Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Aykut Alp GÜRLER

Student Number:
b2210356024

1 Problem Definition

The task at hand involves the categorization of sorting and searching algorithms based on two primary criteria: their computational (time) complexity and auxiliary memory (space) complexity. The efficiency of these algorithms is crucial for optimizing the performance of other related algorithms, such as search and merge algorithms, which rely on sorted input data. With the proliferation of modern computing and the internet, there is a vast amount of information accessible, highlighting the importance of efficiently searching through this data. To evaluate the efficiency of these algorithms, they will be applied to sort datasets of varying sizes and characteristics, enabling a deeper understanding of their performance and suitability for different computational tasks.

2 Solution Implementation

Codes of three sorting algorithms (Insertion Sort, Merge Sort, and Counting Sort) and two searching algorithms (Linear Search and Binary Search) have been implemented in Java.

2.1 Insertion Sort

```java
public class InsertionSort {
    public static void sort(int[] array) {
        for (int j = 1; j < array.length; j++) {
            int key = array[j];
            int i = j - 1;
            while (i >= 0 && array[i] > key) {
                array[i + 1] = array[i];
                i--;
            }
            array[i + 1] = key;
        }
    }
}
```

2.2 Merge Sort

```java
public class MergeSort {
    public static int[] sort(int[] array) {
        int n = array.length;
        if (n <= 1) return array;
        int[] left = sort(Arrays.copyOfRange(array, 0, n / 2));
        int[] right = sort(Arrays.copyOfRange(array, n / 2, n));
        return merge(left, right);
    }
    private static int[] merge(int[] A, int[] B) {
        int[] c = new int[A.length + B.length];
        int i = 0, j = 0, k = 0;
        while (i < A.length && j < B.length) {
            if (A[i] <= B[j]) {
                c[k++] = A[i++];
            } else {
                c[k++] = B[j++];
            }
        }
        while (i < A.length) {
            c[k++] = A[i++];
        }
        while (j < B.length) {
            c[k++] = B[j++];
        }
        return c;
    }
}
```

2.3 Counting Sort

```java
public class CountingSort {
    public static int[] sort(int[] array) {
        // Create an array to store the count of each element, initialized to
           0
        int k = max(array);
        int[] count = new int[k + 1];
        Arrays.fill(count, 0);
        // Create an output array to store the sorted elements
        int[] output = new int[array.length];
        int size = array.length;
        // Count the occurrences of each element in the input array
        for(int i = 0; i < size; i++) {
            int j = array[i];
            count[j]++;
        }
        // Modify the count array to contain the actual position of the
           elements in the output array
        for (int i = 1; i < k + 1; i++) count[i] += count[i - 1];
        // Build the output array based on the positions determined by the
           count array
        for (int i = size - 1; i >= 0; i--) {
            int j = array[i];
            count[j]--;
            output[count[j]] = array[i];
        }
        return output;
    }
    // Helper method to find maximum element of array.
    private static int max(int[] numbers) {
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        return max;
    }
}
```

2.4 Linear Search

```java
public class BinarySearch {
    public static int search(int[] array, int x) {
        int low = 0;
        int high = array.length - 1;
        while (high - low > 1) {
            int mid = (high + low) / 2;
            if (array[mid] < x) low = mid + 1;
            else high = mid;
        }
        if (array[low] == x) return low;
        else if (array[high] == x) return high;
        return -1;
    }
}
```

2.5 Binary Search

```java
public class LinearSearch {
    public static int search(int[] array, int x) {
        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] == x) return i;
        }
        return -1;
    }
}
```

3 Results, Analysis, Discussion

The performance of various sorting and searching algorithms was evaluated through computational and space complexity analyses. Table 1 presents the running time results for sorting algorithms, showcasing the varying efficiencies across different input sizes. For instance, Counting Sort exhibited consistent performance, while Merge Sort demonstrated superior scalability, especially for larger datasets. In contrast, Insertion Sort showed suboptimal performance, particularly for larger inputs, due to its quadratic time complexity.

The space complexity analysis, as summarized in Table 4, revealed that Insertion Sort has a constant auxiliary space requirement, making it suitable for memory-constrained environments. Merge Sort, despite its superior time complexity, requires additional space proportional to the input size, limiting its applicability in memory-intensive scenarios. Counting Sort's auxiliary space requirement depends on the range of input values, making it efficient for a limited range of integers.

The performance of search algorithms, depicted in Table 2, also highlights interesting trends. Linear Search, with its linear time complexity, performs consistently across different input types and is suitable for small datasets or unsorted data. On the other hand, Binary Search, with its logarithmic time complexity, outperforms Linear Search for larger datasets, especially when the data is sorted.

