
Hacettepe University

Computer Engineering Department


BBM204 Software Practicum II - 2024 Spring

# Programming Assignment 1

March 22, 2024


*Student name:*
Yusuf Emir CÖMERT

*Student Number:*
b2220765023

# 1 Problem Definition

The goal of this assignment is to analyze and compare the efficiency of different sorting and searching algorithms for various input data types and sizes.

# 2 Solution Implementation

In this section, I'll provide a detailed explanation of the implementation of the searching and sorting algorithms within the Searches and Sorts classes.

In this section, there are going to be 3 sort algorithms and 2 search algorithms

## 2.1 Insertion Sort

Implementation of Insertion Sort Algorithm:

```
1
2  public static void InsertionSort(int[] array){
3      int n = array.length;
4      for (int j = 1; j < n; j++) {
5          int key = array[j];
6          int i = j - 1;
7          while (i >= 0 && array[i] > key) {
8              array[i + 1] = array[i];
9              i = i - 1;
10         }
11         array[i + 1] = key;
12     }
13 }
```

Best Case Complexity: O(n)
Average Case Complexity: $O(n^2)$
Worst Case Complexity: $O(n^2)$
Auxiliary Space Complexity: O(1)

Insertion sort is simple and efficient for small input sizes or nearly sorted arrays. However, its performance degrades significantly as the input size increases or when the array is reverse sorted.

## 2.2 Merge Sort

Implementation of Merge Sort Algorithm:

```
14
15 public static void MergeSort(int[] array) {
16     if (array.length <= 1) {
17         return;
18     }
19     int middle = array.length / 2;
20     int[] left = new int[middle];
21     int[] right = new int[array.length - middle];
22     for (int i = 0; i < middle; i++) {
23         left[i] = array[i];
24     }
25     for (int i = middle; i < array.length; i++) {
26         right[i - middle] = array[i];
27     }
28     MergeSort(left);
29     MergeSort(right);
30     merge(array, left, right);
31 }
32
33 public static void merge(int[] array, int[] left, int[] right) {
34     int i = 0, j = 0, k = 0;
35     while (i < left.length && j < right.length) {
36         if (left[i] <= right[j]) {
37             array[k] = left[i];
38             i++;
39         } else {
40             array[k] = right[j];
41             j++;
42         }
43         k++;
44     }
45     while (i < left.length) {
46         array[k] = left[i];
47         i++;
48         k++;
49     }
50     while (j < right.length) {
51         array[k] = right[j];
52         j++;
53         k++;
54     }
55 }
```

Best Case Complexity: $O(n \log n)$
Average Case Complexity: $O(n \log n)$
Worst Case Complexity: $O(n \log n)$
Auxiliary Space Complexity: 0(n)

Merge sort guarantees a consistent $O(n \log n)$ time complexity regardless of the input distribution. It's a stable, comparison-based sorting algorithm suitable for large datasets.

### 2.3 Counting Sort

Implementation of Counting Sort Algorithm:

```
56
57 public static int[] CountingSort(int[] array, int Max){
58     int[] countArray = new int[Max + 1];
59     for (int i = 0; i < array.length; i++) {
60         countArray[array[i]]++;
61     }
62     for (int i = 1; i <= Max; i++) {
63         countArray[i] += countArray[i - 1];
64     }
65     int[] outputArray = new int[array.length];
66
67     for (int i = array.length - 1; i >= 0; i--) {
68         outputArray[countArray[array[i]] - 1] = array[i];
69         countArray[array[i]]--;
70     }
71     return outputArray;
72 }
```

Best Case Complexity: $O(n + k)$
Average Case Complexity: $O(n + k)$
Worst Case Complexity: $O(n + k)$
Auxiliary Space Complexity: $O(n + k)$

Counting sort is efficient for sorting integers within a specific range (when k is not significantly larger than n). It doesn't rely on comparisons between elements, making it faster than comparison-based sorting algorithms for certain scenarios.

