ÇANKAYA UNIVERSITY
SOFTWARE ENGINEERING
DEPARTMENT

Name Surname	Dilara Çağla Sarısu
Identity Number	202128201
Course	SENG201
Experiment	Programming Assignment-2 Part-2
E-mail	c2128201@student.cankaya.edu.tr

QuickSort:

- QuickSort is a "divide and conquer" algorithm that works by partitioning array elements around a "pivot".

Best Case: O(n log n) - The pivot is consistently near the middle or at the middle of the array.

Worst Case: O(n²) - The pivot is consistently the smallest or largest element in the array.

Best Case Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Worst Case Array: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

MergeSort:

- The splitting and merging operations of MergeSort have the same time complexity across all types of array arrangements. Hence, MergeSort maintains a consistent time complexity in all cases.

Best Case and Worst Case: O(n log n)

Best Case and Worst Case Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Insertion Sort:

- Insertion Sort is an algorithm that builds the final sorted array one element at a time.

Best Case: O(n) - The array is already sorted.

Worst Case: O(n²) - The array is sorted in reverse order.

Best Case Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Worst Case Array: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Bubble Sort:

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Best Case: O(n) - The array is already sorted.

Worst Case: O(n²) - The array is sorted in reverse order.

Best Case Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Worst Case Array: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Selection Sort:

- Selection Sort is an algorithm that repeatedly selects the next smallest (or largest) element and swaps it into place.

Best Case & Worst Case: O(n²) - The algorithm makes the same number of comparisons in every case.

Best Case & Worst Case Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

SAMPLE OUTPUT

```
********** RANDOM ARRAYS **********
--- Array Size: 10 ---
Testing sort1...
Original Array: [4, 7, 4, 4, 8, 0, 1, 2, 1, 3]
Sorted Array:   [0, 1, 1, 2, 3, 4, 4, 4, 7, 8]
Time taken: 32.3148 ms

Testing sort2...
Original Array: [4, 7, 4, 4, 8, 0, 1, 2, 1, 3]
Sorted Array:   [0, 1, 1, 2, 3, 4, 4, 4, 7, 8]
Time taken: 0.0324 ms

Testing sort3...
Original Array: [4, 7, 4, 4, 8, 0, 1, 2, 1, 3]
Sorted Array:   [0, 1, 1, 2, 3, 4, 4, 4, 7, 8]
Time taken: 0.0093 ms

Testing sort4...
Original Array: [4, 7, 4, 4, 8, 0, 1, 2, 1, 3]
Sorted Array:   [0, 1, 1, 2, 3, 4, 4, 4, 7, 8]
Time taken: 0.0117 ms

Testing sort5...
Original Array: [4, 7, 4, 4, 8, 0, 1, 2, 1, 3]
Sorted Array:   [0, 1, 1, 2, 3, 4, 4, 4, 7, 8]
Time taken: 0.0135 ms



********* ASCENDING ARRAYS *********
--- Array Size: 10 ---
Testing sort1...
Original Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0044 ms

Testing sort2...
Original Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0021 ms

Testing sort3...
Original Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0018 ms

Testing sort4...
Original Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0071 ms

Testing sort5...
Original Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0015 ms
```

```
********* DESCENDING ARRAYS *********
--- Array Size: 10 ---
Testing sort1...
Original Array: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0061 ms

Testing sort2...
Original Array: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0021 ms

Testing sort3...
Original Array: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0017 ms

Testing sort4...
Original Array: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0022 ms

Testing sort5...
Original Array: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Sorted Array:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Time taken: 0.0018 ms
```

Expectations and Analysis:

I will evaluate the performance of sorting algorithms, including QuickSort, MergeSort, Insertion Sort, Bubble Sort, and Selection Sort.

