Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Tuna Özcan

Student Number:
21987058

1 Problem Definition

The problem at this programming assignment is focused on algorithm complexity analysis for several classic algorithms. In this assignment, we are asked to implement and analyze various sorting and searching algorithms in Java 11. The main goal is to demonstrate the relationship between the running time of the algorithms and their theoretical complexities.

The assignment involves implementing sorting algorithms such as "Insertion Sort", "Merge Sort" and "Counting Sort", as well as search algorithms like "Linear Search" and "Binary Search". The analysis includes experiments on different input types, such as random data, sorted data and randomly sorted data, and sizes, such as 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 250000 running times, and plotting the results to show the performance of each algorithm.

Additionally, we are required to discuss our findings and provide an analysis of the algorithms' computational and auxiliary space complexities.

2 Solution Implementation

This section includes all of the implementations of each algorithm that has ben tested during this experiment.

2.1 Insertion Sort

Insertion Sort is a simple sorting algorithm that iterates through the input array, moving elements into their correct positions by comparing them with adjacent elements. It repeatedly selects an element and inserts it into its sorted position among the previously sorted elements as shown in the below implementation. Insertion Sort has a time complexity of $O(n^2)$.

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

It's complexity makes it inefficient for large datasets, but it is efficient for small or nearly sorted arrays.

2.2 Merge Sort

The Merge Sort algorithm is a recursive sorting algorithm that follows the "divide-and-conquer strategy". It divides the input array into two halves, sorts each half recursively, and then merges the sorted halves to get a single sorted array. The key operation is the merge step, where two sorted subarrays are combined into a single sorted array as shown below in the implementation.

```
public class MergeSort {
    public static void mergeSort(int[] arr) {
        if (arr.length <= 1)
            return;

        int mid = arr.length / 2;
        int[] left = new int[mid];
        int[] right = new int[arr.length - mid];

        System.arraycopy(arr, 0, left, 0, left.length);
        System.arraycopy(arr, mid, right, 0, right.length);

        mergeSort(left);
        mergeSort(right);
        merge(arr, left, right);
    }

    private static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j])
                arr[k++] = left[i++];
            else
                arr[k++] = right[j++];
        }
        while (i < left.length)
            arr[k++] = left[i++];
        while (j < right.length)
            arr[k++] = right[j++];
    }
}
```

Merge Sort has a time complexity of O(n log n) in all cases, making it efficient for large datasets. It is stable and works well for sorting linked lists as well as arrays.

2.3 Counting Sort

Counting Sort is a non-comparison-based sorting algorithm suitable for sorting integers with a limited range. It works by counting the frequency of each distinct element in the input array and then placing each element at it's correct position in the output array based on its count. Counting Sort first determines the maximum element in the input array to establish the range of values, then

constructs a counting array to store the frequency of each element. After computing the cumulative frequency, it uses this information to place each element in its sorted position in the output array.

```java
public class CountingSort {
    public static int[] countingSort(int[] input) {
        int k = max(input);
        int[] count = new int[k + 1];
        int[] output = new int[input.length];
        for (int i = 0; i < input.length; i++) {
            count[input[i]]++;
        }
        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }
        for (int i = input.length - 1; i >= 0; i--) {
            output[count[input[i]] - 1] = input[i];
            count[input[i]]--;
        }
        return output;
    }
    private static int max(int[] input){
        int max = input[0];
        for (int i = 1; i < input.length; i++) {
            if (input[i] > max) {
                max = input[i];
            }
        }
        return max;
    }
}
```

Counting Sort has a time complexity of O(n + k), where n is the number of elements in the input array and k is the range of input values. It is stable and efficient for sorting integer arrays with a relatively small range of values.

2.4 Linear Search

Linear Search is a simple searching algorithm that sequentially checks each element of the array until the target element is found or the end of the array is reached. It starts at the beginning of the array and iterates through each element, comparing it with the target element. If the target element is found, the index of that element is returned; otherwise, -1 is returned to indicate that the target element is not present in the array.

```java
public class LinearSearch {
    public static int linearSearch(int[] arr, int x) {
        int size = arr.length;
        for (int i = 0; i < size; i++) {
            if (arr[i] == x) {
```

```
                    return i;
                }
            }
            return -1;
        }
}
```

Linear Search has a time complexity of O(n), where n is the number of elements in the array. It is suitable for small arrays or unordered lists but becomes inefficient for large datasets compared to like Binary Search. (the next and the last algorithm of this assignment)

2.5 Binary Search

Binary Search efficiently locates the position of a target element within a sorted array by repeatedly dividing the search interval in half. It compares the target element with the middle element of the array, then cuts down the search to the necessty half. This process continues until the target element is found or the search interval is empty.