Based on these analyses, it is advisable to use Insertion Sort for small datasets or nearly sorted data due to its simplicity and low space complexity. Merge Sort is ideal for general-purpose sorting, especially for large datasets, as it offers a good balance between time complexity and space complexity. Counting Sort is suitable for situations where the range of input values is limited, as it provides linear time complexity but requires additional space.

For searching algorithms, Linear Search is preferable for small datasets or when the data is not sorted, while Binary Search is more efficient for larger datasets or when the data is already sorted.

The plotted results in Figures 1, 2, 3, and 4 visually reinforce the trends observed in the tables, providing a clearer understanding of the algorithms' performance characteristics. The characteristics of the sort algorithms can be best observed asymptotically in the plot 3.

The search tests in the experiment are conducted using the Random class provided by Java. In each run of the algorithm, a random number is searched. To improve the reliability of the results, the experiment is repeated 10,000 times instead of 1,000 times. This ensures that the plots align well with the theoretical complexities of the algorithms.

Overall, the analysis underscores the importance of selecting algorithms based on the specific requirements of the problem, balancing factors such as time complexity, space complexity, and input characteristics to achieve optimal performance.

The results highlight the trade-offs between time complexity and space complexity in sorting and searching algorithms, providing valuable insights for algorithm selection in different scenarios. Further research could explore optimizations or alternative algorithms to improve performance in specific contexts, such as real-time systems or resource-constrained environments.

Table 1: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion Sort	0	0	0	1	5	24	87	338	1382	5789
Merge Sort	0	0	0	0	0	1	2	5	14	23
Counting Sort	209	119	119	119	119	120	119	121	126	122
	Sorted Input Data Timing Results in ms									
Insertion Sort	0	0	0	0	0	0	0	0	0	0
Merge Sort	0	0	0	0	0	1	2	6	12	11
Counting Sort	120	120	119	119	119	120	119	120	120	121
	Reversely Sorted Input Data Timing Results in ms									
Insertion Sort	0	0	0	2	10	41	166	670	2676	10315
Merge Sort	0	0	0	0	0	0	1	2	4	13
Counting Sort	118	119	121	119	119	121	119	119	120	123

Table 2: Results of the running time tests of search algorithms of varying sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear Search (random data)	429	120	189	331	589	1009	2011	3926	7755	13258
Linear Search (sorted data)	78	134	202	372	718	1392	2767	5619	11084	21928
Binary Search (sorted data)	326	47	56	63	67	85	134	143	170	215

