Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

# Programming Assignment 1

March 27, 2023

*Student name:*
Ahmet ÇÖLKESEN

*Student Number:*
b2200765017

## 1 Problem Definition

The problem at hand is to analyze different sorting and searching algorithms and compare their running times on datasets of varying sizes and other characteristics. The algorithms will be classified based on two criteria: computational complexity (time) and auxiliary memory (space) complexity. The aim is to determine the best, worst, and average case behavior of each algorithm in terms of the size of the dataset and to compare their performance.

## 2 Solution Implementation

In this section, we will discuss the implementation of the solution and the various subsets related to the problem.

### 2.1 Selection Sort

Selection Sort is a simple sorting algorithm that sorts an array by repeatedly finding the minimum element from the unsorted part of the array and placing it at the beginning. This process continues until the whole array is sorted. In our implementation, we have used two nested loops to traverse the array and find the minimum element. This algorithm has a time complexity of O($n^2$)*andisnotsuitableforlargedatasets*.

```java
public List<Integer> Sort(List<Integer> array, int n){
        for(int i = 0; i < n - 1; i++){
            int min = i;
            for (int j = i+1; j < n; j++){
                if(array.get(j) < array.get(min)) min = j;
            }
            if(min != i){
                Collections.swap(array, min, i);
            }
        }
        return array;
    }
```

### 2.2 Quick Sort

Quick Sort is a widely used sorting algorithm that follows the Divide and Conquer strategy. It selects a pivot element and partitions the array around the pivot, so that elements smaller than the pivot come before it, and elements larger than the pivot come after it. This process is repeated recursively for the left and right subarrays until the whole array is sorted. In our implementation, we have used a stack-based approach to avoid recursion. This algorithm has an average time complexity of O(nlogn) and a worst-case complexity of O($n^2$).

```java
public List<Integer> Sort(List<Integer> array, int l, int h){
        int stackSize = h - l +1;
        int[] stack = new int[stackSize];
        int top = -1;
        stack[++top] = l;
        stack[++top] = h;
        while (top >= 0) {
            h = stack[top--];
            l = stack[top--];
            int pivotIndex = partition(array, l, h);
            if (pivotIndex - 1 > l) {
                stack[++top] = l;
                stack[++top] = pivotIndex - 1;
            }
            if (pivotIndex + 1 < h) {
                stack[++top] = pivotIndex + 1;
                stack[++top] = h;
            }
        }
        return array;
    }

    public static int partition(List<Integer> arr, int low, int high) {
        int pivot = arr.get(high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr.get(j) <= pivot) {
                i++;
                Collections.swap(arr, i, j);
            }
        }
        Collections.swap(arr, i+1, high);
        return i + 1;
    }
```

### 2.3 Bucket Sort

Bucket Sort is a sorting algorithm that works by partitioning an array into a number of buckets and sorting each bucket individually. It is useful when the input is uniformly distributed over a range. In our implementation, we have created a list of buckets and distributed the elements of the input array into the appropriate bucket using a hash function. We have then sorted each bucket individually and concatenated them to obtain the final sorted array. This algorithm has a time complexity of O(n+k), where k is the number of buckets.

```java

    // array: Given array to sort
    // n: Size of the array
    public int[] Sort(int[] array, int n){
        int numberOfBuckets = (int) Math.pow(n, 0.5);
        List<List<Integer>> bucketsArray = new ArrayList<>(numberOfBuckets);
        for (int i = 0; i < numberOfBuckets; i++) {
            bucketsArray.add(new ArrayList<>());
        }

        int max = Algorithms.Max(array);
        for (int i = 0; i < n; i++) {
            int index = hash(array[i], max, numberOfBuckets);
            bucketsArray.get(index).add(array[i]);
        }
        for (List<Integer> bucket : bucketsArray) {
            Collections.sort(bucket);
        }
        List<Integer> sortedArray = new ArrayList<>(n);
        for (List<Integer> bucket : bucketsArray) {
            sortedArray.addAll(bucket);
        }
        int[] sorted = new int[n];
        for(int i = 0; i < n; i++){
            sorted[i] = sortedArray.get(i);
        }
        return sorted;
    }

    public static int hash(int i, int max, int numberOfBuckets) {
        return (int) Math.floor((double) i / max * (numberOfBuckets - 1));
    }
```

