Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 19, 2024

Student name:
Bedirhan Gençaslan

Student Number:
b2210356065

1 Problem Definition

We are trying to measure the performance of various sorting and searching algorithms on different data types.

By examining the massive data provided to us through various algorithms, we aim to find the most efficient data processing method. We measure how long it takes to analyze these data types based on input size and then analyze them in a graph. Finally, we compare the algorithms with each other.

2 Solution Implementation

The methods we follow for the solution are as follows:

-Writing a reader code to extract data from the massive data structure up to the input size.
-Implementing algorithms that can process this data.
-Storing all outputs using functions that measure the running times of the algorithms.
-Processing the stored times on a graph for comparison.

Algorithms we will implement:
-Sorting algorithms:
Insertion Sort
Merge Sort
Counting Sort
-Searching algorithms:
Linear Search
Binary Search

2.1 Insertion Sort Algorithm

Insertion Sort is a simple sorting algorithm that works by building a sorted list one element at a time. It maintains a sub-array of already sorted elements and iteratively inserts new elements into the sorted sub-array, shifting elements as needed to make room for the new element.

```
1     public static Integer[] Insertion(Integer []data) throws IOException {
2         int n = data.length;
3         for (int i = 1; i < n; ++i) {
4             int key = data[i];
5             int j = i - 1;
6
7             while (j >= 0 && data[j] > key) {
8                 data[j + 1] = data[j];
9                 j = j - 1;
10            }
11            data[j + 1] = key;
```

```
        }
        return data;
    }
```

2.2 Merge Sort Algorithm

The Merge Sort algorithm is a classic divide-and-conquer sorting algorithm that works by recursively dividing the input array into smaller sub-arrays, sorting them individually, and then merging them back together in a sorted manner. It is known for its stability and guaranteed time complexity of O(n log n) in all cases.

```
    public static Integer[] Merge(Integer []data) throws IOException {
        if (data.length > 1) {
            int size = data.length / 2;
            Integer[] left = new Integer[size];
            Integer[] right = new Integer[data.length - size];
            for (int i = 0; i < size; i++) {
                left[i] = data[i];
            }
            for (int i = 0; i < data.length - size; i++) {
                right[i] = data[i + size];
            }

            left = Merge(left);
            right = Merge(right);
            return Mergesort(left,right,data);
        }
        return data;
    }
    public static Integer[] Mergesort(Integer[] left,Integer[] right,Integer
        []data) throws IOException {
        int leftn = 0;
        int rightn = 0;
        int dataIndex = 0;

        while (leftn < left.length && rightn < right.length) {
            if (left[leftn] <= right[rightn]) {
                data[dataIndex] = left[leftn];
                leftn++;
            } else {
                data[dataIndex] = right[rightn];
                rightn++;
            }
            dataIndex++;
        }

        while (leftn < left.length) {
```

```
            data[dataIndex] = left[leftn];
            leftn++;
            dataIndex++;
        }

        while (rightn < right.length) {
            data[dataIndex] = right[rightn];
            rightn++;
            dataIndex++;
        }

        return data;
    }
```

2.3 Counting Sort Algorithm

The Counting Sort algorithm is a non-comparison-based sorting algorithm that sorts elements by counting the number of occurrences of each element and using this information to place them in the correct order. It is particularly useful when the range of input elements is known and relatively small compared to the number of elements to be sorted.

```
    public static Integer[] Counting(Integer []data) throws IOException {
        if (data.length <= 1) {
            return data;
        }
        int max = data[0];
        int min = data[0];
        for (int num : data) {
            if (num > max) {
                max = num;
            }
            if (num < min) {
                min = num;
            }
        }

        int[] count = new int[max - min + 1];
        for (int num : data) {
            count[num - min]++;
        }

        int index = 0;
        for (int i = 0; i < count.length; i++) {
            while (count[i] > 0) {
                data[index++] = i + min;
                count[i]--;
            }
```

```
89          }
90          return data;
91      }
```