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

Binary Search has a time complexity of O(log n), making it highly efficient for large datasets. However, it requires the array to be sorted before.

3 Results, Analysis, Discussion

In this section of this report, I will explain and show the results of the tests that has been done with different types of inputs and sizes in the following tables and charts. Running time test results for sorting algorithms are given in Table 1.

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	4	19	75	265	1144	5061
Merge sort	0	0	0	0	0	1	2	5	12	25
Counting sort	115	84	83	83	84	83	84	87	88	98
	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	1	4	8	10
Counting sort	83	83	84	84	83	84	84	87	87	90
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	7	32	134	539	2304	9181
Merge sort	0	0	0	0	0	0	1	3	4	10
Counting sort	85	84	83	83	84	84	85	87	87	89

Running time test results for search algorithms are given in Table 2.

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)	2673	1077	2841	449	782	1010	2010	3920	9225	25156
Linear search (sorted data)	182	227	317	509	899	1530	2848	5438	11881	22266
Binary search (sorted data)	186	253	289	444	863	1675	2909	5386	12342	22469

Complexity analysis tables of the algorithms (Table 3 and Table 4):

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	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

- k means number of unique elements in the input dataset

Table 4: Auxiliary space complexity of the given algorithms.

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

From now on, the following plots will include the visualization of the tests on "random data, sorted data and reversely sorted data" and the algorithms' performance measured in Milliseconds for sorting algorithms and in Nanoseconds in searching algorithms.

In Figure 1, we present the results of the random data tests for the Insertion Sort, Merge Sort, and Counting Sort algorithms.