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

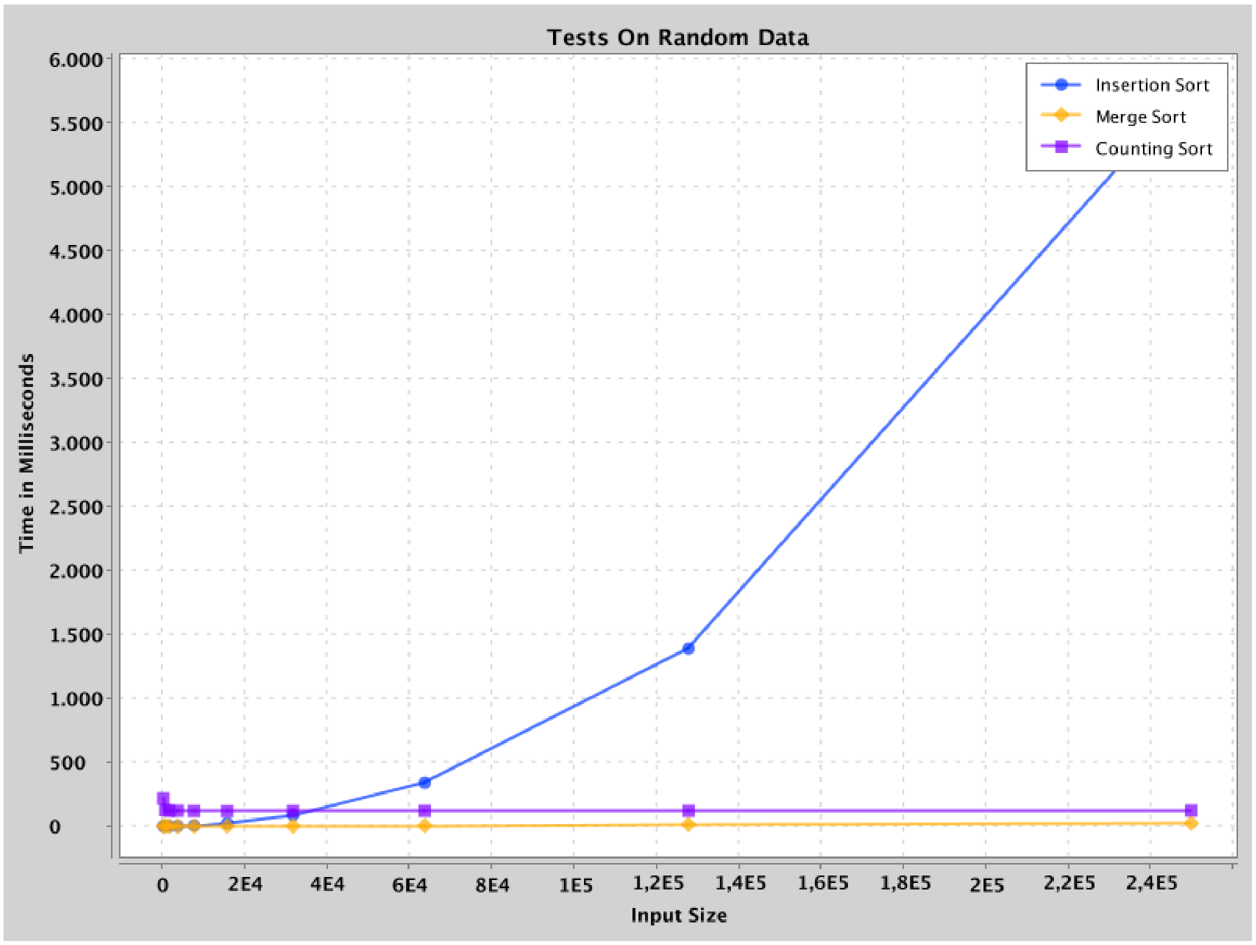

Figure 1: Plot of Sort Tests on Random Data.