2.4 Linear Search Algorithm

The Linear Search algorithm, also known as Sequential Search, is a simple searching algorithm that checks each element in a list one by one until the target element is found or the end of the list is reached. It is applicable to both sorted and unsorted lists, although it is more efficient on sorted lists when compared to unsorted ones.

```
92      public static Integer Linear(Integer []data,int aranan) throws IOException
            {
93          for(int i=0;i<data.length;i++){
94              if(data[i]==aranan){
95                  return i;
96              }
97          }
98          return -1;
99      }
```

2.5 Binary Search Algorithm

The Binary Search algorithm is an efficient search algorithm used to find a specific target element in a sorted array or list. It follows a divide-and-conquer approach by repeatedly dividing the search interval in half until the target element is found or the search interval is empty.

```
100     public static Integer Binary(Integer []data,int aranan) throws IOException
            {
101         int sol = 0;
102         int sag = data.length - 1;
103         while (sol <= sag) {
104             int orta = sol + (sag - sol) / 2;
105             if (data[orta] == aranan) {
106                 return orta;
107             }
108             if (data[orta] < aranan) {
109                 sol = orta + 1;
110             } else {
111                 sag = orta - 1;
112             }
113         }
114         return -1;
115     }
```

3 Results, Analysis, Discussion

The sorting algorithms' completion times for data sizes specified in the table are calculated by repeating the same experiment 10 times and taking the average of the results in milliseconds (ms).

The searching algorithms' completion times for data sizes specified in the table are calculated in nanoseconds (ns), and the average time to find the specified random 1000 elements among the data is also taken into account.

The types of algorithms, data sizes, and completion times for the algorithms are as follows. 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	0.6	0.2	0.2	0.7	2.7	13.3	47.7	334.1	3928.9	19150.2
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	10.0	41.0
Counting sort	207.9	114.4	115.6	115.4	114.5	114.7	116.3	121.5	142.4	150.6
	Sorted Input Data Timing Results in ms									
Insertion sort	0.1	0.0	0.0	0.0	0.1	0.0	0.2	0.2	2.2	3.0
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	20.0	30.0
Counting sort	153.5	114.3	115.0	116.2	116.7	113.3	114.9	117.5	123.4	128.2
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.4	1.0	1.7	7.1	28.6	114.3	459.0	1834.6	7340.6	27030.7
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	10.0	30.0
Counting sort	115.9	116.5	116.3	117.4	113.7	116.6	115.8	117.0	123.9	127.5

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)	2803.8	340.5	554.8	1341.3	2511.4	5105.1	9925.5	19669.1	44211.7	110847.3
Linear search (sorted data)	1893.3	246.2	581.3	1318.4	2520.1	4691.7	9945.7	19887.7	41677.9	106163.2
Binary search (sorted data)	704.8	240.9	275.6	149.6	176.7	169.6	197.9	320.2	283.2	359.1

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	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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	$O(n)$

