Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 20, 2024

Student name:
Muzaffer Berke SAVAŞ

Student Number:
b2220356044

1 Problem Definition

The main purpose of this assignment is to examine sorting and searching algorithms for efficiency in time complexity, space complexity, and stability. The assignment analyzes and categorizes algorithms, comparing their performance in various scenarios.

2 Solution Implementation

There are three sorting and two searching algorithms to compare. In the below subsections there are implementations and explanations of those algorithms .

2.1 Insertion Sort

Source code of insertion sort:

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

Insertion sort is comparison based sorting algorithm. It iterates through each element in the input list, inserting it into its correct position in the already sorted part of the list. This process continues until the entire list is sorted.

2.2 Merge Sort

Source code of merge sort:

```
15 public class Sort {
16
17     public static void Merge_Sort(int[] array){
18         int length = array.length;
19         if (length <= 1){
20             return;
21         }
22         int middle = length / 2;
23         int[] leftArray = Arrays.copyOfRange(array,0,middle);
24         int[] rightArray = Arrays.copyOfRange(array,middle,array.length);
25         Merge_Sort(leftArray);
26         Merge_Sort(rightArray);
27         Merge(array,leftArray,rightArray);
28     }
29
30     private static void Merge(int[] array,int[] first, int[] second){
31         int index_1 = 0;
32         int index_2 = 0;
33         while (index_1 != first.length && index_2 != second.length){
34             if (first[index_1] > second[index_2]){
35                 array[index_1 + index_2] = second[index_2++];
36             } else{
37                 array[index_1 + index_2] = first[index_1++];
38             }
39         }
40         while (index_1 != first.length){
41             array[index_1 + index_2] = first[index_1++];
42         }
43         while (index_2 != second.length){
44             array[index_1 + index_2] = second[index_2++];
45         }
46     }
47 }
```

Merge sort is a highly efficient sorting algorithm known for its divide-and-conquer approach. It divides the unsorted array into smaller subarrays until each subarrays contains only one element. Then, it merges these subarrays in a pairwise manner, combining them into larger sorted subarrays until the entire array is sorted.

2.3 Counting Sort

Source code of counting sort:

```
48 public class Sort {
49
50     public static void Counting_Sort(int[] array){
51         int max = array[0];
52         for (int i = 1; i < array.length; i++) {
53             if (array[i] > max){
54                 max = array[i];
55             }
56         }
57         int[] count = new int[max+1];
58         for (int i = 0; i <= max; i++){
59             count[i] = 0;
60         }
61         int size = array.length;
62         int[] output = new int[size];
63
64         for (int j : array) {
65             count[j] = count[j] + 1;
66         }
67         for (int i = 1; i <= max; i++){
68             count[i] = count[i] + count[i-1];
69         }
70         for (int i = size - 1; i >= 0; i--){
71             int j = array[i];
72             output[count[j] - 1] = array[i];
73             count[j] = count[j] - 1;
74         }
75         System.arraycopy(output, 0, array, 0, size);
76     }
77 }
```

Counting sort is an efficient non-comparison-based sorting algorithm that operates by counting the number of occurrences of each unique element in the input array. It creates a counting array to store the frequency of each element, then modifies the counting array to represent the actual position of each element in the sorted output. Finally, it iterates through the input list, placing each element into its correct position in the output array based on the information stored in the counting array.

2.4 Linear Search

Source code of linear search:

```
78 public class Search {
79     public static int Linear_Search(int[] array, int val){
80         int size = array.length;
81         for(int i = 0; i < size; i++){
82             if (array[i] == val){
83                 return i;
84             }
85         }
86         return -1;
87     }
88 }
```

Linear search, also known as sequential search, is a simple searching algorithm that sequentially checks each element in a list until the target element is found or the end of the list is reached. It works well for unsorted lists or small datasets.

2.5 Binary Search

Source code of binary search:

```
89 public class Search {
90     public static int Binary_Search(int[] sortedArray, int val){
91         int low = 0;
92         int high = sortedArray.length - 1;
93         while (high - low > 1){
94             int mid = (high + low) / 2;
95             if (sortedArray[mid] < val){
96                 low = mid + 1;
97             }else {
98                 high = mid;
99             }
100        }
101        if (sortedArray[low] == val){
102            return low;
103        }else if (sortedArray[high] == val){
104            return high;
105        }
106        return -1;
107    }
108 }
```

Binary search is a highly efficient search algorithm used to locate a specific target value within a sorted list or array. It operates by repeatedly dividing the search interval in half and narrowing down the possible locations for the target element. At each step, it compares the target value with the middle element of the current interval and adjusts the search range accordingly.

