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

Muhammet Talha Memişoğlu

1) Selection Sort

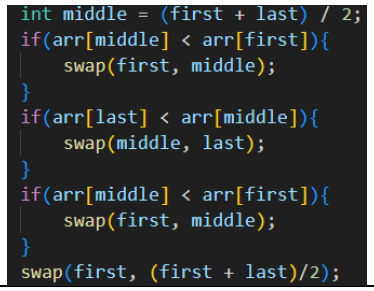
Time Analysis	The overall time complexity of this sorting is O(n^2). Even if the array is nearly sorted, selection sort makes the same number of comparisons as it would for an unsorted array. This is because it always searches for the minimum value in the unsorted portion of the array. Selection sort may make unnecessary swaps. Even if the minimum element is already at its correct position, so I decided to insert condition below to avoid unnecessary swaps. `if(i != minVal)`
Space Analysis	This algorithm has a space complexity of O(1) because it sorts the array in place, i.e., it doesn't use any extra space except for a few variables.

2) Bubble Sort

Time Analysis	• **Best Case:** O(n) - When the array is already sorted. It performs better in sorted array because of this condition : `if(swap_counter == 0)` • **Worst Case:** O(n^2) - When the array is sorted in reverse order. • **Average Case:** O(n^2) - Average time complexity is also O(n^2) Bubble sort performs a high number of comparisons, even when the array is nearly sorted, which is highly inefficient compared to more advanced sorting algorithms like quicksort or mergesort.
Space Analysis	O(1) - The space complexity is constant because the sorting algorithm operates in-place and doesn't require any additional space proportional to the input size.

3) Quick Sort

Time Analysis	• **Best Case:** O(n log n) - Occurs when the partitioning is as balanced as possible. • **Worst Case:** O(n^2) - Occurs when the partitioning is highly unbalanced. • **Average Case:** O(n log n) - On average, quicksort has good performance Quicksort has a worst-case time complexity of O(n^2), which occurs when the partitioning is highly unbalanced. Therefore, I inserted this code for reducing unbalance :

	 So numbers of comparison and swap is decreased in my algorithm.
Space Analysis	The space complexity is O(n) due to recursive calls.

4) Merge Sort

Time Analysis	My algorithm is a typical divide and conquer algorithm with time complexity $O(n\log n)$, where n is the number of elements in the input array. It performs less comparing operations because of dividing. Since the elements are directly copied without the need for swapping, the merge operation avoids the need for additional swapping operations.
Space Analysis	The merge sort is a recursive function, and it consumes additional memory for the recursive calls. The space complexity is $O(n)$. Space complexity can be a disadvantage for very large arrays, as it consumes additional memory for the recursive calls and the merge operation.

General Comparison of the Algorithms

Selection sort and **bubble sort** have $O(n2)$ time complexity, meaning they are highly inefficient for large datasets. **Quick sort** and **merge sort** are significantly more efficient for large datasets, with $O(n\log n)$ time complexity.

- **Sorted Arrays**

  Bubble sort works best on already sorted arrays since it can quickly detect when the array is sorted and exit the loop, making its performance superior in such cases.

- **Random Arrays**

  Quick sort and **merge sort** are more efficient than other sorting algorithms. However, it's worth noting that merge sort doesn't involve swapping, which contributes to its efficiency and should not be overlooked.

- **Reversed Order Arrays**

  Bubble sort performs the worst in terms of the number of comparisons and swap operations. In contrast, **merge sort** performs exceptionally well, especially in such cases