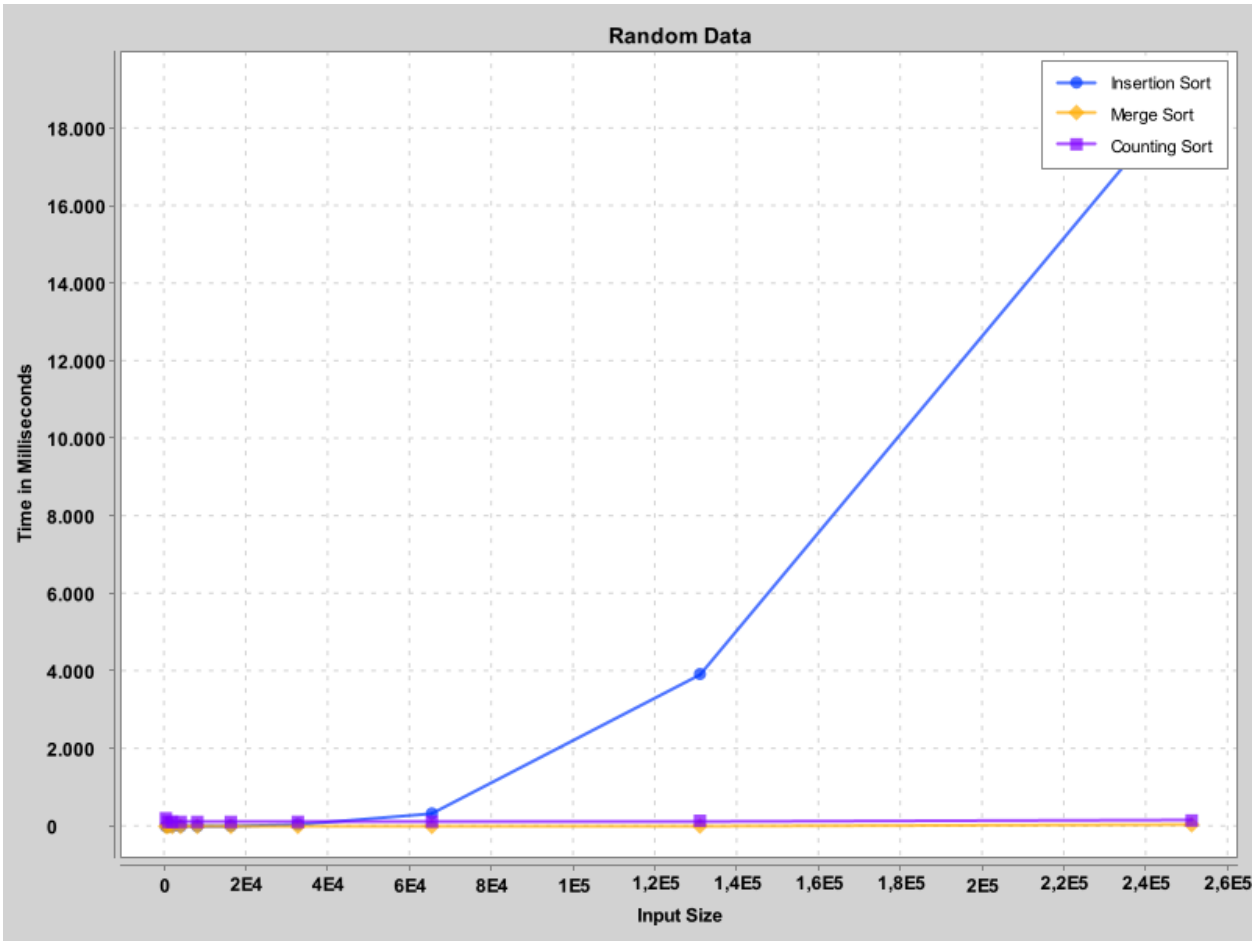

Figure 1: Plot of the functions.

When sorting random data arrays, the time complexities are as mentioned earlier: $O(n^2)$ for Insertion Sort, $O(n \log n)$ for Merge Sort, and $O(n+k)$ for Counting Sort. As seen in the graph, the time taken by Insertion Sort, indicated by the blue line, is much higher compared to Merge

Sort and Counting Sort. This is due to the massive size of the data arrays we are dealing with and the exponential nature of Insertion Sort.

In contrast, Merge Sort and Counting Sort, although not clearly visible in the graph but evident in the table, show a distinct difference in time. This is because although Counting Sort has a lower time complexity, it deals with numbers in vastly different ranges (i.e., it maintains a large value for k), which makes Merge Sort more efficient.

From this observation, we can infer that if we have highly independent numbers in very large data structures, Merge Sort is the most efficient algorithm. However, if the numbers are closely spaced, Counting Sort becomes a sensible choice. 1.

