1 Problem Definition

In this study, we explore the performance of three distinct sorting algorithms (Insertion Sort, Merge Sort, and Counting Sort) across different data set conditions to understand their complexity and efficiency. The algorithms are tested against data sets that are in random order, sorted order, and reverse sorted order. Our objective is to analyze how each algorithm behaves under these varying conditions and to depict their performances through graphical representations.
Insertion Sort, known for its simplicity, works well with small data sets or nearly sorted arrays but may struggle with larger, randomly ordered data sets due to its quadratic time complexity. Merge Sort, on the other hand, is a divide-and-conquer algorithm that performs consistently across different data sets, attributed to its logarithmic time complexity. Counting Sort, a non-comparison based algorithm, excels with integers within a specific range, offering linear time complexity, yet its efficiency is limited by the range of the numeric values in the data set.

2 Solution Implementation

2.1 Sorting Algorithm 1 : Insertion Sort

```
public static void insertionSort(int[] A)
    {
        for (int j = 1; j < A.length; j++)
        {
            int key = A[j];
            int i = j - 1;

            while (i >= 0 && A[i] > key)
            {
                A[i + 1] = A[i];
                i = i - 1;
            }
            A[i + 1] = key;
        }
    }
```

2.2 Sorting Algorithm 2 : Merge Sort

```
public static int[] mergeSort(int[] array)
    {
        if (array == null || array.length <= 1) {
            return array;
        }
        int mid = array.length / 2;
        int[] leftArray = new int[mid];
        int[] rightArray = new int[array.length - mid];
```

```java

        System.arraycopy(array, 0, leftArray, 0, mid);

        if (array.length - mid >= 0)
            System.arraycopy(array, mid, rightArray,
                    0, array.length - mid);

        leftArray = mergeSort(leftArray);
        rightArray = mergeSort(rightArray);
        return merge(leftArray, rightArray, array);
    }

    public static int[] merge(int[] leftArray, int[] rightArray, int[] array){
        int i = 0, j = 0, k = 0;

        while (i < leftArray.length && j < rightArray.length) {
            if (leftArray[i] <= rightArray[j]) {
                array[k++] = leftArray[i++];
            } else {
                array[k++] = rightArray[j++];
            }
        }
        while (i < leftArray.length) {
            array[k++] = leftArray[i++];
        }
        while (j < rightArray.length) {
            array[k++] = rightArray[j++];
        }

        return  array;
    }
```

2.3 Sorting Algorithm 3 : Counting Sort

```java
public static int[] countingSort(int[] inputArray, int N) {

        int M = 0;

        for (int i = 0; i < N; i++) {
            M = Math.max(M, inputArray[i]);
        }

        int[] countArray = new int[M + 1];

        for (int i = 0; i < N; i++) {
            countArray[inputArray[i]]++;
        }
```

```

        for (int i = 1; i <= M; i++) {
            countArray[i] += countArray[i - 1];
        }

        int[] outputArray = new int[N];

        for (int i = N - 1; i >= 0; i--) {
            outputArray[countArray[inputArray[i]] - 1] = inputArray[i];
            countArray[inputArray[i]]--;
        }

        return outputArray;
    }
```

2.4 Searching Algorithm 1 : Linear Search

```
int LineerSearch(int[] arr, int value){
        int size = arr.length;
        for(int i=0;i<=size-1;i++){
            if(arr[i]==value){
                return i;
            }
        }
        return -1;

    }
```

2.5 Searching Algorithm 2 : Binary Search

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

```
        else if (arr[high]==value){
            return high;
        }
        return -1;
    }
```

3 Results, Analysis, Discussion

In this section, the results obtained from the "TrafficFlowDataset.csv" file according to the sorting algorithms mentioned above will be shown in the table and graphically.

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	0,16044	0,14774	0,39287	1,05689	3,47423	12,93149	49,70701	196,06686	823,97442	3499,56287
Merge sort	0,05177	0,09269	0,20351	0,26374	0,56288	1,18089	2,62539	5,06293	10,8293	21,18562
Counting sort	64,50735	58,07284	57,29936	53,84098	54,45965	54,28739	54,39911	55,40725	56,23472	58,00466
	Sorted Input Data Timing Results in ms									
Insertion sort	4,60000	7,30000	0,0015	0,00283	0,00563	0,01122	0,02234	0,04492	0,08946	0,17447
Merge sort	0,0177	0,03463	0,07194	0,15104	0,32007	0,67331	1,36799	2,75200	5,35052	11,61932
Counting sort	53,96037	53,90328	53,86141	53,98759	54,19196	54,19583	54,28936	54,7337	55,00621	55,97061
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0,02643	0,09178	0,3462	01,3468	5,38093	23,36668	97,24778	394,17785	1598,92477	6129,53246
Merge sort	0,01179	0,02527	0,05156	0,11236	0,22239	0,46909	1,02693	2,00707	3,98084	8,45256
Counting sort	54,16274	53,34742	53,94264	53,93019	53,06282	53,46155	54,32317	54,6653	54,4867	55,9981