3 Results, Analysis, Discussion

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	0.4	0.1	0.2	1.1	3.6	16.0	56.0	215.2	892.1	3702.8
Merge sort	1.7	0.0	0.0	0.0	1.7	1.4	3.1	6.2	12.7	23.8
Counting sort	108.7	87.9	83.3	84.0	83.1	82.5	83.6	84.9	86.1	88.0
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Merge sort	0.0	1.6	0.0	0.5	0.6	0.5	3.1	6.4	8.1	18.8
Counting sort	86.8	84.7	83.2	85.1	86.5	84.8	84.7	84.9	86.6	87.9
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.1	0.8	1.4	7.1	26.7	105.0	421.6	1706.4	6543.5
Merge sort	0.0	0.0	0.0	0.0	0.0	1.5	0.0	3.2	6.2	10.9
Counting sort	85.5	83.0	85.8	84.8	86.2	84.8	84.6	92.7	87.4	89.6

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)	3175.0	4611.3	7444.1	884.8	1159.5	2110.1	3940.3	5303.2	10500.3	18343.2
Linear search (sorted data)	124.7	142.7	207.6	344.2	595.2	1132.2	2179.9	4265.9	8974.6	16776.6
Binary search (sorted data)	346.5	84.8	157.4	123.4	131.3	131.2	376.4	58.4	65.4	299.0

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	$O(1)$

As you can see in the Table1, we can say that for time complexity of the insertion sort with random data when the input size doubles, the time is quadrupled. This fact is similar for other algorithms according to their complexities at Table3.

Table4 says that the auxiliary space complexity of Merge Sort is O(n) and auxiliary space complexity of Counting Sort is O(n + k) means that the extra space required by the algorithms.For Merge Sort apart from the input array itself, grows linearly with the size of the input. We can see that fact at 33rd, 40th and 43rd lines at Merge Sort source code. Because It is using an extra array. On the other hand Counting Sort space complexity depends on input size and it's max value. We can see this fact at 57th and 62nd lines at Counting Sort source code.