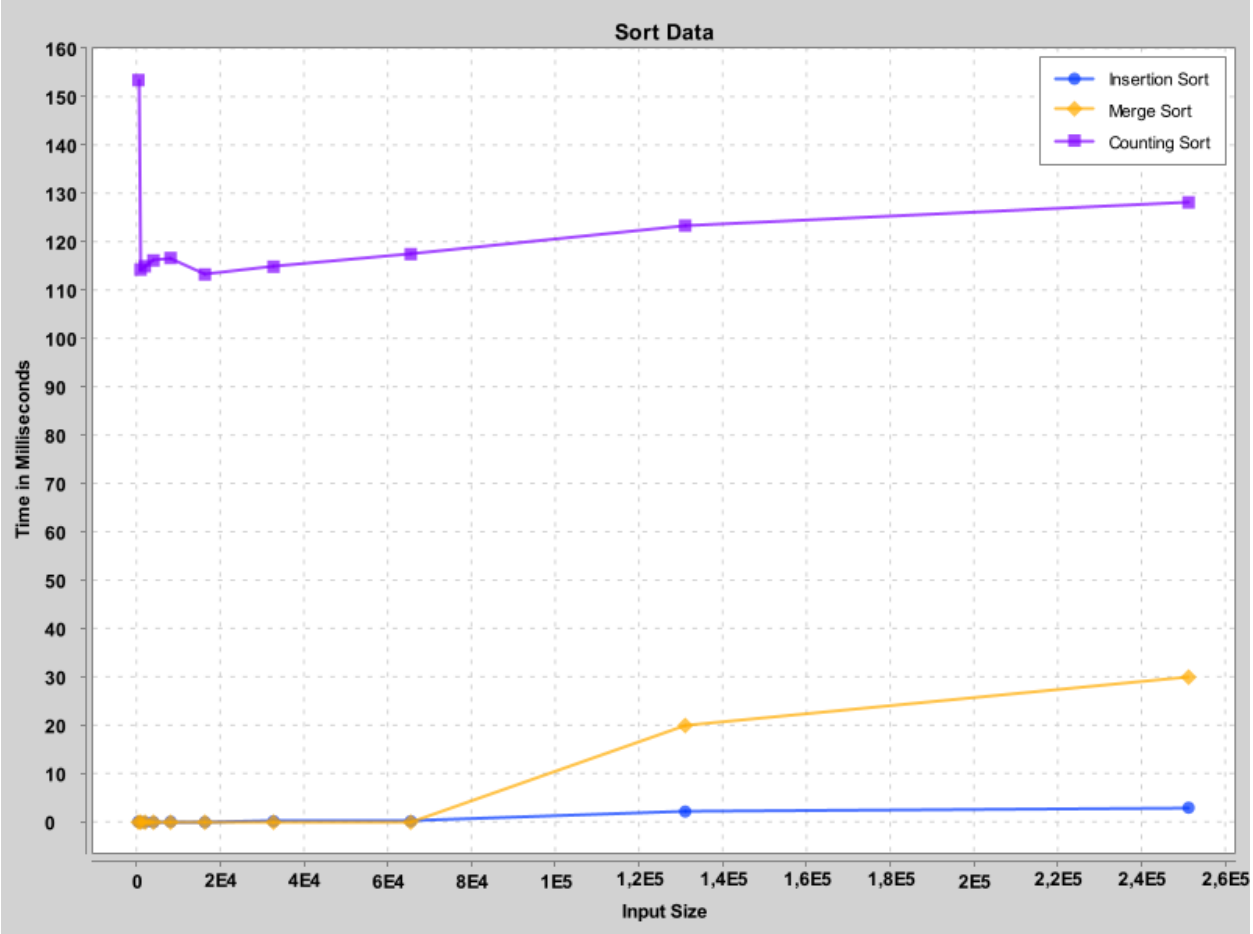

Figure 2: Plot of the functions.

When running sorting algorithms within sorted data arrays, the time complexities are as follows: best case $\Omega(n)$ for Insertion Sort, $O(n \log n)$ for Merge Sort, and $O(n + k)$ for Counting Sort. Therefore, among the sorting algorithms within sorted data arrays, the only algorithm that will show a difference in terms of time is Insertion Sort.