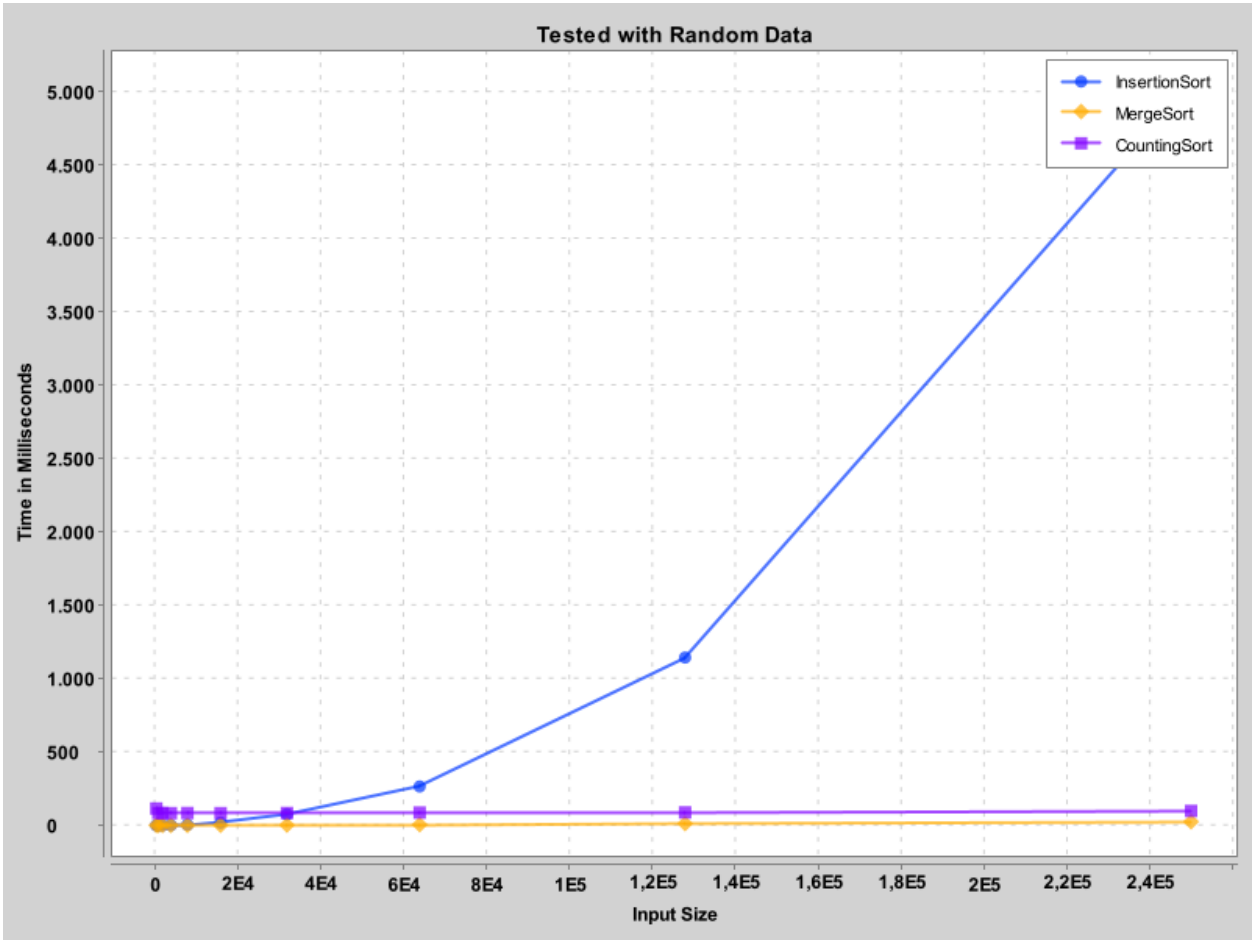

Figure 1: Plot of the functions: InsertionSort, MergeSort, and CountingSort on Random Data tests

By looking at the graph we have, we see that the sorting algorithms work in accordance with the time complexity. The test results plot the varying performance of three sorting algorithms on random data: Insertion Sort, Merge Sort, and Counting Sort. Insertion Sort shows poor scalability, (with its time spent increasing quadratically) as the input size grows resulting in a sharp increase in time spent as the input size grows. This inefficiency is critical for larger input sizes, where the time spent becomes notably high. Conversely, Merge Sort shows a significantly better performance, with its time complexity of O(n log n) ensuring a more stable increase in time spent as input size increases. Even for increasing input sizes, Merge Sort maintains reasonable time consumption, proving its scalability and efficiency in handling large datasets. On the other hand, Counting Sort presents stable performance across different input sizes, with a consistent time spent. Overall, Merge Sort emerges as the most efficient choice for large datasets, while Counting Sort offers stability and efficiency for smaller datasets with manageable input element ranges. In contrast, Insertion Sort's usability is limited to small or nearly sorted datasets due to it's poor scalability.

