Hacettepe University
Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 17, 2024

Student Name:
Yusuf Demir

Student Number:
b2210356074

Contents

1 Problem Definition

Sorting and search algorithms are fundamental to computer science, finding applications in data organization, targeted data retrieval, and financial analytics. While search algorithms locate specific items within data structures, sorting algorithms arrange elements within lists or arrays according to a defined order. Understanding the characteristics and performance of these algorithms is crucial for effective problem-solving. Therefore, it's essential to select the most suitable algorithm based on the specific requirements of a given task. In this report, we analyze the computational complexities and performance characteristics of various sorting and search algorithms, including insertion sort, merge sort, counting sort, linear search, and binary search, through empirical experiments conducted on random, sorted, and reversed datasets of varying sizes. By evaluating their running times and space complexities, we aim to gain insights into their practical utility and efficiency in real-world scenarios.

2 Solution Implementation

2.1 Insertion Sort Algorithm

```
class InsertionSort {
    public void sort(int[] arr) {
        for (int j = 1; j < arr.length; 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;
        }
    }
}
```

Insertion sort is a straightforward sorting algorithm where the sorted array is built one element at a time. It's efficient for small datasets and particularly effective for nearly sorted data. The provided Java code showcases the implementation of the insertion algorithm. The process involves iterating through the array, comparing each element with those before it, and shifting larger elements to the right until the list is sorted. Being an in-place algorithm, insertion sort sorts the elements within the original array without requiring extra memory allocation. This makes it memory-efficient compared to other sorting algorithms that may need additional space proportional to the input size.

2.2 MergeSort Algorithm

```java
public class MergeSort {

    public int[] sort(int[] arr) {
        return mergeSort(arr);
    }

    private int[] mergeSort(int[] arr) {
        if (arr.length <= 1) {
            return arr;
        }

        int middle = arr.length / 2;
        int[] left = Arrays.copyOfRange(arr, 0, middle);
        int[] right = Arrays.copyOfRange(arr, middle, arr.length);

        left = mergeSort(left);
        right = mergeSort(right);

        return merge(left, right);
    }

    private int[] merge(int[] left, int[] right) {
        int[] result = new int[left.length + right.length];
        int leftIndex = 0, rightIndex = 0, resultIndex = 0;

        while (leftIndex < left.length && rightIndex < right.length) {
            if (left[leftIndex] <= right[rightIndex]) {
                result[resultIndex++] = left[leftIndex++];
            } else {
                result[resultIndex++] = right[rightIndex++];
            }
        }

        while (leftIndex < left.length) {
            result[resultIndex++] = left[leftIndex++];
        }

        while (rightIndex < right.length) {
            result[resultIndex++] = right[rightIndex++];
        }

        return result;
    }
}
```

Merge Sort is a divide-and-conquer algorithm that systematically breaks down the array into smaller subarrays until each subarray contains only one element. These individual elements are then merged together, gradually constructing a fully sorted array. Initially, the array is divided into halves recursively until each subarray has only one element, which inherently is sorted and requires no further division. The merging step involves intelligently comparing and arranging the elements of these sorted subarrays to produce the final sorted array.

2.3 Counting Sort Algorithm

```
58 public class CountingSort {
59     public int[] sort(int[] arr) {
60         int max = getMax(arr);
61         int[] count = new int[max + 1];
62         int[] output = new int[arr.length];
63 
64         for (int num : arr)
65             count[num]++;
66 
67         for (int i = 1; i <= max; i++)
68             count[i] += count[i - 1];
69 
70         for (int i = arr.length - 1; i >= 0; i--) {
71             output[count[arr[i]] - 1] = arr[i];
72             count[arr[i]]--;
73         }
74 
75         System.arraycopy(output, 0, arr, 0, arr.length);
76         return arr;
77     }
78 
79     private int getMax(int[] arr) {
80         int max = Integer.MIN_VALUE;
81         for (int num : arr) {
82             if (num > max) {
83                 max = num;
84             }
85         }
86         return max;
87     }
88 }
```

Counting Sort operates by counting the occurrences of each unique element in the input array and leveraging this information to organize the elements in ascending order. It excels when the range of elements is limited compared to the array's size. This approach involves creating a frequency array to tally the occurrences of each element. Then, by iterating through the input array and referencing the frequency array, Counting Sort efficiently positions each element in its sorted position. This

algorithm's efficiency stems from its ability to directly map each element to its correct position without any comparisons, making it particularly suitable for scenarios with a constrained range of values.