### 2.4 Linear Search

Implementation of Linear Search Algorithm:

```
73
74  public static int LinearSearch(int[] array, int value){
75      for(int i = 0; i < array.length; ++i){
76          if (array[i] == value)
77              return i;
78      }
79      return -1;
80  }
```

Best Case Complexity: O(1)
Average Case Complexity: O(n)
Worst Case Complexity: O(n)
Auxiliary Space Complexity: O(1)

Linear search is straightforward but inefficient for large datasets, as it scans through each element sequentially. It's suitable for unsorted or small-sized arrays. Usage of this algorithm is, it is easy to implement.

### 2.5 Binary Search

Implementation of Binary Search Algorithm:

```
81
82  public static int BinarySearch(int[] array, int value){
83
84      int low = 0;
85      int high = array.length -1;
86      while ((high - low) > 1){
87          int mid = (high + low) / 2;
88          if (array[mid] < value)
89              low = mid + 1;
90          else high = mid;
91
92      }
93
94      if (array[low] == value)
95          return low;
96      else if (array[high] == value)
97          return high;
98
99      return -1;
100
101 }
```

Best Case Complexity: O(1)
Average Case Complexity: O($\log n$)
Worst Case Complexity: O($\log n$)
Auxiliary Space Complexity: O($\log n$)

Binary search efficiently locates elements in sorted arrays by repeatedly dividing the search interval in half. It's suitable for large datasets and ensures a logarithmic time complexity.

# 3 Results, Analysis, Discussion

***Overall Analysis:***
*Counting Sort vs. Comparison-Based Sorts:*

-Counting sort has a linear time complexity but requires additional space proportional to the range of input values. It's efficient for certain scenarios but may not be suitable for large ranges or non-integer inputs.

-Comparison-based sorts like insertion sort and merge sort have O($n^2$) and O($n \log n$) time complexities, respectively, but they don't have specific constraints on input values.

*Efficiency vs. Space Trade-offs:*

-Algorithms like insertion sort and binary search are space-efficient (O(1) auxiliary space) but may have higher time complexities for certain inputs.

-Algorithms like merge sort and counting sort may require additional space (O(n) or O(n + k) auxiliary space) but offer better time complexities and scalability for larger datasets.

*Application-specific Considerations:*

-The choice of algorithm depends on various factors such as input size, input distribution, memory constraints, and expected performance requirements.

-Understanding the trade-offs between time complexity, space complexity, and algorithmic efficiency is crucial for selecting the most appropriate algorithm for a given application.

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 | 0.3 | 0.2 | 0.2 | 1.0 | 3.6 | 17.0 | 66.9 | 268.5 | 1065.6 | 4223.2 |
| Merge sort | 0.0 | 0.0 | 0.7 | 0.8 | 1.5 | 1.9 | 4.0 | 8.6 | 19.9 | 32.3 |
| Counting sort | 271.7 | 229.3 | 96.7 | 66.0 | 63.6 | 63.6 | 65.2 | 68.9 | 64.9 | 67.4 |
| | **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 | 0.4 | 1.0 |
| Merge sort | 0.0 | 0.1 | 0.1 | 0.0 | 0.6 | 0.9 | 2.0 | 3.7 | 6.7 | 13.1 |
| Counting sort | 68.7 | 61.9 | 62.9 | 63.8 | 68.0 | 62.8 | 62.3 | 63.8 | 63.3 | 64.0 |
| | **Reversely Sorted Input Data Timing Results in ms** | | | | | | | | | |
| Insertion sort | 0.0 | 0.3 | 0.5 | 2.2 | 9.0 | 41.1 | 142.5 | 581.0 | 2276.3 | 8738.3 |
| Merge sort | 0.0 | 0.0 | 0.2 | 0.2 | 0.3 | 0.8 | 1.4 | 3.2 | 6.4 | 13.1 |
| Counting sort | 61.2 | 62.2 | 62.5 | 61.1 | 59.7 | 59.7 | 60.8 | 67.5 | 65.3 | 66.3 |