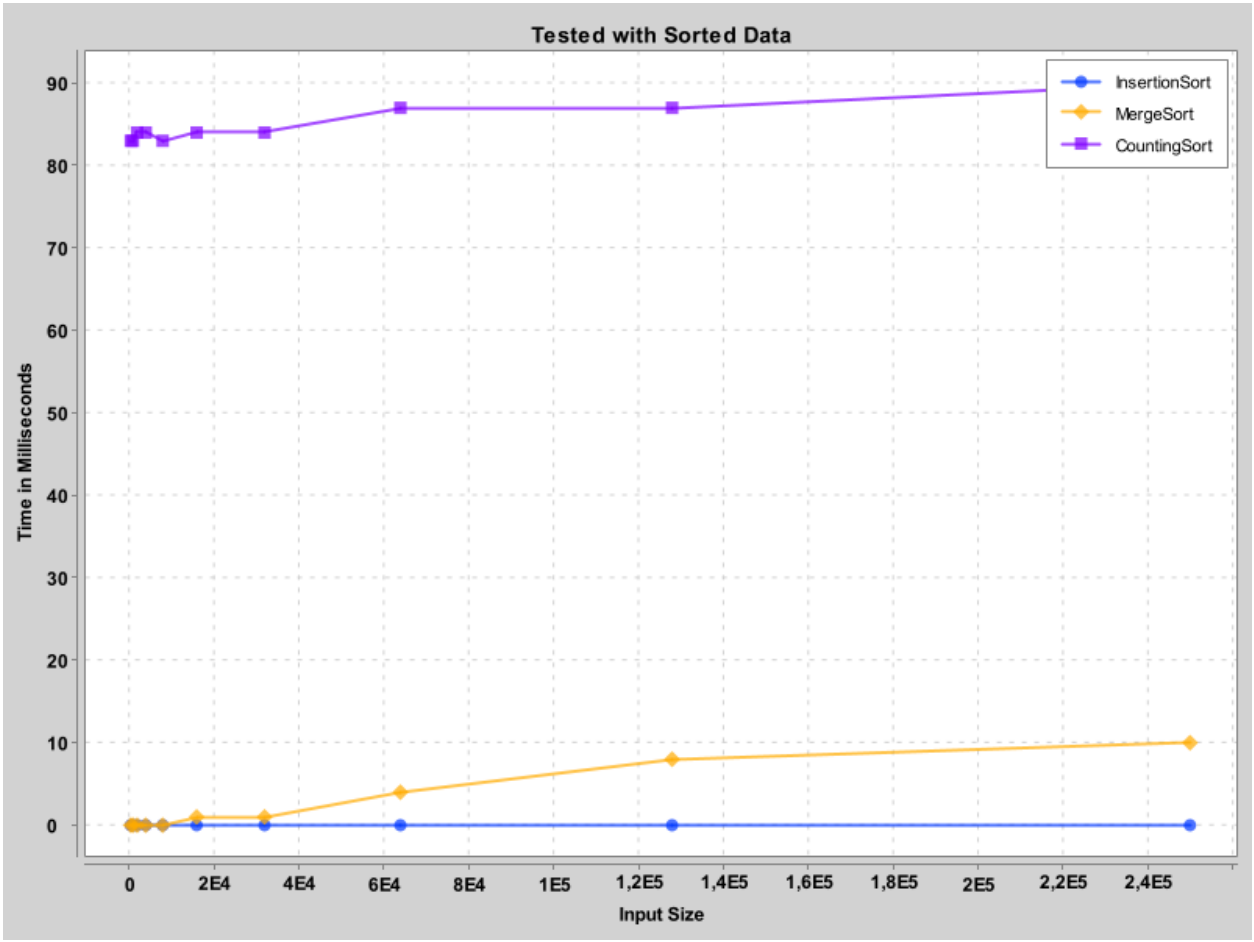

Figure 2: Plot of the functions: InsertionSort, MergeSort, and CountingSort on Sorted Data tests

The provided results shows the performance of three sorting algorithms on to sorted data sets of increasing sizes. Beginning with Sorted Insertion Sort, the time spent remains consistently at zero across all tested input sizes. This outcome aligns with expectations, as Insertion Sort exhibits optimal performance for already sorted data, with a time complexity of O(n) in such cases. Similarly, Sorted Merge Sort shows efficient performance, with negligible(remissable) time spent for smaller input sizes and a slight increase as the input size grows. Merge Sort's time complexity of O(n log n) ensures its effectiveness, even with sorted data. Meanwhile, Sorted Counting Sort exhibits stable performance across different input sizes, BUT with much higher time spent compared to the other two algorithms. When compared with the results from random data analysis, these sorted data tests reconfirm the efficiency and stability of Sorted Insertion Sort and Sorted Merge Sort. However, Sorted Counting Sort's performance remains consistent across both sorted and random data sets, indicating its reliability but also highlighting its much more slower processing speed compared to the other two algorithms for sorted data. Overall, for already sorted data, Sorted

Insertion Sort and Sorted Merge Sort emerge as highly efficient options, with Sorted Counting Sort offering stable but highly slower performance.