2.4 Linear Search Algorithm

```
class LinearSearch extends SearchAlgorithm {
    @Override
    public int search(int[] arr, int x) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == x) {
                return i;
            }
        }
        return -1;
    }
}
```

Linear Search is a basic search algorithm that sequentially examines each element in a list until finding the target. It starts from the first element and compares each element to the target one by one. If a match is found, it returns the index of the element; otherwise, it continues searching. When all elements have been checked without finding a match, the algorithm concludes that the target is not present and returns -1.

2.5 Binary Search Algorithm

```
class BinarySearch extends SearchAlgorithm {
    @Override
    public int search(int[] arr, int x) {
        int low = 0;
        int high = arr.length - 1;

        while (high - low > 1) {
            int mid = (high + low) / 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 efficiently locates elements in a sorted array by repeatedly dividing the search range in half and narrowing down the potential locations for the target. Initially, it compares the target with the middle element of the array. If they match, the search is successful. Otherwise, it continues in the appropriate half of the array based on the comparison result. This process iterates until the target is found or the search interval becomes empty.

3 Results, Analysis, Discussion

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

Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion sort	0	0	0	1	3	16	55	227	934	3984
Merge sort	0	0	0	0	1	1	2	6	12	29
Counting sort	134	104	103	103	103	103	103	104	106	107
	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	1	1	1	2	7	16
Counting sort	112	105	103	103	105	103	106	104	106	105
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	2	7	28	107	439	1754	7037
Merge sort	0	0	0	0	0	0	1	2	5	14
Counting sort	129	104	103	103	104	104	104	104	109	105

Upon analyzing the data from the random list, insertion sort exhibits a steep increase in running time as the input size grows, aligning with its $O(n^2)$ time complexity. This indicates decreasing efficiency for larger datasets. In contrast, merge sort and counting sort show more controlled increases in running times, reflecting their $O(n \log n)$ and $O(n+k)$ time complexities, respectively. Merge sort's efficiency remains consistent due to its divide-and-conquer approach, while counting sort's performance may vary depending on the range of values in the dataset.