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)	101,3	138,4	207,6	324,5	687,0	1085,3	1874,5	4062,9	8066,8	12494,3
Linear search (sorted data)	98,7	126,5	187,8	336,4	614,3	1175,1	2294,6	4535,9	9246,8	17671,1
Binary search (sorted data)	115,2	125,6	141,9	153,2	159,9	168,2	176,0	188,2	198,2	233,8

Complexity analysis tables to complete (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	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

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	$O(1)$

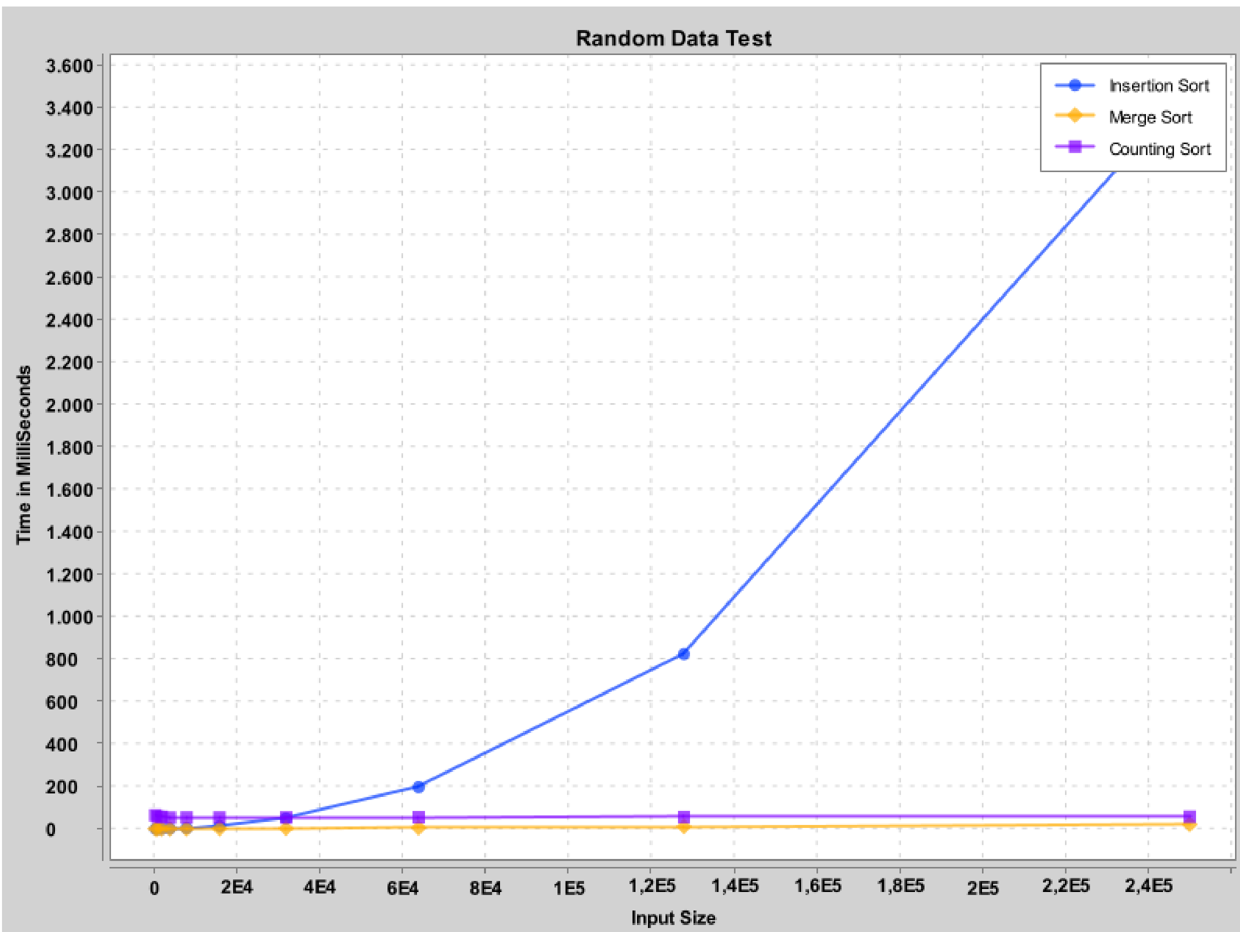

Figure 1: Plot of the random dataset.