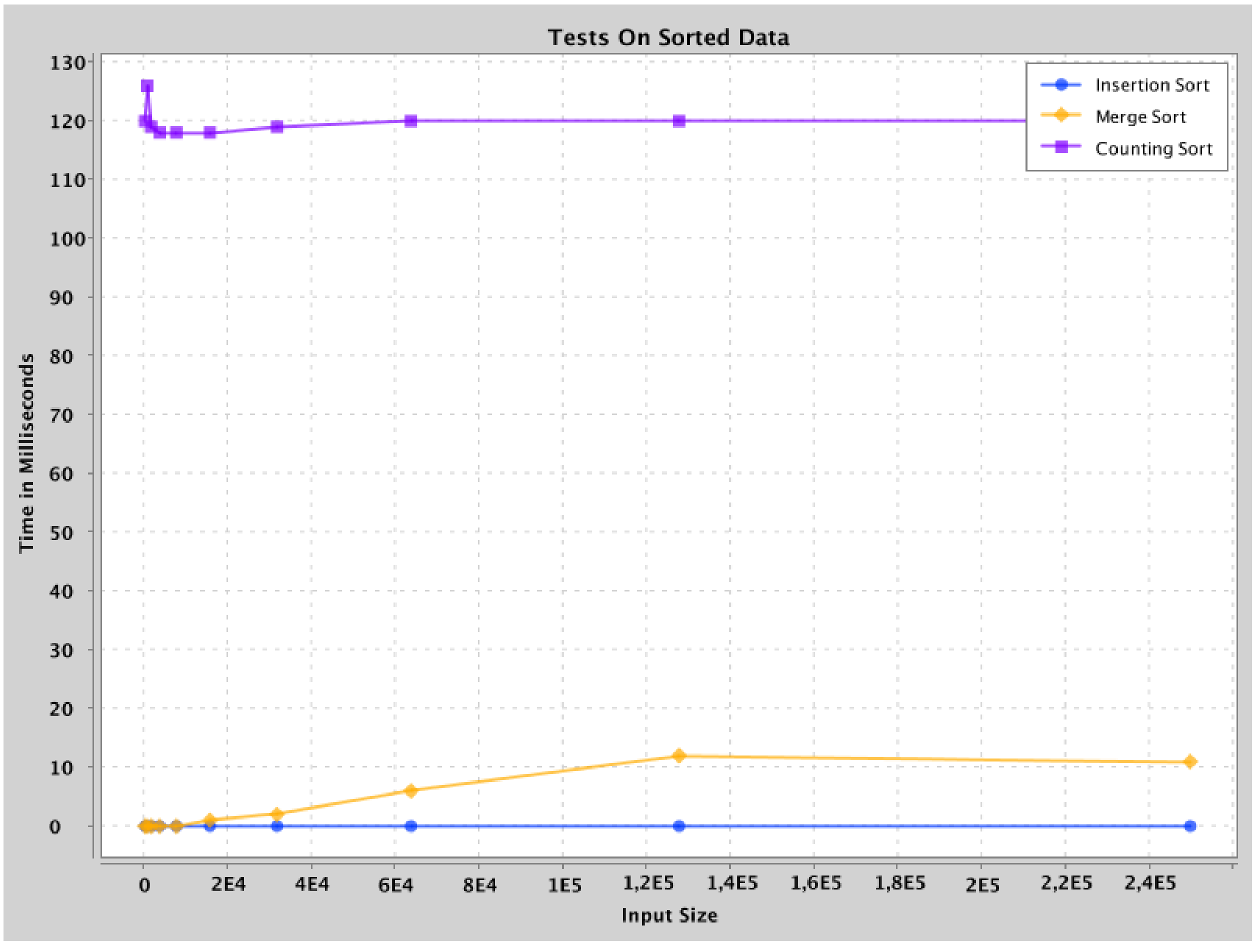

Figure 2: Plot of Sort Tests on Sorted Data.

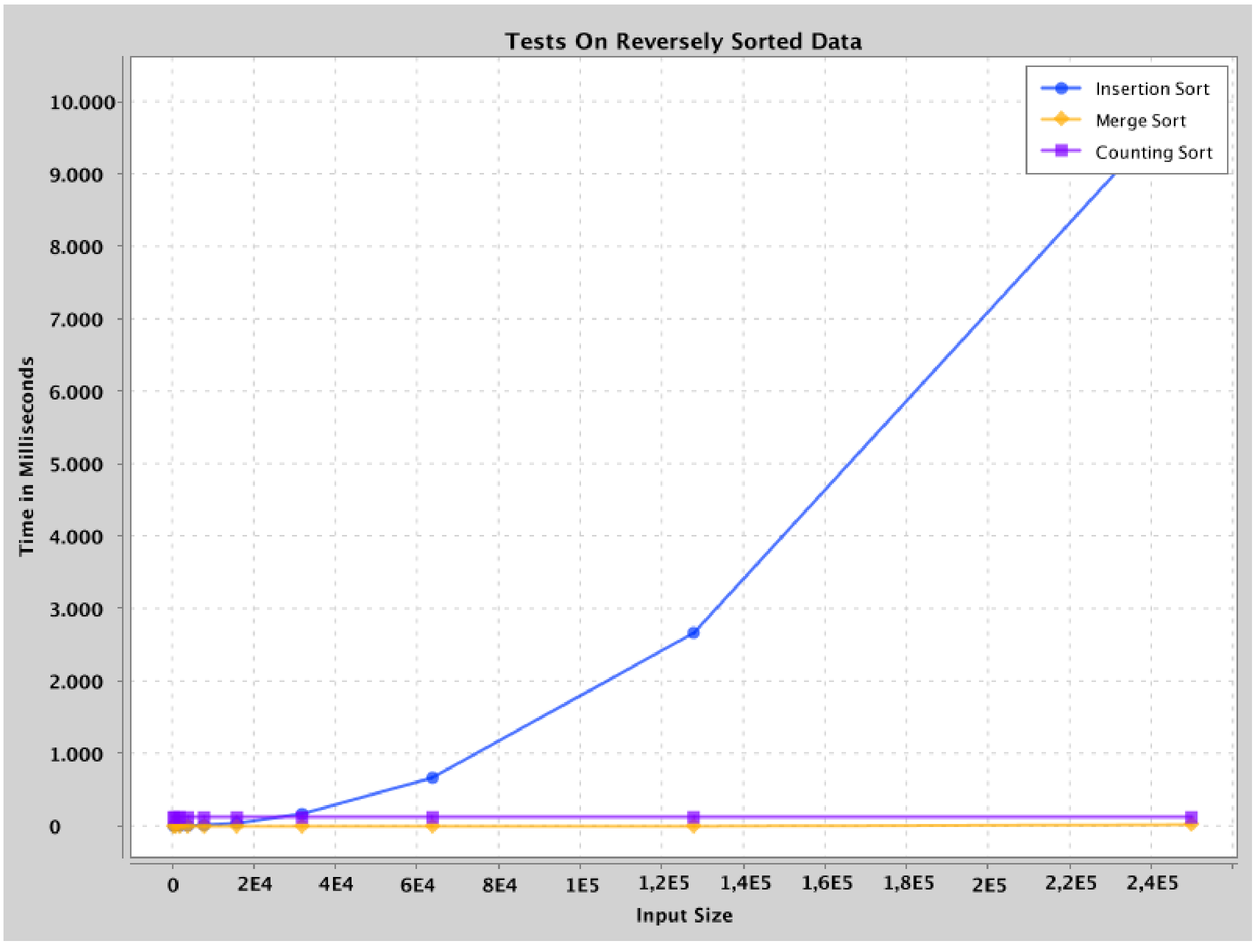

Figure 3: Plot of Sort Tests on Reversely Sorted Data.