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) | 3636.0 | 5699.0 | 9257.1 | 11524.8 | 12034.5 | 12461.5 | 12693.2 | 12740.2 | 15433.8 | 25139.3 |
| Linear search (sorted data) | 80.3 | 107.8 | 170.6 | 296.5 | 522.2 | 877.8 | 1709.8 | 3609.0 | 7477.1 | 14442.0 |
| Binary search (sorted data) | 255.2 | 134.3 | 149.6 | 121.3 | 133.4 | 144.5 | 161.5 | 211.3 | 365.7 | 568.1 |

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

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 | $O(1)$ |

Runtime of sort algorithms with RANDOM data, figure: Fig. 1.

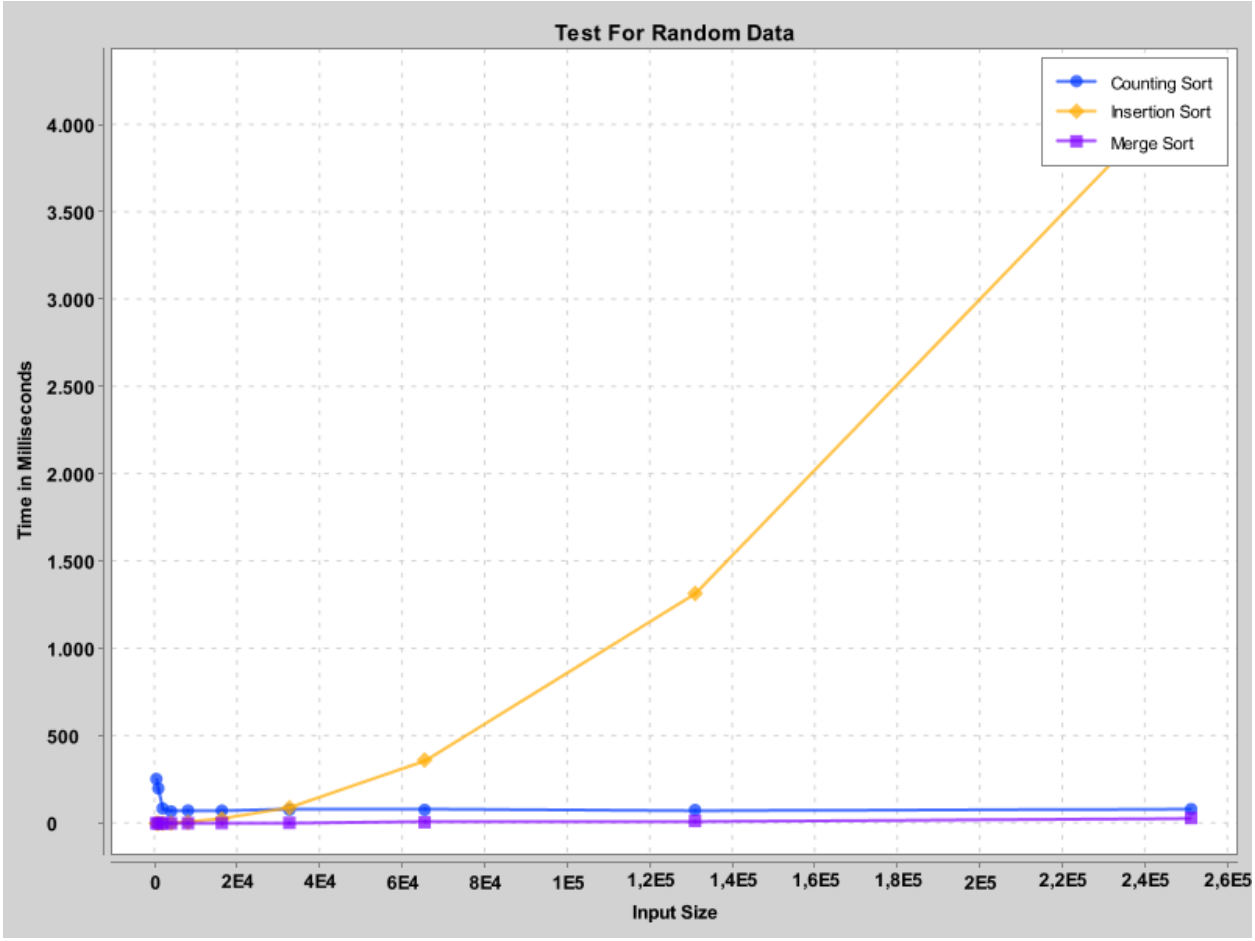


Figure 1: Plot of the Sort Algorithms with RANDOM Data.

**Counting Sort:** Exhibits decreasing running times as the input size increases, reflecting its linear time complexity (O(n+k)) and efficient sorting of random data.

**Insertion Sort:** Shows significantly increasing running times with larger input sizes, highlighting its inefficiency for large datasets. Insertion Sort's time complexity (O($n^2$)) leads to longer execution times as the input size grows.

**Merge Sort:** Demonstrates stable and low running times across different input sizes. Merge Sort's divide-and-conquer strategy ensures consistent O($nlogn$) performance, making it suitable for random data.

Runtime of sort algorithms with SORTED data, figure: Fig. 4.

