).

2) Bubble Sort

Time Analysis	**Why Slow:** Bubble Sort is slow in general due to its O(n^2) complexity from nested loops, where each element is compared and potentially swapped. **Why Fast:** It performs better when the array is nearly sorted, as I use a flag to terminate early if no swaps occur in a pass. This optimization reduces complexity to O(n) when minimal rearrangement is needed.
Space Analysis	**Space Usage:** It operates in-place, requiring minimal space beyond the input array, with a constant space complexity of O(1). Only a few auxiliary variables are used, like the flag for tracking swaps. This small footprint makes it space-efficient for memory-sensitive applications.

3) Quick Sort

Time Analysis	**Why Slow:** In the worst case, if the array is sorted or nearly sorted, and the last element is always chosen as the pivot, the complexity degrades to O(n^2). This happens because the partitioning step fails to divide the array efficiently, leading to one very small and one very large partition repeatedly. **Why Fast:** Quick Sort is efficient due to its average and best-case time complexity of O(nlogn), which results from the divide-and-conquer strategy. The array is divided into partitions, and each part is sorted independently. Recursive approach efficiently reduces the problem size with each call, speeding up the sorting process significantly compared to O(n^2) algorithms.
Space Analysis	**Space Usage:** Quick Sort is not a space-efficient algorithm in its recursive implementation due to the space required for the recursion stack. The depth of the recursion tree can go up to O(n) in the worst case, though it's typically O(logn) in the best or average cases. While the algorithm sorts in place, the additional space for the recursion stack depends on the depth of the recursive calls, which varies with the input array's nature and the choice of pivot.

4) Merge Sort

Time Analysis	**Why Fast:** Merge Sort is highly efficient with a guaranteed time complexity of O(nlogn) across all cases—best, average, and worst—due to its divide-and-conquer strategy. This approach involves recursively splitting the array into halves, sorting each half, and merging them back together. This method ensures that each element is part of a sorted subset, which are then merged in a manner that preserves order efficiently. **Consistent Performance:** Unlike some other sorts, the performance of Merge Sort does not degrade based on the initial order of the array because it systematically divides and conquers regardless of the initial data distribution.
Space Analysis	**Space Usage:** Unlike in-place sorting algorithms, Merge Sort requires additional space to hold the temporary array used for merging. This leads to a space complexity of O(n).Here, a temporary array of the same size as the input array is used to help in merging steps. This array is crucial as it temporarily holds the merged results before copying them back to the original array.

General Comparison of the Algorithms

Best Case Performance:

Quick Sort and Merge Sort excel in the best-case scenarios. Quick Sort can approach O(nlogn) efficiency when the pivot elements divide the array into nearly equal parts. Merge Sort always performs at O(nlogn) because it consistently divides the array in half.

Bubble Sort shows significant improvement in best-case scenarios, performing at O(n) when the array is already sorted due to its early termination if no swaps are made on a new pass.

Selection Sort remains O(n^2), even in the best case, because it must still scan through each element to find the minimum.

Average Case Performance:

Merge Sort maintains a steady O(nlogn) performance regardless of the initial order of the elements due to its methodical divide and conquer approach.

Quick Sort typically offers O(nlogn) performance, assuming good pivot selection that reasonably divides the array.

Both Bubble Sort and Selection Sort generally operate at O(n^2). Their simplistic comparison and swap operations don't adapt based on the average distribution of data.

Worst Case Performance:

Merge Sort is the most reliable, continuing to perform at O(nlogn) due to its inherent design, which doesn't depend on data distribution.

Quick Sort can degrade to O(n^2) in the worst-case scenarios, particularly when the smallest or largest element is consistently chosen as the pivot.

Bubble Sort and Selection Sort are inherently O(n^2) due to their need to repeatedly pass through the list to sort or find the minimum element, making them inefficient for large or complex datasets.