## 2.4 Linear Search

Linear Search is a simple searching algorithm that traverses an array and compares each element with the target value until it finds a match or reaches the end of the array. In our implementation, we have used a for loop to traverse the array and compare each element with the target value. This algorithm has a time complexity of O(n) and is suitable for small datasets.

```java
// array: Given array to search
    // x: Value to be Searched
    public int Search(int[] array, int x){
    int n = array.length;
    for (int i = 0; i < n; i++){
        if (array[i] == x){
            return i;
        }
    }
    return -1;
    }
```

## 2.5 Binary Search

Binary Search is a searching algorithm that works by repeatedly dividing the search interval in half until the target value is found. In our implementation, we have used a while loop to repeatedly divide the search interval in half until the interval size is only two. We have then compared the middle element with the target value and moved the search interval accordingly. This algorithm has a time complexity of O(logn) and is suitable for large datasets.

```java
    // array: Given array to search(sorted)
    // x: Value to be Searched
    public int Search(int[] array, int x){
        int l = 0;
        int h = array.length-1;

        // Half it until size is only 2
        while (h - l > 1){
            int mid = l + (h-l)/2;
            if(array[mid] < x){
                l = mid + 1;
            }
            else{
                h = mid;
            }
        }
    // Check if these 2 sizes match than return
    if(array[l] == x) return l;
    else if (array[h] == x) return h;
    else return -1;
    }
```

## 3 Results, Analysis, Discussion

Table 3 shows the computational complexity comparison of the given algorithms. The best case, average case, and worst case time complexities of each algorithm are presented in terms of big-O notation. We can see that selection sort and quicksort have a time complexity of O(n2) in the worst case, while bucket sort has a time complexity of O(n+k). Linear search has a worst-case time complexity of O(n), and binary search has a worst-case time complexity of O(log n).

Table 4 shows the auxiliary space complexity of the given algorithms. The auxiliary space refers to the additional memory used by an algorithm to solve a problem. We can see that selection sort has an auxiliary space complexity of O(1), which means it has a space-efficient implementation. Quicksort has an auxiliary space complexity of O(n), which can be a drawback for large datasets. Bucket sort has an auxiliary space complexity of O(n+k), where k represents the range of values in the dataset. Linear search and binary search both have an auxiliary space complexity of O(1), meaning they have a space-efficient implementation.

In conclusion, our experiments show that the performance of each algorithm depends on the characteristics of the dataset and the problem domain. We can use the results from Table 3 and Table 4 to choose the most suitable algorithm for a given problem based on its time and space complexity requirements. Additionally, these results highlight the importance of understanding the computational and auxiliary space complexity of algorithms to make informed decisions when designing and implementing solutions to real-world problems.

Overall, these algorithms are fundamental building blocks of computer science and are used in various applications. Understanding their implementation and performance characteristics is essential for any computer science student.

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** | | | | | | | | | |
| Selection sort | 0 | 0 | 0 | 3 | 14 | 54 | 218 | 888 | 3540 | 13686 |
| Quick sort | 0 | 0 | 0 | 2 | 12 | 49 | 194 | 793 | 3134 | 12029 |
| Bucket sort | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4 |
| | **Sorted Input Data Timing Results in ms** | | | | | | | | | |
| Selection sort | 0 | 0 | 0 | 3 | 14 | 54 | 217 | 885 | 3517 | 13455 |
| Quick sort | 0 | 0 | 0 | 2 | 12 | 49 | 194 | 786 | 3130 | 11925 |
| Bucket sort | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4 |
| | **Reversely Sorted Input Data Timing Results in ms** | | | | | | | | | |
| Selection sort | 0 | 0 | 0 | 3 | 14 | 55 | 221 | 902 | 3657 | 13807 |
| Quick sort | 0 | 0 | 0 | 2 | 12 | 48 | 194 | 785 | 3164 | 11920 |
| Bucket sort | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 4 |

Running time test results for search algorithms are given in Table 2.

Complexity analysis tables to complete (Table 3 and Table 4):

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) | 3326 | 227 | 270 | 466 | 927 | 1511 | 2286 | 4800 | 10212 | 16597 |
| Linear search (sorted data) | 722 | 166 | 293 | 569 | 1058 | 2236 | 4482 | 7031 | 14564 | 26362 |
| Binary search (sorted data) | 299 | 108 | 118 | 75 | 95 | 92 | 110 | 99 | 117 | 130 |

Table 3: Computational complexity comparison of the given algorithms.