As seen in the graph, Insertion Sort has a much lower time complexity compared to Merge Sort in terms of time complexity. This is because Insertion Sort's time complexity in the best case scenario is $\Omega(n)$, which makes Insertion Sort have the lowest values in terms of time complexity in this scenario. Next in terms of time efficiency is Merge Sort, followed by Counting Sort.

The interesting aspect in the graph is the significantly larger time taken by Counting Sort, despite it being able to sort very small and very large data arrays in the same time. The explanation for this interesting situation lies in the time complexity of Counting Sort, which is $O(n + k)$. In Counting Sort, it is not the size of the data array that determines the time efficiency, but rather the difference between the smallest and largest elements in the data. Therefore, regardless of how small the data array is, if the difference between the numbers in the array is large, Counting Sort does not work efficiently. 2.

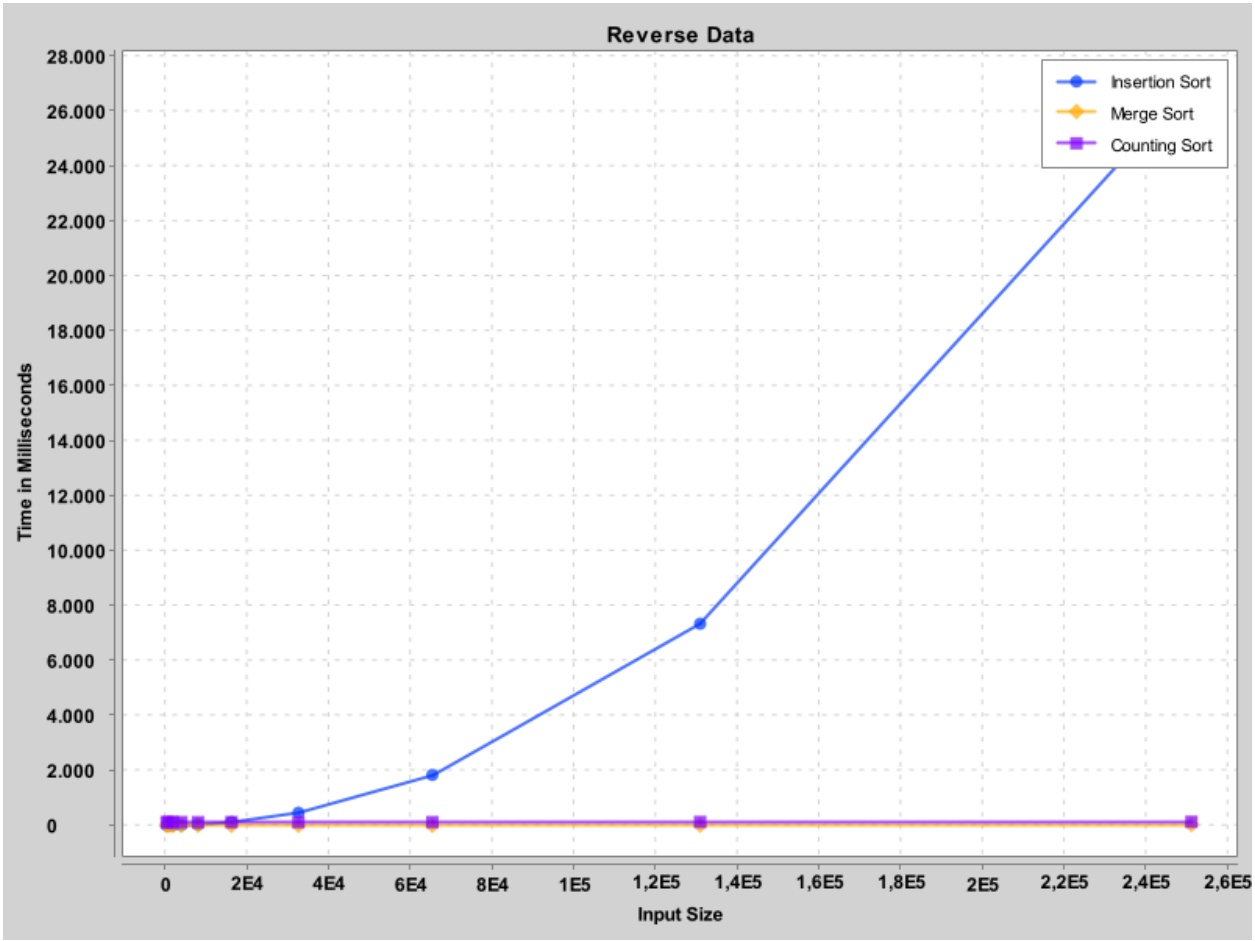

Figure 3: Plot of the functions.

In a reverse-sorted data array, the time complexities are as follows: $O(n^2)$ for Insertion Sort, $O(n \log n)$ for Merge Sort, and $O(n + k)$ for Counting Sort, as mentioned earlier. In fact, Merge

Sort and Counting Sort have the same time complexities as in the sorted data array above, because these two algorithms work regardless of the order of the data array. Therefore, they are in the same position in terms of time complexity compared to each other. However, we cannot say the same for Insertion Sort.

Insertion Sort is a sorting algorithm that is dependent on the order of the data array, and a reverse-sorted data type is the worst-case scenario for the Insertion Sort algorithm. This is because Insertion Sort, by its working logic, finds the smallest element while traversing the array from the beginning to the end, resulting in a time complexity of $O(n^2)$, which is exponential. Therefore, in very large data structures, Insertion Sort will work much less efficiently and take a longer time compared to Merge Sort and Counting Sort.. 3.

