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

MUHAMMET AKKURT

1) Selection Sort

Time Analysis	Selection sort always performs O(n^2) comparisons as it scans the entire unsorted segment of the array to find the minimum or maximum element and then swaps it into place. Two nested loops are used and the comparison is done in the inner loop and the comparison_counter is incremented accordingly. The number of swaps is at most "n-1" in the worst case. In the outer loop the swap is performed and the number of swaps is therefore one less than the number of elements in the array. It is slow for large data as the time complexity is high.
Space Analysis	This algorithm uses a fixed number of local variables and all operations are performed on the given array. For this reason, the space complexity is O(1), meaning that it works without the need for extra space.

2) Bubble Sort

Time Analysis	The best-case for this algorithm is when the sequence is sorted. Since it makes no swaps, it finishes with a comparison depending on the size of the array, so its time complexity is O(n). It repeatedly compares adjacent elements in the array over two nested loops for average and worst cases and swaps them if they are in the wrong order. This is repeated until no replacement is needed. The time complexity is therefore O(n^2). It can be fast for an ordered sequence, but in general it is a slow algorithm due to time complexity.
Space Analysis	This algorithm uses a fixed number of local variables and all operations are performed on the given array. For this reason, the space complexity is O(1), meaning that it works without the need for extra space.

3) Quick Sort

Time Analysis	In the algorithm, the last element of the segment is always chosen as the pivot. For the best and average case, it divides the sequence into two approximately equal parts at each step. This ensures that the depth of the algorithm is "log(n)" and "n" operations are performed at each level, so the total time complexity is O(n logn). For the worst case, the algorithm leaves one side of the array empty while keeping all elements on the other side. This can happen when the array is given sequential or reverse-sequential input. In this case, each partition operation requires n operations and continues through n levels, so the time complexity is O(n^2).

	Choosing a good pivot is fast for large and unordered arrays.
Space Analysis	The variables used in this algorithm are defined locally in each recursive call, so a fixed amount of memory is used. The space complexity of the algorithm depends heavily on the depth of the recursive call stack. Although the algorithm usually runs at O(log n) complexity, it can go as high as O(n) due to poor pivot choices. This indicates that the algorithm may require variable memory usage depending on the array size, its initial ordering and pivot selection strategies.

4) Merge Sort

Time Analysis	This algorithm always works by splitting the array in half, sorting the segments one by one and then merging them. This approach ensures a constant O(n log n) time complexity regardless of the initial state of the array. Two temporary arrays are created and compared and copied sequentially to the main array, so there is no swap operation. The same performance in all cases and lower time complexity than others make it fast and ideal for large arrays.
Space Analysis	In this algorithm, sorting is done recursively. It splits the array in two at each step. Each merge operation uses additional memory equal to the size of the array. However, this memory usage is limited locally in each merge call and these temporary arrays are deleted from memory when the merge is complete. The highest memory usage occurs when temporary arrays of "n" elements are created to cover the entire array. Therefore, the space complexity is O(n).

General Comparison of the Algorithms

Bubble Sort: Works most efficiently on ordered arrays and does not swap. Performance degrades for shuffled and reverse-ordered arrays. These cases also require more comparisons and swaps.

Quick Sort: In the case of mixed sorting, it offered one of the lowest number of comparisons, but the number of swaps remained high. The effect of pivot selection plays a big role in these results. Therefore, it is not ideal in terms of both efficiency and memory usage.

Selection Sort: It does the same number of comparisons and swaps in each case. It can be very inefficient on large arrays because the number of comparisons and swaps is high compared to other algorithms.

Merge Sort: In all three cases, it performed consistently with the least number of comparisons and no swaps. Despite the high memory usage, it offers stability in the number of comparisons and swaps.

In summary, Bubble sort is faster for a sorted array, while Merge sort is ideal for other cases.