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

RECEP FURKAN AKIN

## 1) Selection Sort

| | |
|---|---|
| **Time Analysis** | The outer loop runs `n - 1` times, where `n` is the length of the array. For each iteration of the outer loop, the inner loop runs `n-i` times, where `i` is the current iteration of the outer loop.<br><br>This results in a time complexity of O(n^2) in the worst-case, average-case, and best-case scenarios, as the algorithm always iterates over all pairs of elements, regardless of the initial order of the array. |
| **Space Analysis** | Selection Sort algorithm has a space complexity of O(1).<br><br>The algorithm sorts the array in-place, meaning it doesn't require any additional space that scales with the input size. |

## 2) Bubble Sort

| | |
|---|---|
| **Time Analysis** | - Worst-case scenario (O(n^2)): The worst-case scenario for Bubble Sort is when the input is in reverse order. In this case, Bubble Sort would need to traverse through all the elements for each element in the array, leading to a quadratic time complexity.<br><br>- Average-case scenario (O(n^2)): Even on average, Bubble Sort has a quadratic time complexity. This is because, on average, Bubble Sort still needs to make a significant number of comparisons and swaps.<br><br>- Best-case scenario (O(n)): The best-case scenario for Bubble Sort is when the input is already sorted. In this case, Bubble Sort only needs to traverse the array once without making any swaps, leading to a linear time complexity. |
| **Space Analysis** | In terms of space complexity, space complexity is O(1), meaning it uses a constant amount of space. This is because it only uses a single additional memory space for the 'swapped' variable, regardless of the size of the input array. |

### 3) Quick Sort

| | |
|---|---|
| **Time Analysis** | - Best Case: O(n log n) - This is the case when the pivot element is always the median of the array. This results in a balanced partitioning.<br>- Average Case: O(n log n) - Even in the average case, Quick Sort performs well and has a time complexity of O(n log n).<br>- Worst Case: O(n^2) - This occurs when the pivot element is the smallest or the largest in the array, leading to an unbalanced partitioning. |
| **Space Analysis** | The space complexity of Quick Sort is O(log n). This is because the maximum depth of the recursion tree is log n and for each recursive call, we need O(1) space. Hence, the space complexity is O(log n). |

### 4) Merge Sort

| | |
|---|---|
| **Time Analysis** | - Best Case: O(n log n) – Merge Sort always divides the array into two halves and takes linear time to merge two halves.<br>- Average Case: O(n log n) - Same as best case.<br>- Worst Case: O(n log n) - Same as best case. |
| **Space Analysis** | The space complexity of Merge Sort is O(n). This is because the algorithm requires a temporary array of size n for the merge process. |

## General Comparison of the Algorithms

Bubble Sort is a comparison-based algorithm that swaps adjacent elements if they're in the wrong order. It's efficient for small or mostly sorted lists but not for large ones, with a time complexity of O(n^2).

Selection Sort finds the minimum element in an unsorted array and moves it to the start, repeating this process until sorted. It's efficient for small lists but not large ones, with a time complexity of O(n^2).

Quick Sort is a divide-and-conquer algorithm that selects a pivot and partitions elements into two sub-arrays based on the pivot, then recursively sorts them. It has an average and best-case performance of O(n log n), but a worst-case of O(n^2). It's not a stable sort.

Merge Sort is a divide-and-conquer algorithm that divides an unsorted list into sublists, then merges them into sorted sublists until one remains. It has a performance of O(n log n) and is a stable sort.