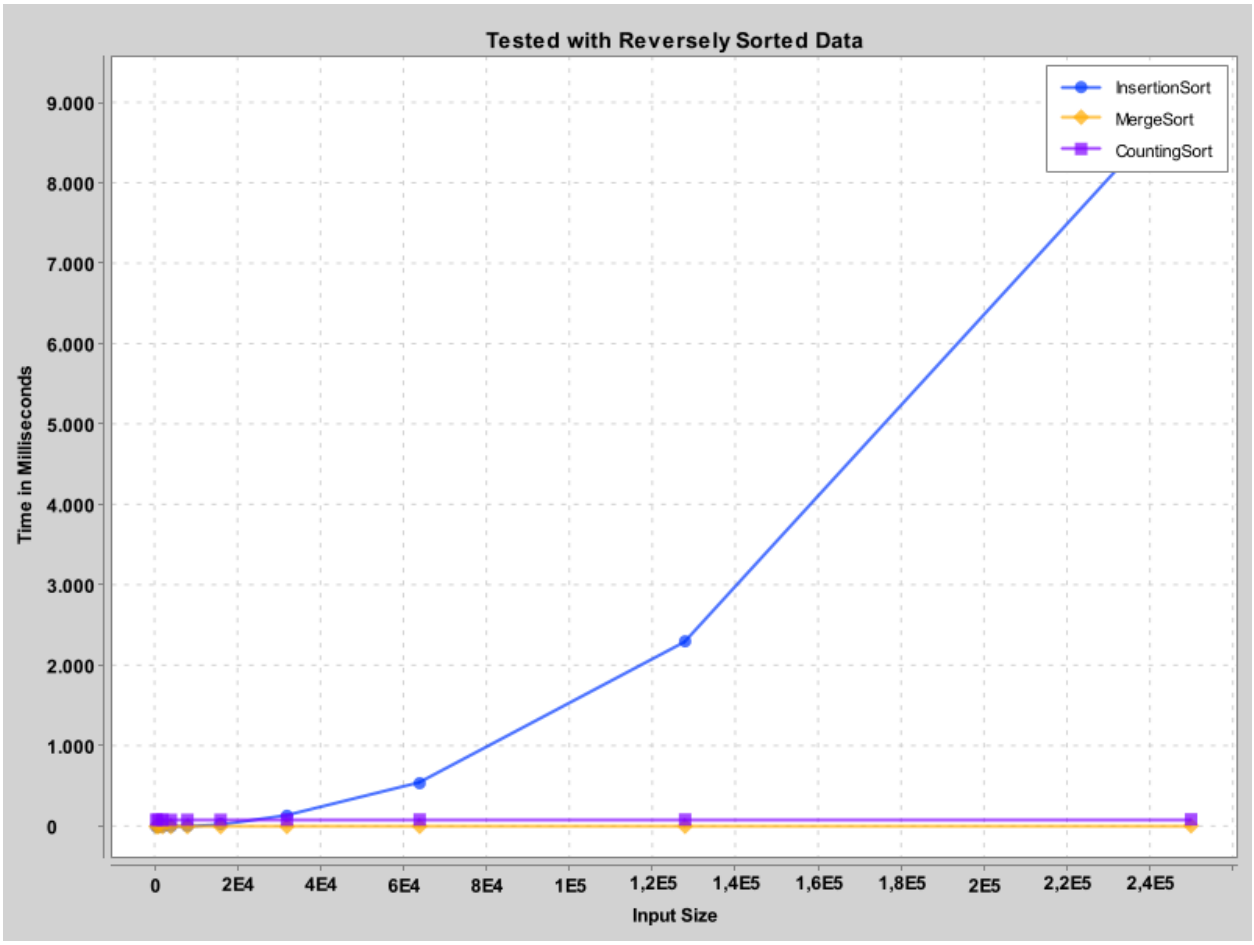

Figure 3: Plot of the functions: InsertionSort, MergeSort, and CountingSort on ReverselySorted Data tests

The provided results shows the performance of three sorting algorithms on reversely sorted datasets of increasing sizes. Reversely Sorted Insertion Sort is clearly shows a notable increase in time spent with larger input sizes due to its quadratic time complexity, whereas Reversely Sorted Merge Sort shows efficient performance, maintaining negligible time spent for smaller inputs and a gradual increase for larger ones, in line with its O(n log n) time complexity. Reversely Sorted Counting Sort performs stable across different input sizes, although with slightly higher times compared to Merge Sort. Comparing these results with previous analyses, Merge Sort consistently shows efficiency across various data distributions, while Counting Sort remains stable but slightly slower, and Insertion Sort proves the least efficient, with reversely sorted data.