When we observe the results and graphs of sorting algorithms, they do not always perform as accurately as theory would suggest. This discrepancy could be attributed to several factors, including the capabilities of the computer running the project or the code not being written to the

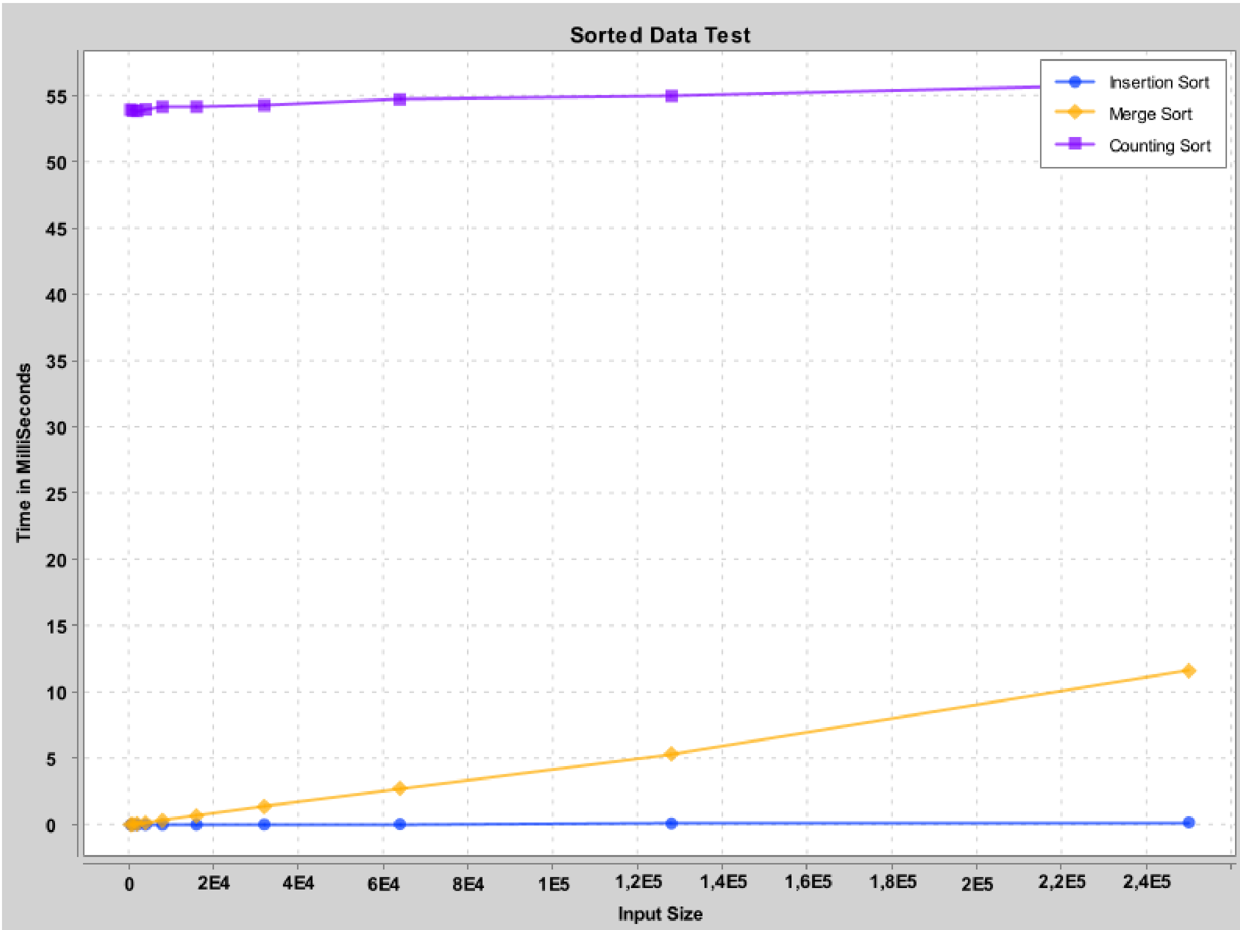

Figure 2: Plot of the sorted dataset.