In the sorted list scenario, insertion sort demonstrates minimal increases in running time, consistent with its best-case $O(n)$ time complexity. Merge sort maintains efficiency with a similar growth rate to the random data scenario, attributed to its $O(n \log n)$ time complexity. However, counting sort's performance declines, in line with its $O(n + k)$ time complexity, as it relies on counting occurrences within a specific range.

In the reversely sorted list, insertion sort's running time increases steeply, reflecting its $O(n^2)$ time complexity. Merge sort and counting sort show controlled increases, similar to their behavior with random data, reflecting their respective time complexities of $O(n \log n)$ and $O(n + k)$. These

algorithms remain efficient choices for handling larger datasets, even with reversely sorted data.

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

Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	474	185	271	430	709	1115	2217	4420	8318	14446
Linear search (sorted data	418	208	319	466	820	1697	3180	6647	12300	23197
Binary search (sorted data)	237	222	251	231	229	231	277	508	813	1115

Linear search maintains consistent performance on randomly sorted and already sorted data, with a time complexity of $O(n)$, where n is the input size. Therefore, its running time increases linearly with input size in both scenarios.

In contrast, binary search on sorted data exhibits logarithmic growth in running time due to its time complexity of $O(logn)$, where n is the input size. This logarithmic growth makes binary search more efficient for larger datasets compared to linear search's linear growth.

Table 3: Table 3: Computational complexity comparison of the given algorithms.

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

Table 4: 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	$O(1)$

Insertion sort, with its minimal memory usage and O(1) space complexity, is suitable for small datasets or nearly sorted arrays due to its simplicity and efficiency. Merge sort's efficient divide-and-conquer approach and O(n log n) time complexity make it ideal for large datasets, especially when stable sorting and predictable performance are required. Counting sort, with its linear time complexity O(n + k), is optimal for sorting