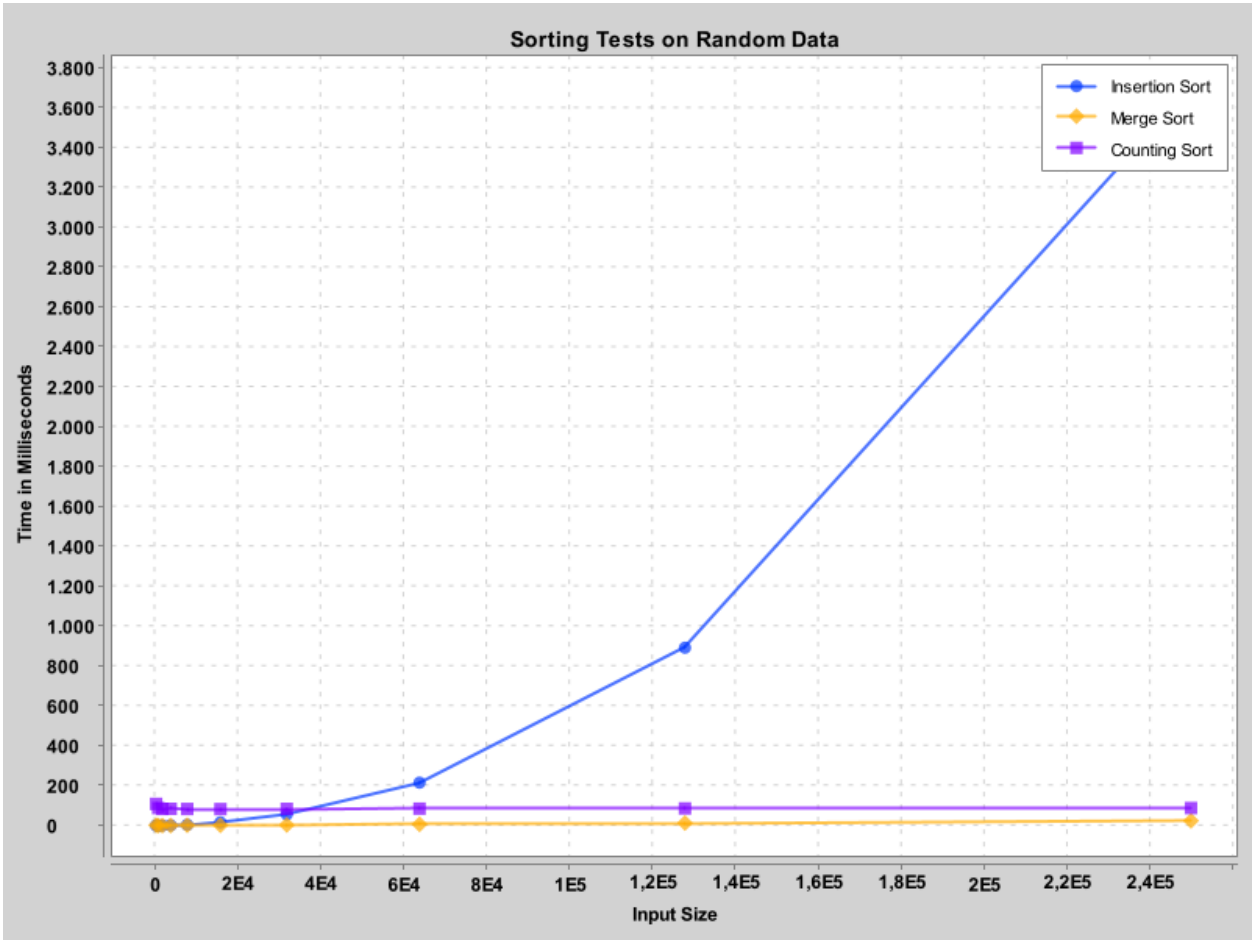

Figure 1: Time Chart of Random Data

For Insertion Sort, as the number of inputs increases, the average time taken also goes up. This method becomes quite inefficient when dealing with large inputs because it takes a lot of time to process them. On the other hand, Merge Sort consistently shows low average times across datasets of different sizes. Counting Sort also follows a similar pattern in average time as the dataset size grows. It's not as affected by the size of the input, but it doesn't perform well even with smaller datasets. This is as expected because Counting Sort has a time complexity of O(n + k).