desired standard. Nevertheless, we still managed to achieve a result that is close to generalized expectations. Some conclusions that can be drawn from this experience include:

Sometimes, for small datasets, sorting algorithms that do not appear very efficient in theory can still be used effectively. Sorting algorithms that seem highly efficient for ordered datasets may not always perform as well in practice. For ordered datasets, if the dataset size increases, utilizing binary search for searching operations is definitively efficient. Expanding on this, it's important to recognize that theoretical efficiency and real-world performance can diverge due to practical considerations like hardware limitations, programming language overheads, and specific characteristics of the data being processed. This insight underscores the value of empirical testing in addition to theoretical analysis when evaluating sorting algorithms. Moreover, it highlights the necessity of adapting algorithmic choices to the specific requirements and constraints of each application, rather than relying solely on generalized assumptions about performance.

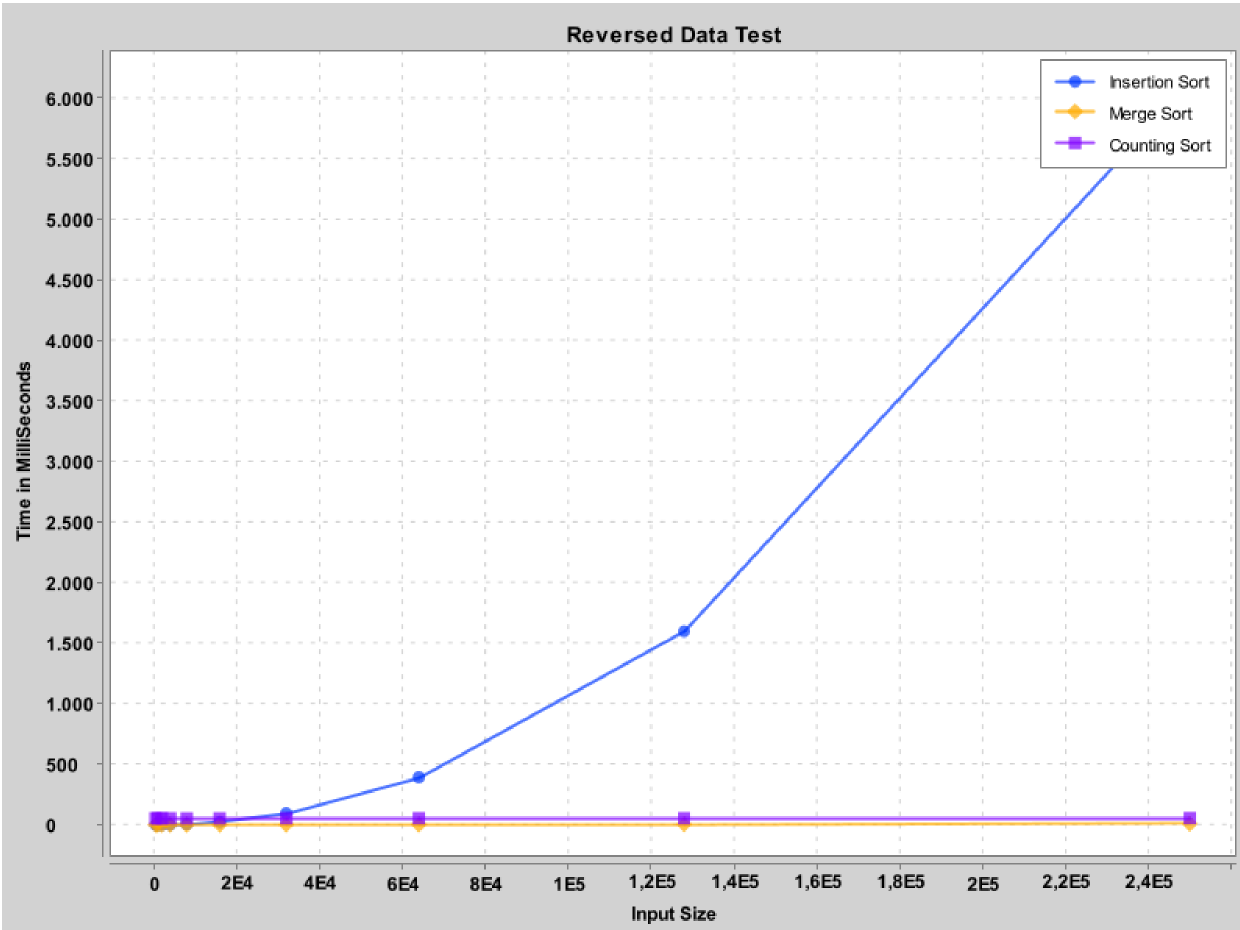

Figure 3: Plot of the reversely sorted dataset.