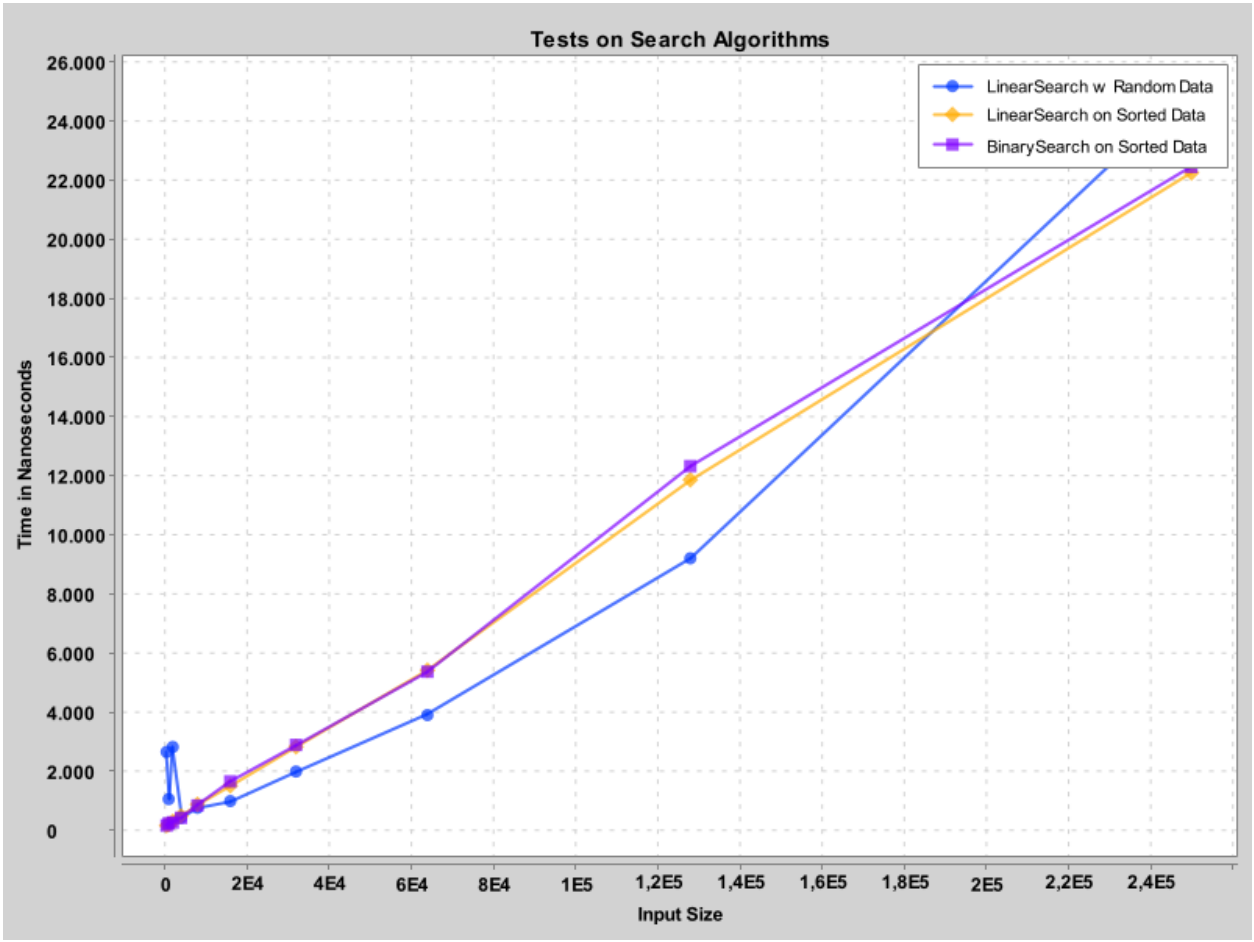

Figure 4: Plot of the functions: Linear Search on Random Data, Linear Search on Sorted Data and BinarySearch on Sorted Data

The provided test results compare the average time spent for different search algorithms: Random Linear Search, Sorted Linear Search, and Sorted Binary Search, across various input sizes. Notably, the Random Linear Search exhibits a significantly higher average time spent compared to the other two algorithms across higher input sizes. This is expected since Random Linear Search does not utilize any specific order or optimization, resulting in a linear time complexity, making it less efficient, especially for larger datasets. On the other hand, both Sorted Linear Search and Sorted Binary Search show similar trends, with Sorted Linear Search outperforming Sorted Binary Search. This is because Sorted Binary Search operates on a sorted dataset, allowing for a more efficient search process by halving the search space with each iteration, resulting in a logarithmic time complexity.In conclusion, Sorted Linear Search demonstrates slightly better performance compared to Sorted Linear Search and Random Linear Search,

References

- https://en.wikipedia.org/wiki/
- https://www.geeksforgeeks.org/
- https://stackoverflow.com/
- https://www.javatpoint.com/