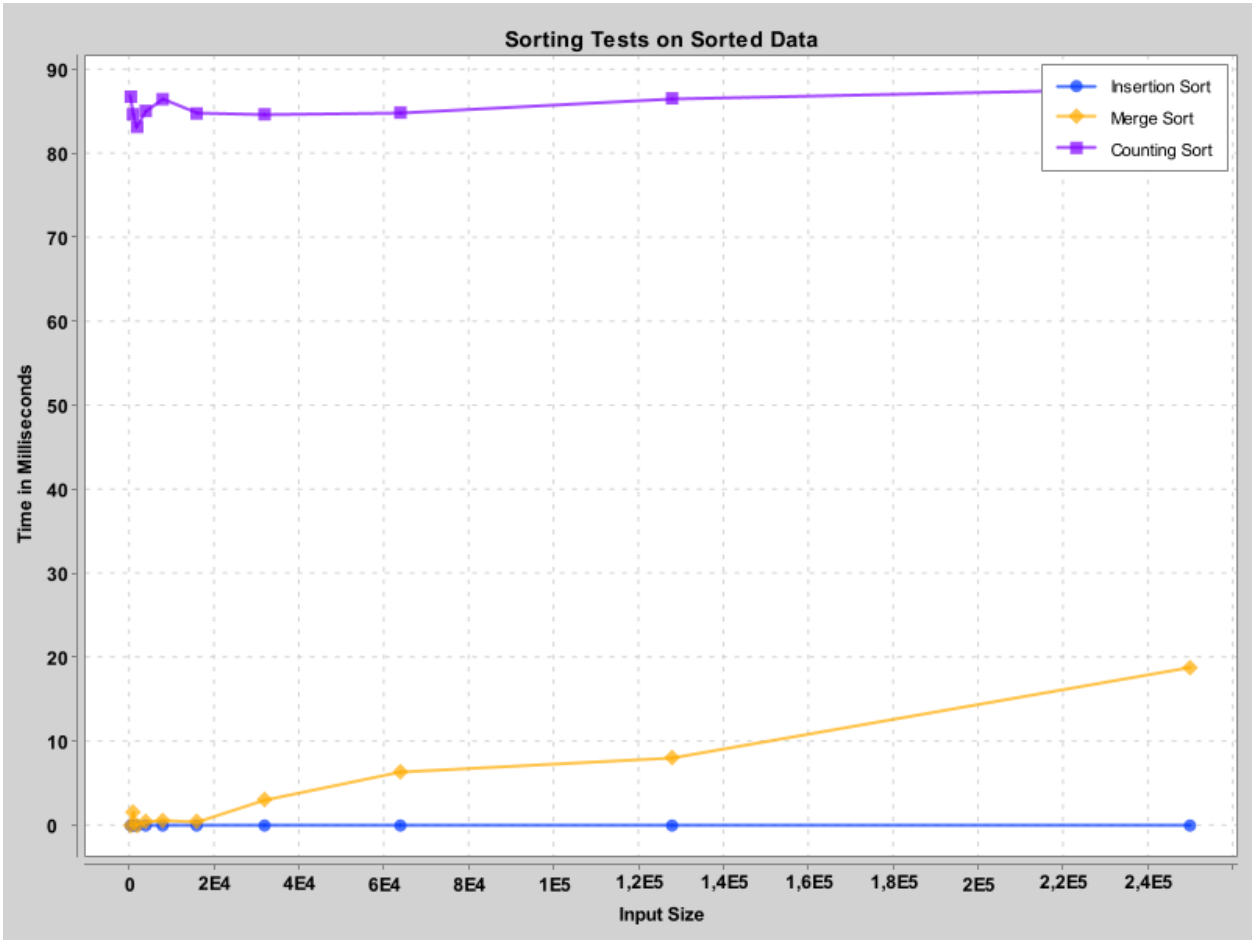

Figure 2: Time Chart of Sorted Data

When it comes to sorted data, Insertion Sort works well, showing consistently low average processing times across datasets of all sizes. It's good at handling already sorted data, making it a reliable option. Merge Sort also does a decent job, though there's a slight increase in average processing time with larger datasets. Because it's complexity is O(n logn). But it still manages to handle them effectively. On the other hand, Counting Sort doesn't perform as well, with relatively higher and inconsistent average processing times. This inconsistency can be a problem, especially with larger datasets, affecting how reliable and efficient the sorting process is.

