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

**Adem Murat ÖZDEMİR**

## 1) Selection Sort

| | |
|---|---|
| **Time Analysis** | This Selection Sort iterates through an array, identifying the smallest element in each pass and placing it at the beginning. This process continues until the array is sorted. Despite its simplicity, it exhibits a time complexity of O($n^2$), as it necessitates nested loops to compare elements and perform swaps. Consequently, its efficiency diminishes quadratically with the number of elements. |
| **Space Analysis** | Selection Sort's space complexity is constant, denoted as O(1), meaning it requires a fixed amount of extra space regardless of the size of the input array. This is because it sorts the array in place, without requiring additional memory proportional to the input size. |

## 2) Bubble Sort

| | |
|---|---|
| **Time Analysis** | Bubble Sort compares adjacent elements and swaps them if they are in the wrong order, repeatedly traversing the array until no more swaps are needed, indicating the array is sorted. In the worst-case scenario, where the array is sorted in reverse order, Bubble Sort requires O($n^2$) comparisons and swaps, as it must traverse the array multiple times. Despite its simplicity, its time complexity makes it inefficient for large datasets. However, for small datasets or nearly sorted arrays, Bubble Sort can perform reasonably well. |
| **Space Analysis** | Bubble Sort's space complexity is constant, denoted as O(1), since it only requires a fixed amount of extra space for temporary variables, regardless of the size of the input array. This makes it memory-efficient and suitable for sorting large datasets with limited memory resources. |

## 3) Quick Sort

| | |
|---|---|
| **Time Analysis** | Quick Sort employs a divide-and-conquer strategy, selecting a pivot element and partitioning the array into two subarrays such that elements less than the pivot are on the left, and elements greater than the pivot are on the right. In my implementation , pivot is last element of array. It then recursively applies this process to the subarrays. In the best and average cases, Quick Sort has a time complexity of O(nlogn), where n is the number of elements in the array. However, in the worst-case scenario, Quick Sort may degrade to O($n^2$) if poorly chosen pivots result in highly unbalanced partitions, although this is rare. |

| | |
|---|---|
| **Space Analysis** | The space complexity of Quick Sort can range from O(logn) to O(n). This is because recursive calls are placed on the stack, which can worst-case require space proportional to the size of the array, O(n). However, Quick Sort typically uses O(logn) extra space on average. |

### 4) Merge Sort

| | |
|---|---|
| **Time Analysis** | Merge Sort operates by dividing the array into smaller subarrays, sorting each subarray individually, and then merging the sorted subarrays. It utilizes a divide-and-conquer approach. The time complexity of Merge Sort is O(nlogn), where n represents the number of elements in the array. This efficiency is achieved because Merge Sort splits the array into halves recursively until each subarray contains only one element, and then merges them in sorted order, resulting in O(nlogn) comparisons and assignments in total. |
| **Space Analysis** | Merge Sort's space complexity is O(n) because it requires additional memory proportional to the size of the input array to store temporary arrays during the merging process. Despite this, Merge Sort is efficient in terms of memory usage, making it suitable for large datasets. |

## General Comparison of the Algorithms

Comparing the sorting algorithms based on the comparison counter, we observe distinct patterns in their performance. Selection Sort consistently demonstrates the highest number of comparisons among the four algorithms. This is expected, as Selection Sort iterates through the entire array in each iteration to find the minimum element, resulting in $O(n^2)$ comparisons in the worst-case scenario. Bubble Sort follows closely behind Selection Sort in terms of comparison count, as it also compares adjacent elements in each pass through the array. Quick Sort exhibits a moderate number of comparisons, comparable to Selection Sort and Bubble Sort. However, Quick Sort's comparisons are more efficient, as it utilizes a divide-and-conquer strategy to quickly sort the elements. On the other hand, Merge Sort consistently requires the fewest comparisons among the four algorithms. Its divide-and-conquer approach divides the array into smaller subarrays, significantly reducing the number of comparisons needed to sort the elements efficiently.

Comparing the sorting algorithms based on the swap counter reveals additional insights into their efficiency. Selection Sort consistently shows a low number of swaps or even none at all in some cases. This is because Selection Sort primarily focuses on finding the minimum element and placing it in the correct position, leading to minimal or no swaps if the array is already partially sorted. Bubble Sort, on the other hand, tends to exhibit a higher number of swaps compared to Selection Sort. Bubble Sort compares adjacent elements and swaps them if they are in the wrong order, potentially leading to more swaps, especially with random or reverse-sorted arrays. Quick Sort demonstrates varying swap counts depending on the choice of pivot and the input array's initial order. While it generally exhibits fewer swaps than Bubble Sort, it may occasionally require a significant number of swaps in the worst-case scenario. Merge Sort consistently shows no swaps due to save elements to the temp arrays which are named “R” and “L”.