References

- Sorting Algotihm, Wikipedia
- N. Faujdar and S. P. Ghrera, "Analysis and Testing of Sorting Algorithms on a Standard Dataset," 2015 Fifth International Conference on Communication Systems and Network Technologies, 2015, pp. 962-967.
- G. Batista, "Big O," Towards Data Science, Nov 5. 2018

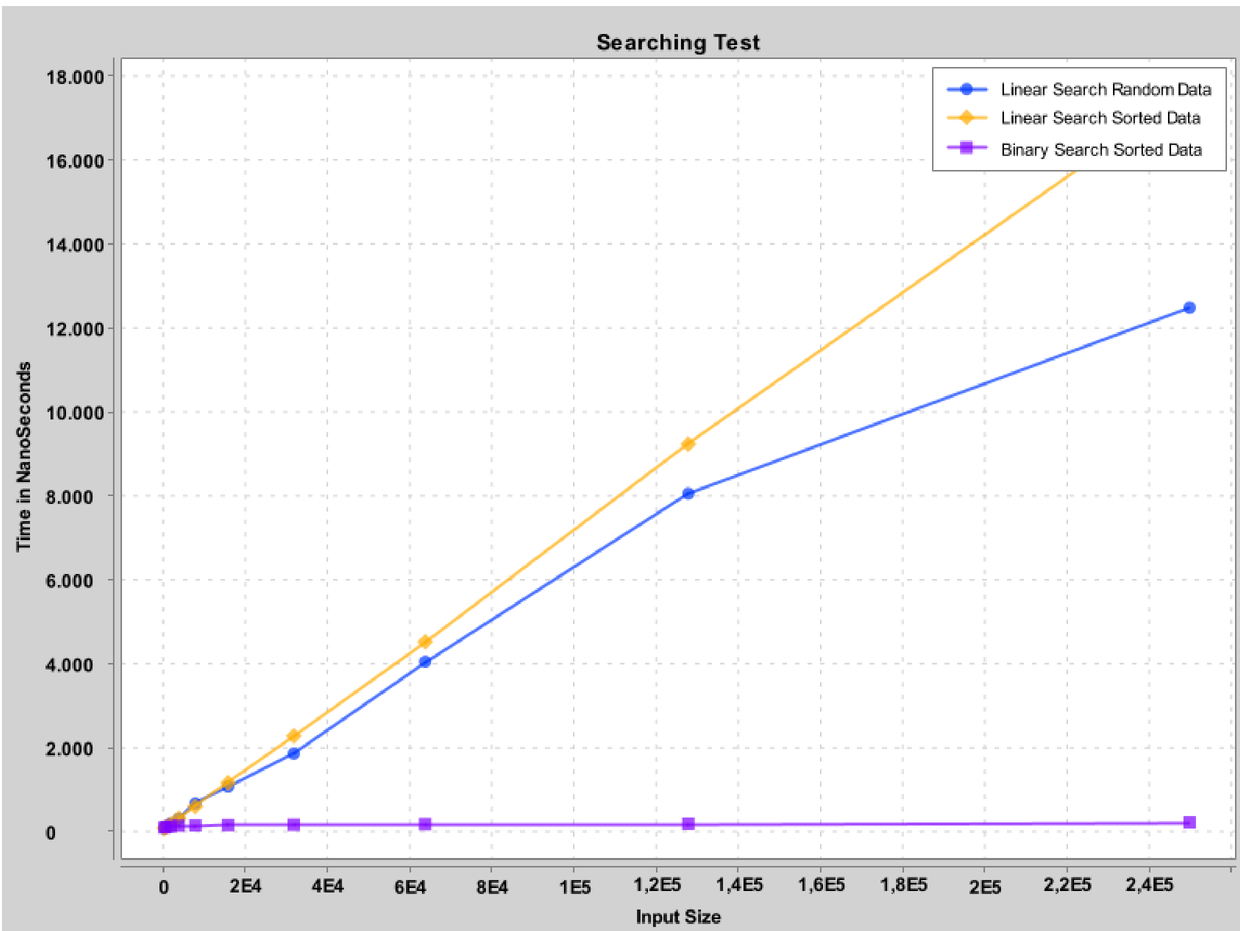

Figure 4: Plot of the searching algorithms.