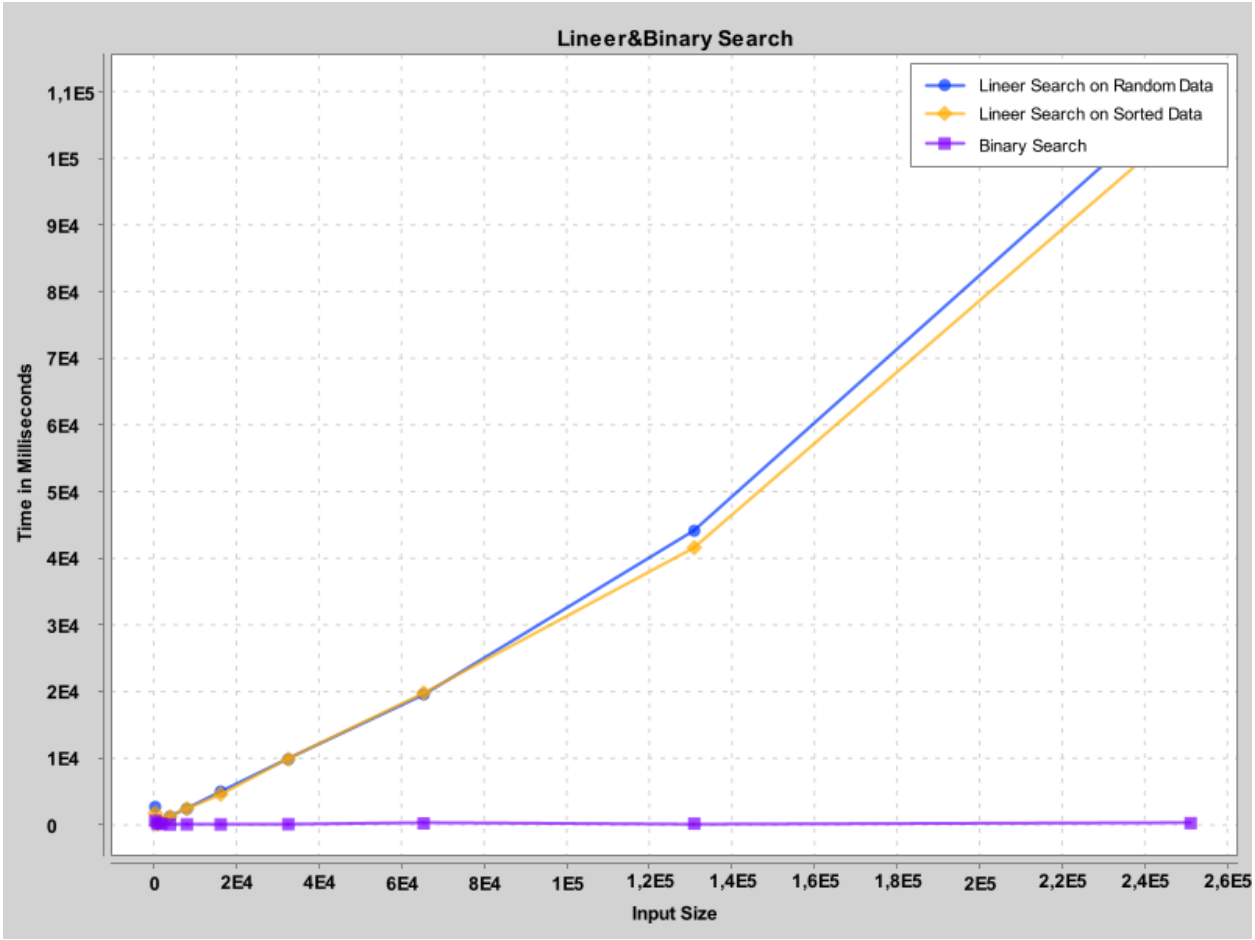

Figure 4: Plot of the functions.

When searching in data arrays, we use two different search algorithms with different time complexities. One of them is Linear Search ($O(n)$ time complexity), which works independently of whether the array is sorted or not. This search method traverses all indices until it finds the

desired index, and statistically, the time spent searching for the index will be almost the same in both cases. As seen in the graph, this logic results in a consistent outcome.

The other search method is Binary Search, which works on sorted data arrays. It compares the value in the middle of the array each time and can find the desired index with $O(\log n)$ time complexity. In data arrays with a large number of elements, Binary Search provides much smaller time complexities compared to Linear Search. Therefore, if we have a sorted data array, it is more logical to prefer Binary Search. If we have an unsorted data array and are searching for a limited number of elements, Linear Search can be preferred. However, if we are trying to find a large number of elements, it is more logical to first sort the array using Merge Sort and then use Binary Search. The data in the graph aligns with what we have described.

Another interesting aspect in the graph is that the linear search graphs are not flat. This is because, to obtain statistically accurate times, we take the arithmetic average of the times for 1000 different elements in each data entry. 4.

3.1 Results and Analysis