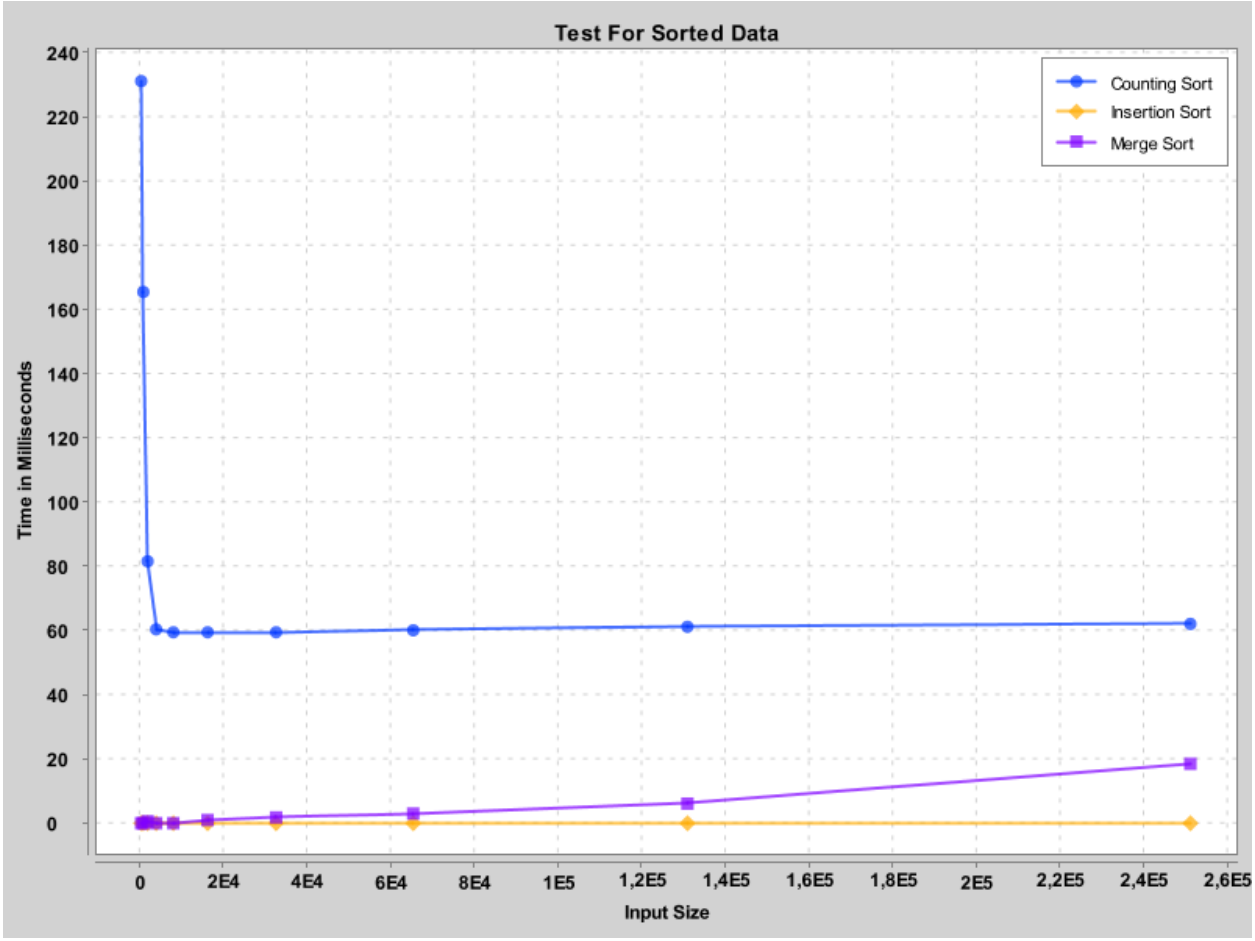


Figure 2: Plot of the Sort Algorithms with SORTED Data.

This graph may be wrong. I could not fixed it.

**Counting Sort:** Exhibits stable and consistent running times, indicating linear time complexity (O(n+k)) regardless of the input size. Sorted input maintains a consistent range of values, leading to similar execution times.

**Insertion Sort:** Demonstrates negligible running times due to its efficiency with sorted input. Minimal comparisons and shifts are required, resulting in nearly constant execution times.

**Merge Sort:** Shows stable and low running times similar to those with random input. Its time complexity (O($nlogn$)) ensures efficient performance irrespective of the input distribution.

Runtime of sort algorithms with REVERSELY SORTED data, figure: Fig. 4.