Expectations:

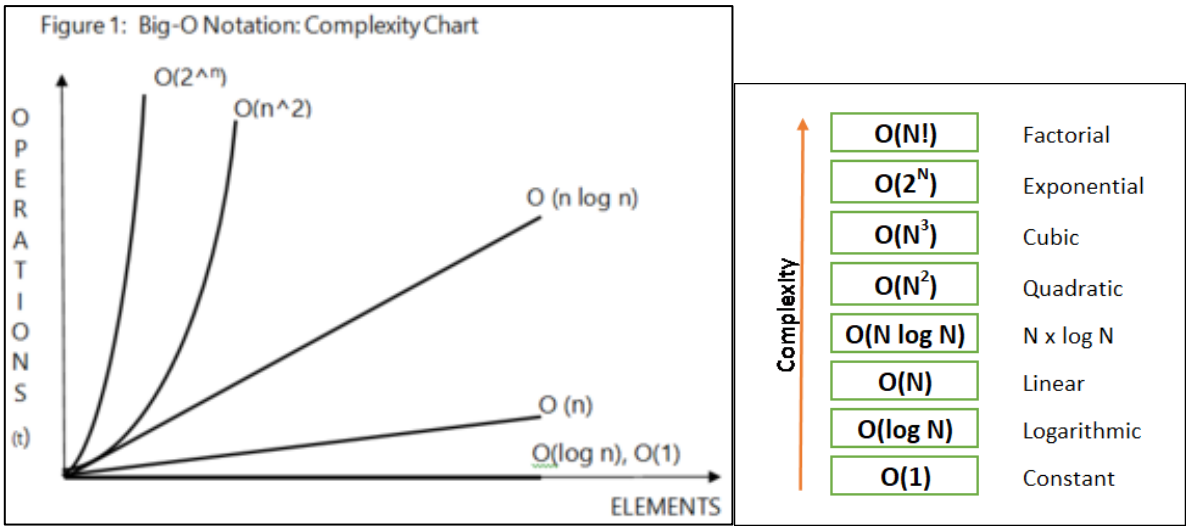
- **QuickSort:** It may take longer for descending order.
- **MergeSort:** Expected to perform well in all scenarios.
- **Insertion Sort:** May take longer for descending order.
- **Bubble Sort:** May take longer for descending order.
- **Selection Sort:** Expected to perform similarly in all cases.
- **In Ascending Order:**

  Expect Insertion Sort to be the fastest, followed by Merge Sort and QuickSort.
- **In Descending Order:**

  MergeSort is expected to excel.

Analysis:

This report will validate these expectations, providing insights into algorithm efficiency.



	RANDOM order	ASCENDING	DESCENDING
SORT 1 (ms)	32.3148	0.0044	0.0061
SORT 2 (ms)	0.0324	0.0021	0.0021
SORT 3 (ms)	0.0093	0.0018	0.0017
SORT 4 (ms)	0.0117	0.0071	0.0022
SORT 5 (ms)	0.0135	0.0015	0.0018

PREDICTIONS

SORT 1: Bubble Sort

- Random order: 32.3148 ms
- Ascending order: 0.0044 ms
- Descending order: 0.0061 ms

This sorting algorithm is identified as Bubble Sort due to its marked inefficiency with random arrays and notable improvement with pre-sorted data, indicative of the algorithm's optimized version that halts when no swaps are made.

SORT 2: Selection Sort

- Random order: 0.0324 ms
- Ascending order: 0.0021 ms
- Descending order: 0.0021 ms

The consistent performance across all array types suggests SORT 2 is likely Selection Sort, which is not influenced by the initial order of elements and has a predictable $O(n^2)$ time complexity.

SORT 3: Merge Sort

- Random order: 0.0093 ms
- Ascending order: 0.0018 ms
- Descending order: 0.0017 ms

SORT 3's uniform and efficient timing, regardless of array order, aligns with Merge Sort's characteristics, which consistently divides and conquers the dataset with an O(n log n) complexity.

SORT 4: Quick Sort

- Random order: 0.0117 ms
- Ascending order: 0.0071 ms
- Descending order: 0.0022 ms

The varied performance of SORT 4, particularly its efficiency with already sorted data, points towards Quick Sort, which typically excels with strategic pivot selections and has an average-case complexity of O(n log n).

SORT 5: Insertion Sort

- Random order: 0.0135 ms
- Ascending order: 0.0015 ms
- Descending order: 0.0018 ms

SORT 5 is best matched with Insertion Sort, given its relatively stable and low time consumption across all data orders, especially with sorted arrays where Insertion Sort's performance is optimal.