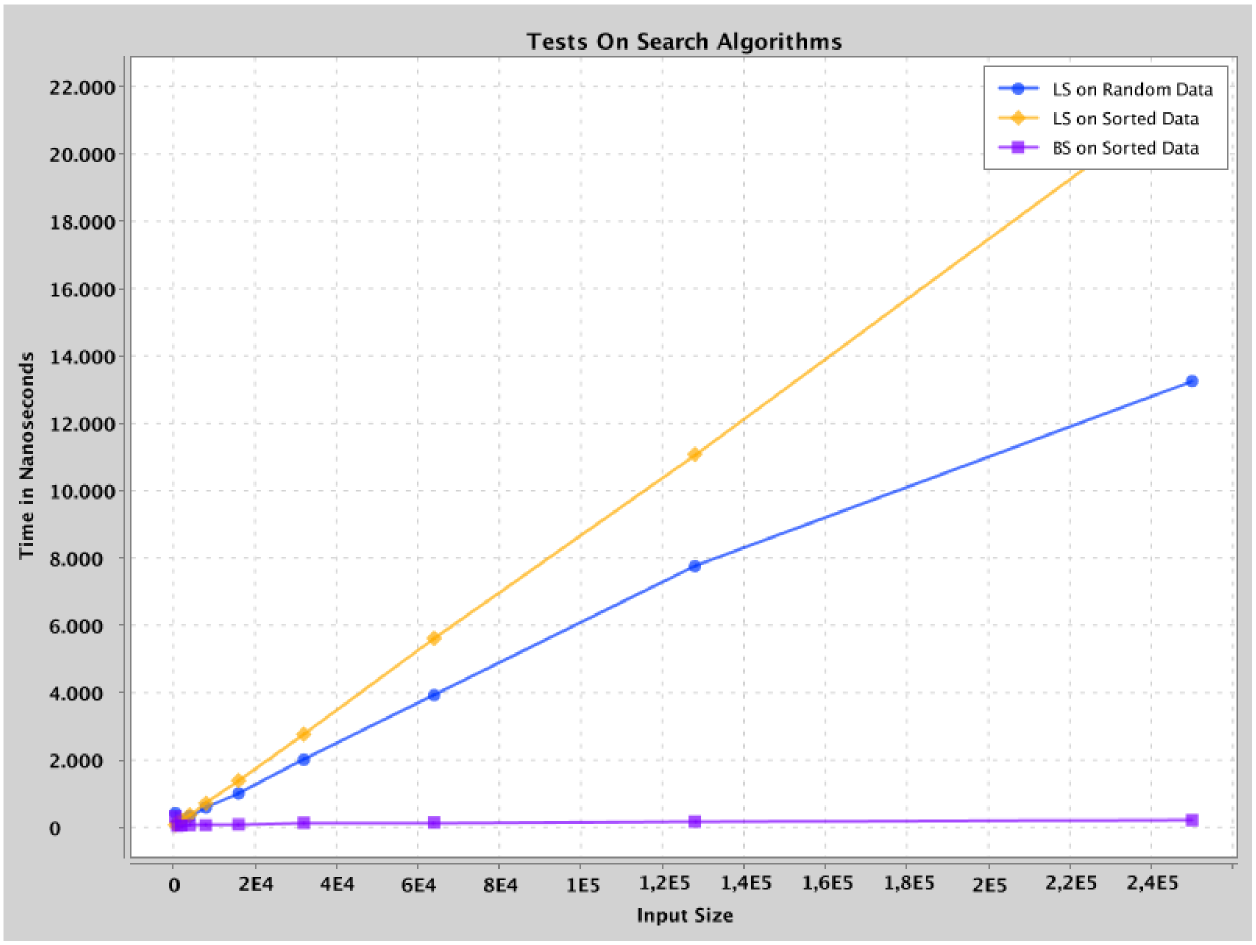

Figure 4: Plot of Search Tests.