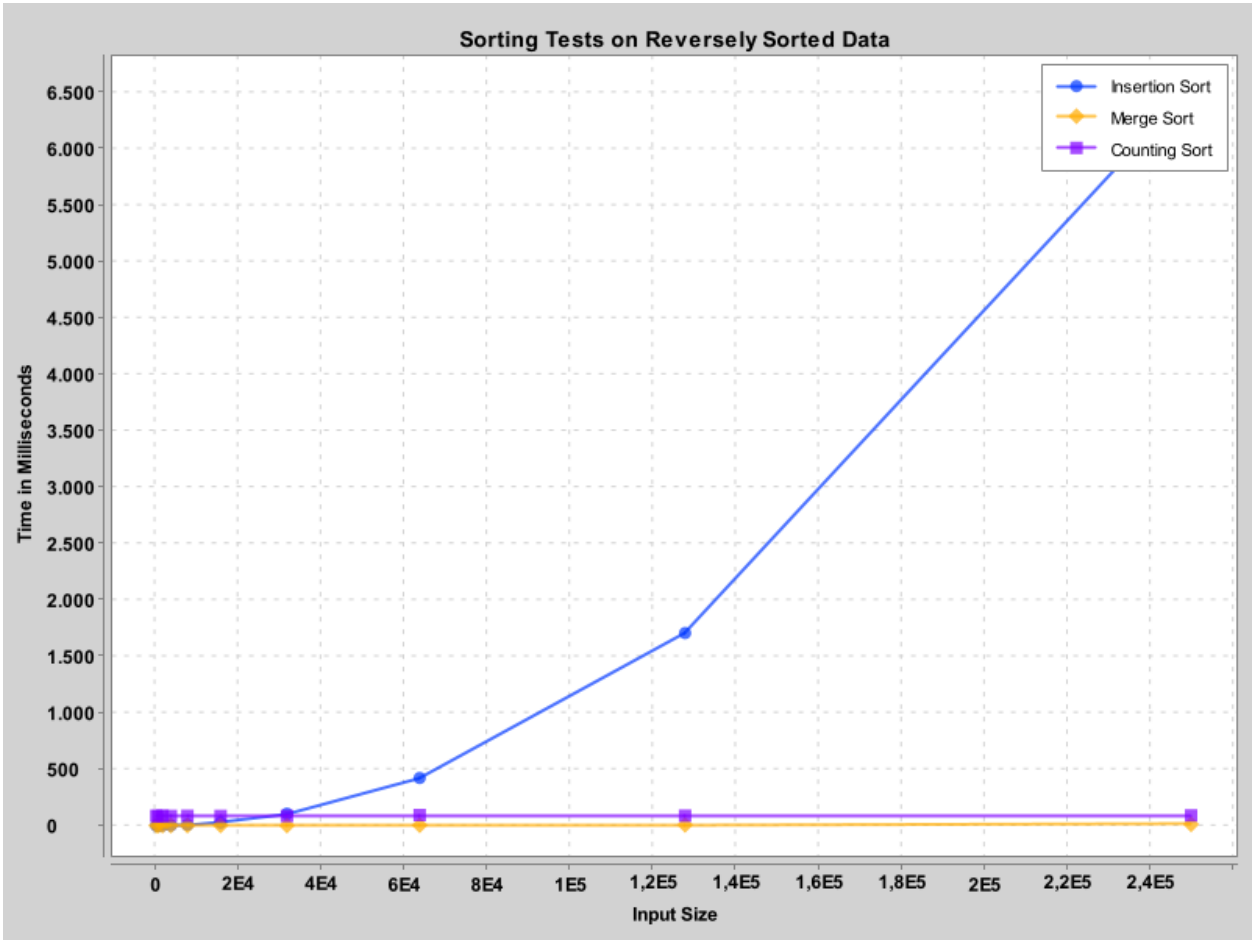

Figure 3: Time Chart of Reversely Sorted Data

With reversely sorted data, Insertion Sort's average processing time tends to go up as dataset sizes increase. However, Merge Sort stays efficient, showing low average processing times similar to those seen with random and sorted data. Counting Sort also remains stable, with processing times not changing much as the dataset grows.