integers within a limited range, making it suitable for scenarios where the range of input values is known and space efficiency is crucial. Linear search is appropriate for unsorted or small datasets due to its simplicity and linear time complexity O(n). Binary search, with its logarithmic time complexity O(log n), excels in searching sorted arrays, providing fast and efficient retrieval of elements. These considerations should guide the selection of algorithms based on the specific characteristics of the data and performance requirements.

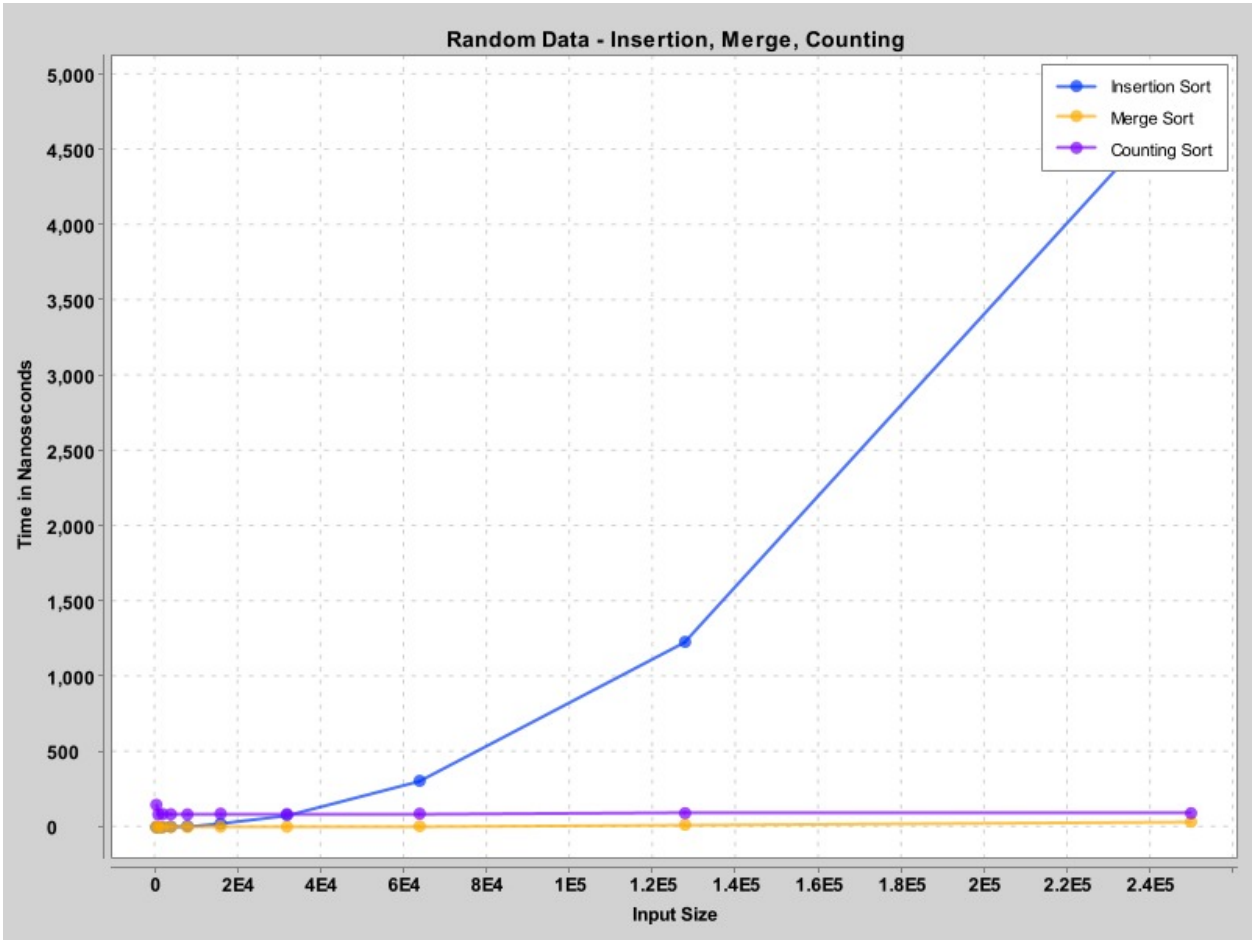

Figure 1: Algorithms Running Times - Random Data Set

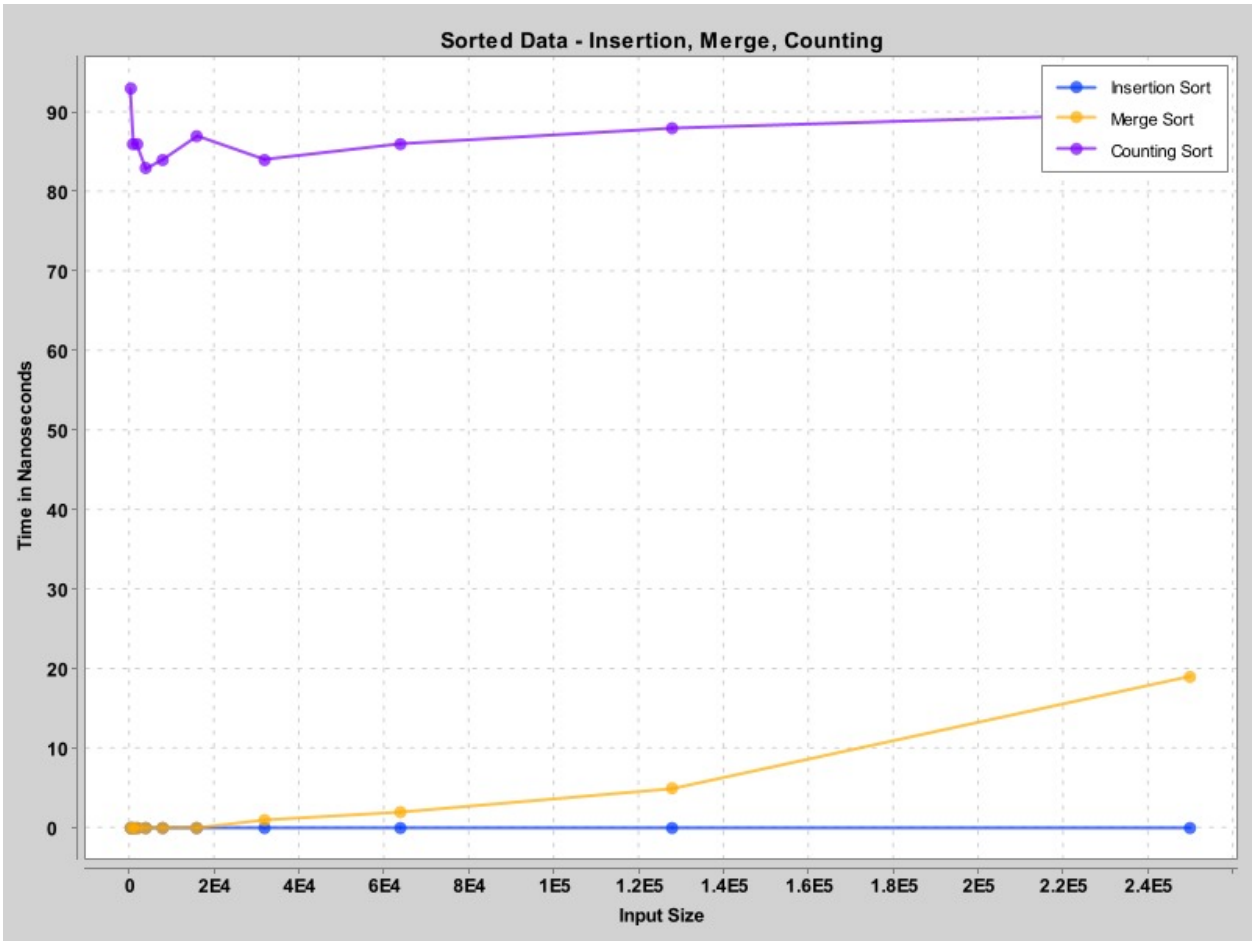

Figure 2: Algorithms Running Times - Sorted Data Set

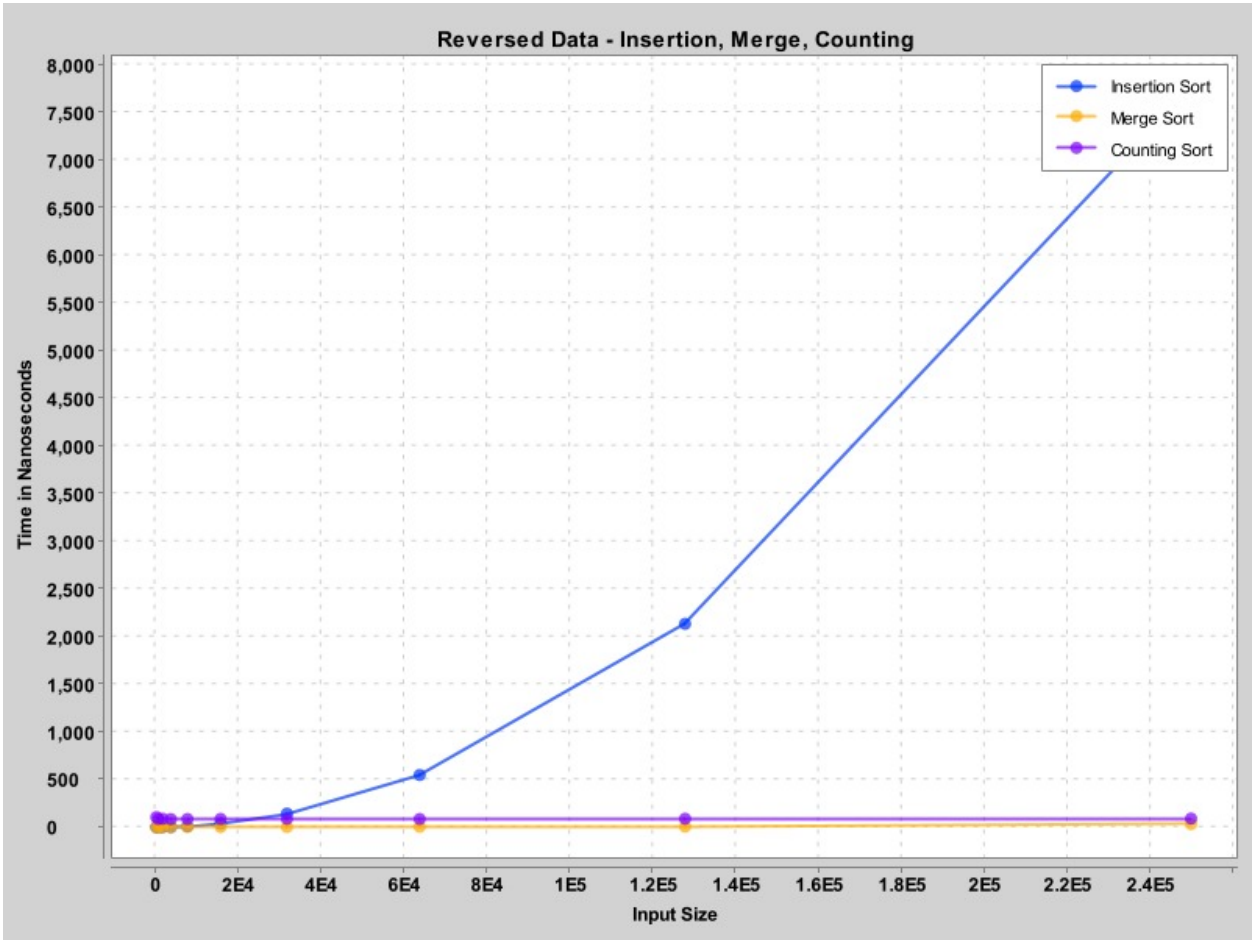

Figure 3: Algorithms Running Times - Reverse Sorted Data Set

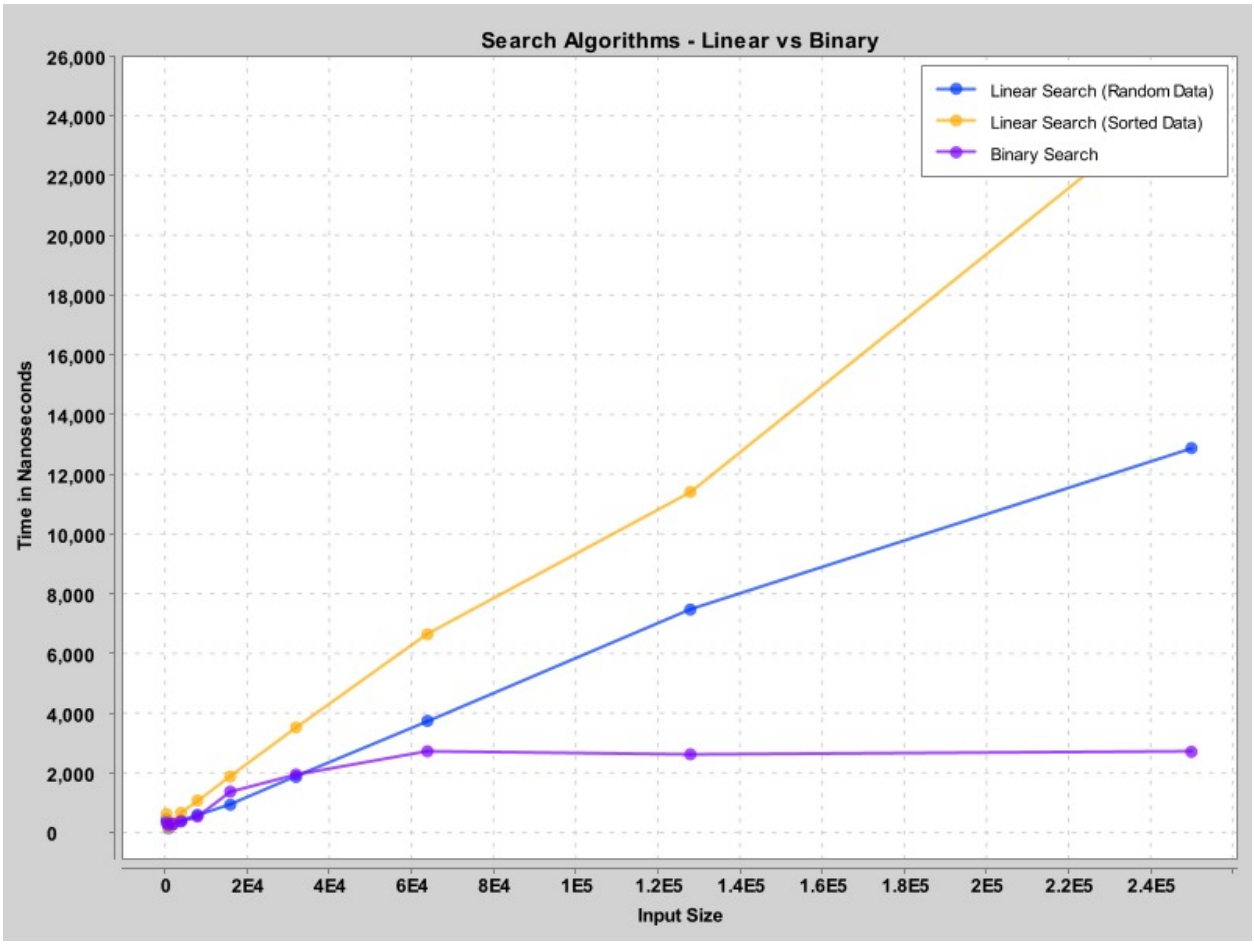

Figure 4: Search Algorithms Running Times - Random&Sorted Data Set

References

- https://www.javatpoint.com/insertion-sort
- https://www.javatpoint.com/merge-sort
- https://www.geeksforgeeks.org/counting-sort/
- https://www.freecodecamp.org/news/search-algorithms-linear-and-binary-search-
- https://www.javatpoint.com/searching-algorithms