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

**Kağan Çakıroğlu**

1. **Selection Sort**

|  |  |
| --- | --- |
| **Time Analysis** | There are two for loops as in nested form. Both loops will run O(n) timesand since swap method’s time complexity is O(1) so time complexity of sorting method will be O(N^2). This scenario is valid for best,worst and average case so  sorted, reversely sorted, randomized array wont effect the speed ofit. As we can see from outputs to for given array number of comparisons are 28 and number of swaps are 7. |
| **Space Analysis** | Regardless of input size selection sort uses initial array for all operations. Swapping elements happens in place without the need of new space. So space complexity is O(1). |

1. **Bubble Sort**

|  |  |
| --- | --- |
| **Time Analysis** | For best case (whichis sorted array) only single traversing of list is enough so its time complexity isO(n). In worst case (which is reversely sorted array)scenario there are two for loops as in nested form just like selection sort. Both loops will run n times so their complexity isO(n) and since swap method is same in all sorting methods overall time complexity is O(n^2).Forgiven array number of comparisons and swaps are 28 for reversely ordered array butfor sorted its 7 and 0 as we stated bubble sort benefical only for best case scenario. |
| **Space Analysis** | Like Selection Sort, Bubble Sort is an in-place sorting algorithm. This means that all the sorting operations are performed within the original array, and no additional array is needed to hold parts of the array during sorting. |

1. **Quick Sort**

|  |  |
| --- | --- |
| **Time Analysis** | Efficiency of QuickSort depends significantly on the choice of the pivot.Recursive calls to sort each segment mean that the depth of the recursion tree approaches log n and at each level entire array isprocessed so number of runs are n. Therefore complexity is N(n\*log n). When pivot is smallest or largest element worst case scenariooccurs then depth of recursion is n so complexity becomes O(n^2). Quick gives best numbers inrandomized and reversely sorted array as we can see from outputs in randomized case number of comparisons are 14 and swap counter is 10. |
| **Space Analysis** | For best case since space is related to recursion stack then space complexity is O(log n). In worst case, depth of recursion becomes linear so complexity becomes O(n). |

1. **Merge Sort**

|  |  |
| --- | --- |
| **Time Analysis** | Since each time array size is halved recursion depth is log n and merging involves comparing elements from both halves. This process is linear relative to the number of elements in the segments being merged. So it will cost O(n) then overall time complexity is O(n log n). For randomized array merge sort gives best result which is 8 with given array. In Sorted and reversely sorted scenario comparison count is increased but still the best among others. |
| **Space Analysis** | Merge method uses two temporary arrays (**leftArr** and **rightArr**) to hold copies of the array segments being merged. The combined size of these temporary arrays at any level of recursion is equal to the size of the original segment being merged, which is n. The recursive calls consume stack space. The depth of the recursive calls in Merge Sort is log n in the best case (perfectly balanced splits). So space complexity is O(n). |

**General Comparison of the Algorithms**

**Time complexity**

**QuickSort** generally has a best and average case time complexity of O(𝑛 log 𝑛), but it can deteriorate to *o*(*n^*2) in the worst case if the pivot selection is poor. **Merge Sort** offers stable time complexity in all cases—best, average, and worst—being O(𝑛 log 𝑛). It's highly efficient for large data sets. **Selection Sort** has a consistent time complexity of *O*(*n*2) in the best, average, and worst cases, making it inefficient for large datasets. **Bubble Sort** also has a time complexity of *O*(*n*2) in average and worst cases. Its best case is *O*(*n*) when the array is already sorted and an optimized version is used that stops if no swaps are made in a new pass.

**Space Complexity**

**Merge Sort** Requires additional space *O*(*n*) for the temporary arrays used during the merge process, making it less space efficient. **QuickSort** Typically uses (log 𝑛) space due to recursive calls; however, this can grow to *O*(*n*) in the worst case due to the recursion stack. **Selection Sort** and **Bubble Sort** both are in-place sorting algorithms with a space complexity of *O*(1), making them more space-efficient than either **Merge** or **QuickSort**.

**Real world Scenarios**

**Merge Sort** highly favored for sorting linked lists and in scenarios where stability is crucial, such as database sorting that must preserve previous orders (like timestamps or entries). **QuickSort** often preferred for arrays because of its cache efficiency and because in-place versions of it use less memory. **Selection Sort** due to its simplicity, it's often taught in introductory computer science courses but rarely used in practice due to its poor performance with large datasets. **Bubble Sort** is impractical for most real-world applications due to its inefficiency. However, it can be useful for small or nearly sorted datasets or where memory space is extremely limited.