The Insertion Sort algorithm is the least efficient algorithm for many sorting scenarios, mainly due to its exponential time complexity. For data arrays with elements that are very close in value, Counting Sort offers the possibility of sorting in nearly $O(n + k)$ time complexity when k is small, whereas in the opposite case where k is large, it will yield inefficient results with a time complexity of $O(n + k)$. In summary, if we are performing a sorting operation and have unrelated elements with a very large data array, using Merge Sort is the most logical choice because it consistently maintains a time complexity of $O(n \log n)$, which behaves almost linearly for large data structures.

If we need to perform a small number of index searches based on the array size, Linear Search can be preferred. However, if we are dealing with a significant number of elements alongside a large data array, it is more efficient to first sort the array and then use Binary Search. This is because searching with Insertion Sort in large numbers results in a time complexity of $O(n^2)$, while using Binary Search after sorting will take approximately $O(n \log n)$ time.

Upon analyzing the results from the experiments, Merge Sort stands out as the most effective algorithm for sorting, while Binary Search emerges as the most efficient algorithm for search operations.

4 Notes

-The time complexity graphs were redrawn more than 5 times.
-The time measurement functions were repeated 10 times for sorting algorithms and 1000 times for search algorithms to obtain average values and prevent potential outliers.
-The time measurement functions were used in Main.java.
-The algorithms were used in algo.java.
-The reading function was used in reader.java.

References

- Assignment PDF for algorithms
- Overleaf for Assignment report