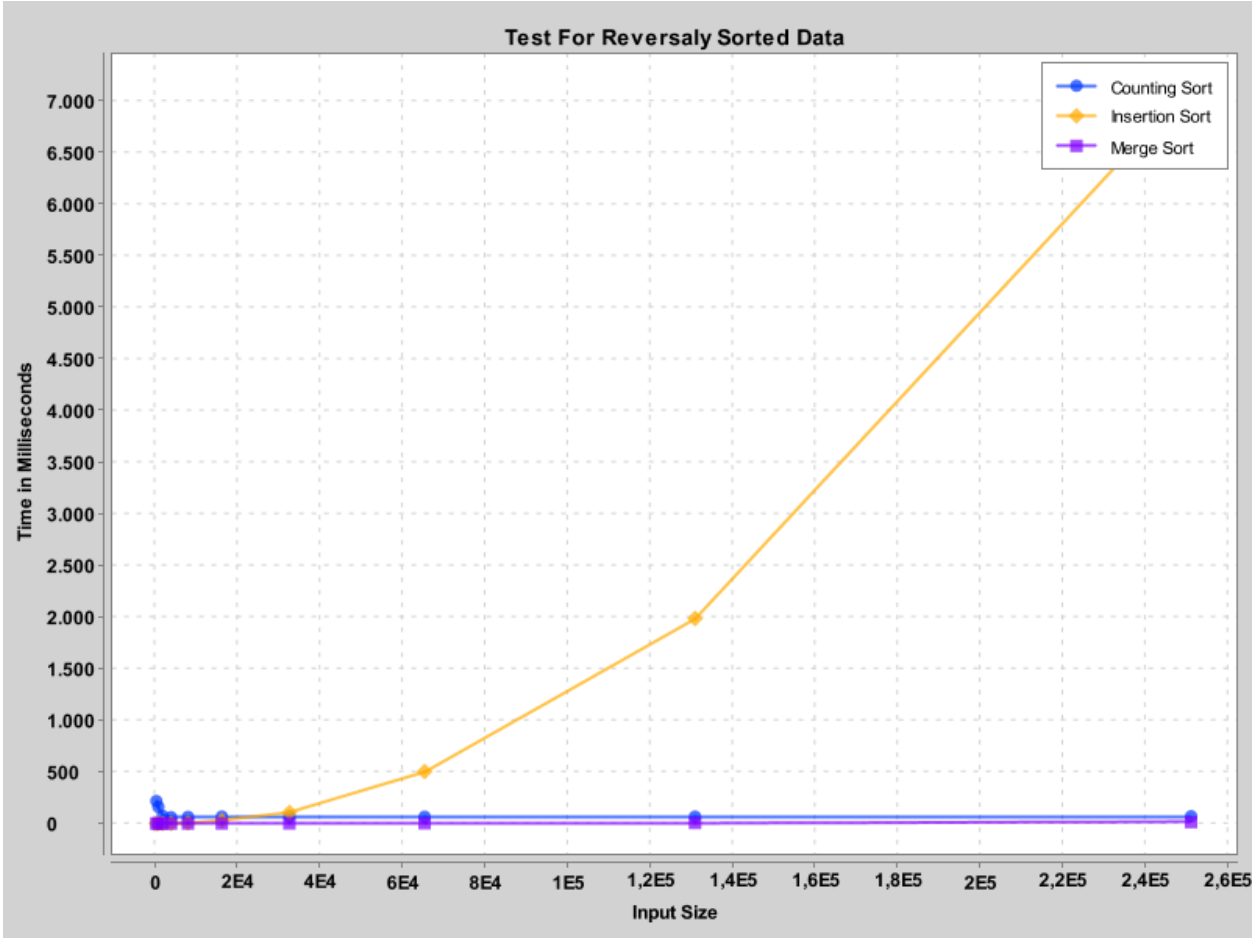


Figure 3: Plot of the Sort Algorithms with REVERSELY SORTED Data.

**Counting Sort:** Maintains stable running times, similar to those with sorted input. Reversely sorted input still leads to linear time complexity for Counting Sort, resulting in consistent execution times.

**Insertion Sort:** Demonstrates significantly increasing running times with larger input sizes due to the worst-case scenario for Insertion Sort. Reversely sorted input requires maximum comparisons and shifts, leading to escalating execution times.

**Merge Sort:** Shows stable running times, slightly higher than those with sorted or random input. Merge Sort's efficiency remains intact even with reversely sorted input, maintaining O($nlogn$) time complexity.

Runtime of search algorithms, figure: Fig. 4.