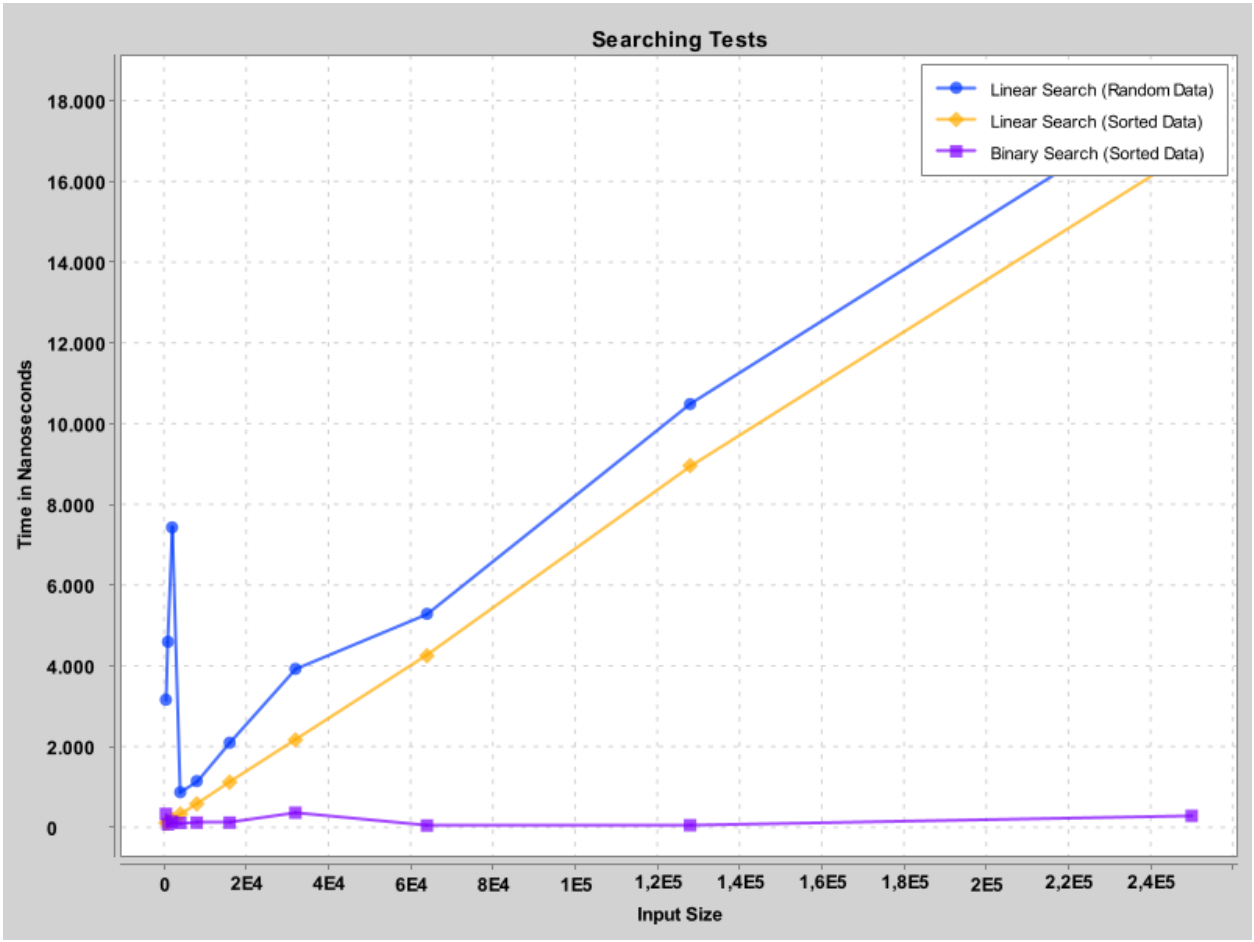

Figure 4: Time Chart of Searching Algorithms

When performing linear search on random data, the time required naturally increases as the dataset expands. This is because linear search methodically examines each element, and as the dataset size grows, the process takes longer. However, when dealing with sorted data, linear search performs more efficiently. This is because it can terminate early upon finding the target element, making the search faster compared to random data. Even with larger datasets, the time taken for linear search on sorted data remains lower than that for random data.

In contrast, binary search on sorted data offers a significantly faster and more consistent search process compared to linear search. Binary search swiftly narrows down the search space, resulting in quicker searches, particularly for larger datasets. Moreover, the time taken for binary search remains relatively stable even as the dataset size increases, highlighting its efficiency when dealing with sorted data.

In conclusion, although there are some small differences, the overall trend of the graphs and data matched what was expected based on the theoretical information provided in the table. While there wasn't much variation in processing times for small random datasets, using Merge Sort becomes a more logical choice as the input size increases. After that, Counting Sort can also be considered, but due to the $O(n^2)$ complexity of Insertion Sort, it's not ideal for larger random datasets as it results in much longer processing times. For sorted datasets, Insertion Sort operates faster than Merge Sort and Counting Sort in terms of processing time since it's comparison-based. However, it's not logical to use Counting Sort for sorted data. On the other hand, for reversely sorted data, using Merge Sort remains the best option, but Insertion Sort shows its worst-case scenario due to the reversed order. Insertion Sort still operates with $O(n^2)$ complexity, but this time, with the maximum level of comparison, resulting in considerably longer processing times. Counting Sort, as before, operates consistently with similar processing times as seen in previous examples. For the searching part, it's obvious that binary search is more effective than the linear search algorithm.

References

- $https://en.wikipedia.org/wiki/Insertion_sort$
- $https://en.wikipedia.org/wiki/Merge_sort$
- $https://en.wikipedia.org/wiki/Counting_sort$
- $https://www.geeksforgeeks.org/merge-sort/$