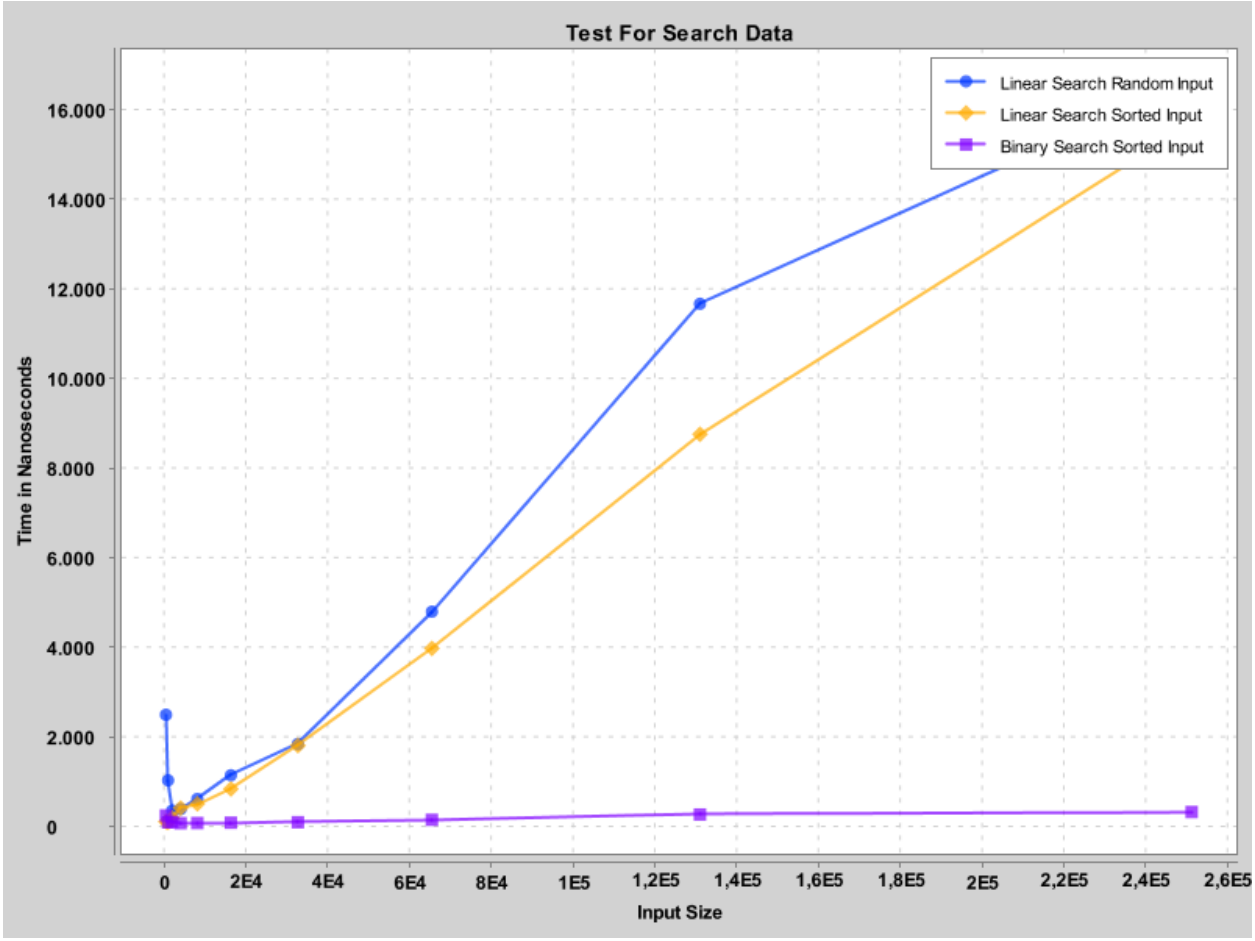


Figure 4: Plot of the Search Algorithms.

*NOTE:* This graph is changing every time you run the code.

**Linear Search:** Consistently increasing running times with larger input sizes, reflecting its linear time complexity (O(n)). As the input size grows, the search operation requires more iterations, leading to longer execution times.

**Binary Search:** Exhibits stable and low running times across different input sizes, demonstrating its logarithmic time complexity (O($logn$)). Binary Search efficiently locates elements in sorted arrays with minimal iterations.

# 4 Notes

Overall, the analysis indicates that the algorithms perform differently depending on the input characteristics. Understanding these behaviors helps in selecting the appropriate algorithm for specific use cases to achieve optimal performance.

# References


- https://www.youtube.com/watch?v=xJNOwfL7WNg
- https://stackoverflow.com/questions/4023137/generating-a-random-index-for-an-array
- https://www.geeksforgeeks.org/counting-sort/
- https://www.geeksforgeeks.org/merge-sort/
- https://stackoverflow.com/questions/5204051/how-to-calculate-the-running-time-of-my-program