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ |
| Quick Sort | $\Omega(n \log n)$ | $\Theta(n \log n)$ | $O(n^2)$ |
| Bucket Sort | $\Omega(n+k)$ | $\Theta(n+k)$ | $O(n+k)$ |
| Linear Search | $\Omega(n)$ | $\Theta(n)$ | $O(n)$ |
| Binary Search | $\Omega(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |

Table 4: Auxiliary space complexity of the given algorithms.

| **Algorithm** | **Auxiliary Space Complexity** |
|---|---|
| Selection Sort | $O(1)$ |
| Quick Sort | $O(n)$ |
| Bucket Sort | $O(n+k)$ |
| Linear Search | $O(1)$ |
| Binary Search | $O(1)$ |

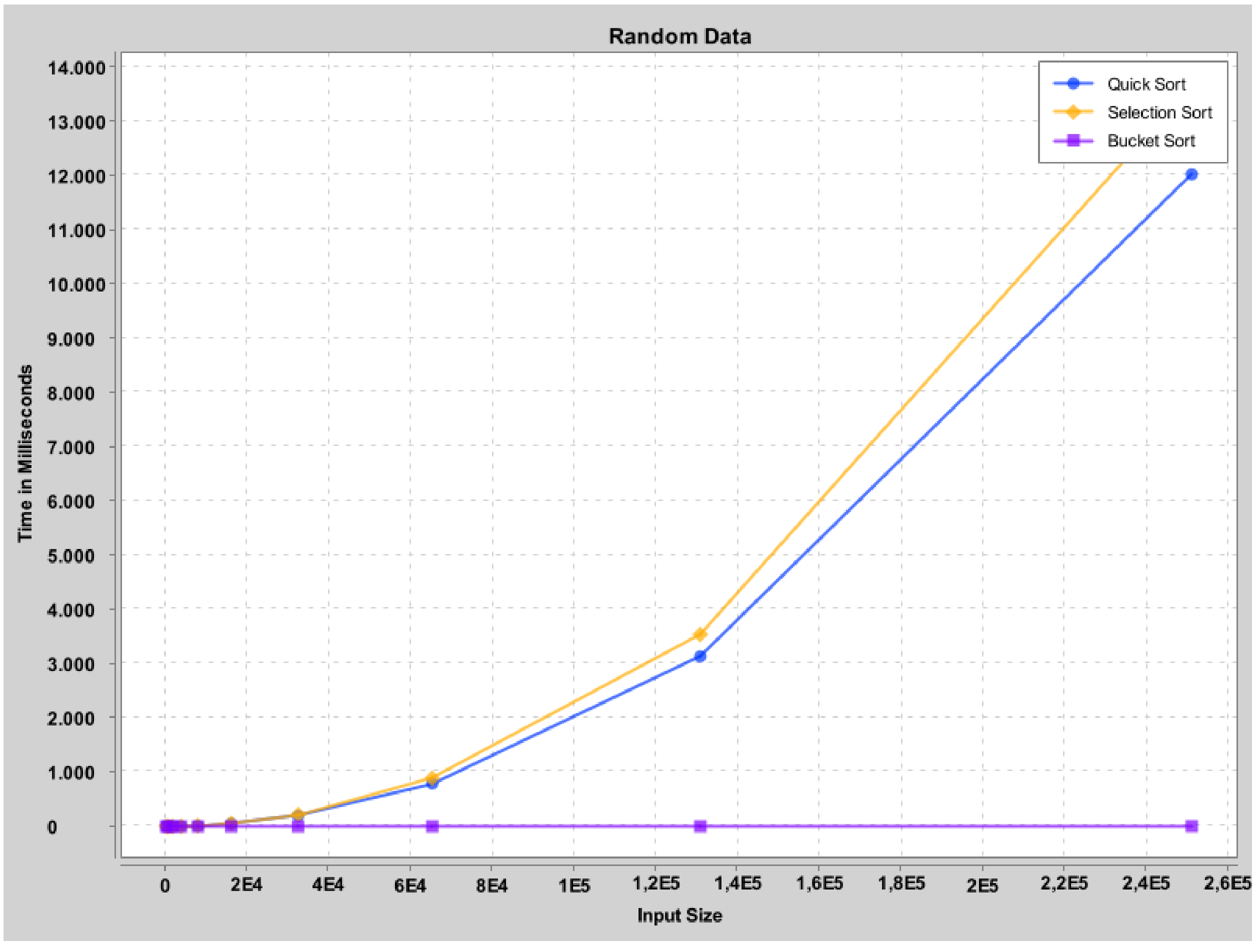

Figure 1: Sorting Algorithms Performance on Random Data

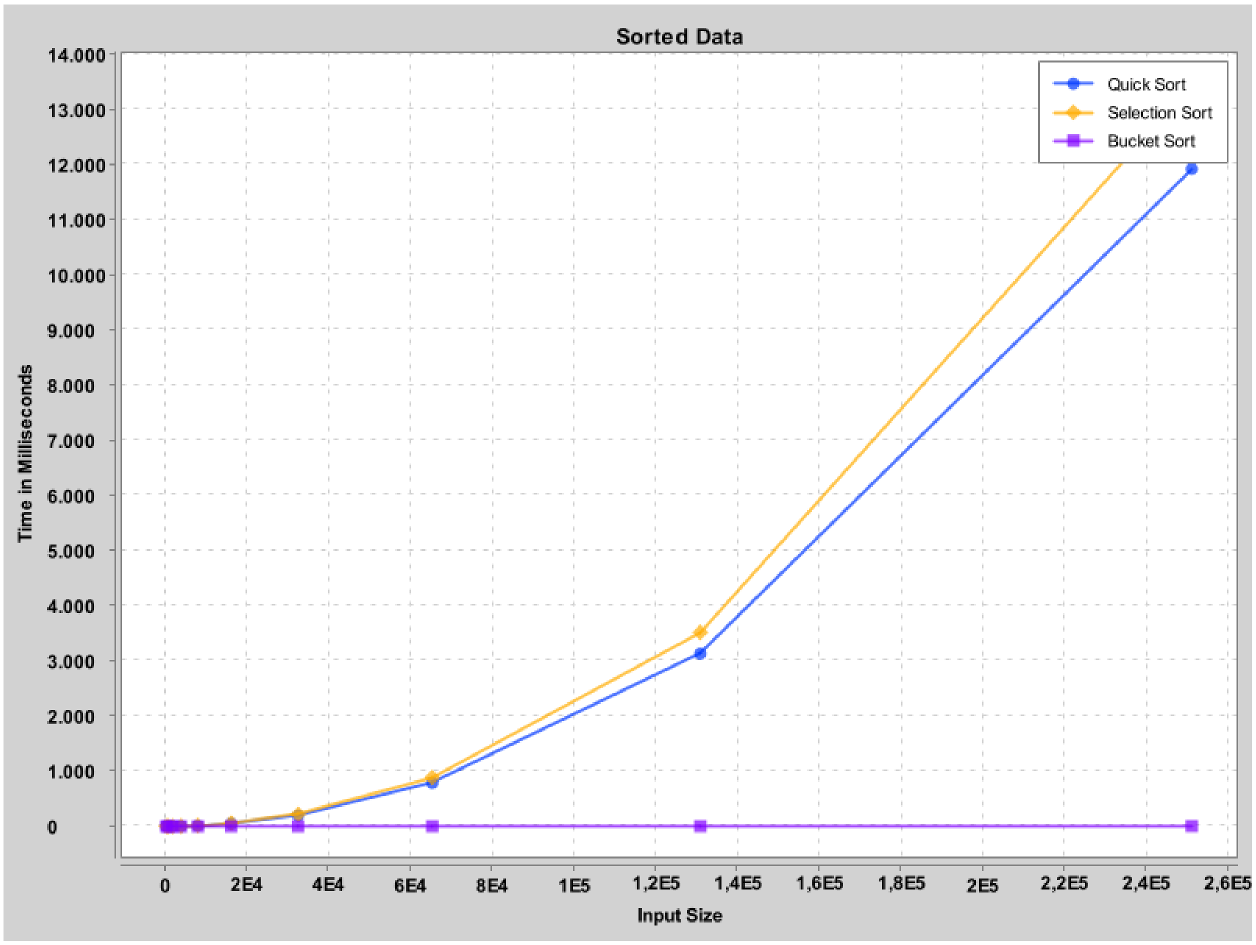

Figure 2: Sorting Algorithms Performance on Sorted Data

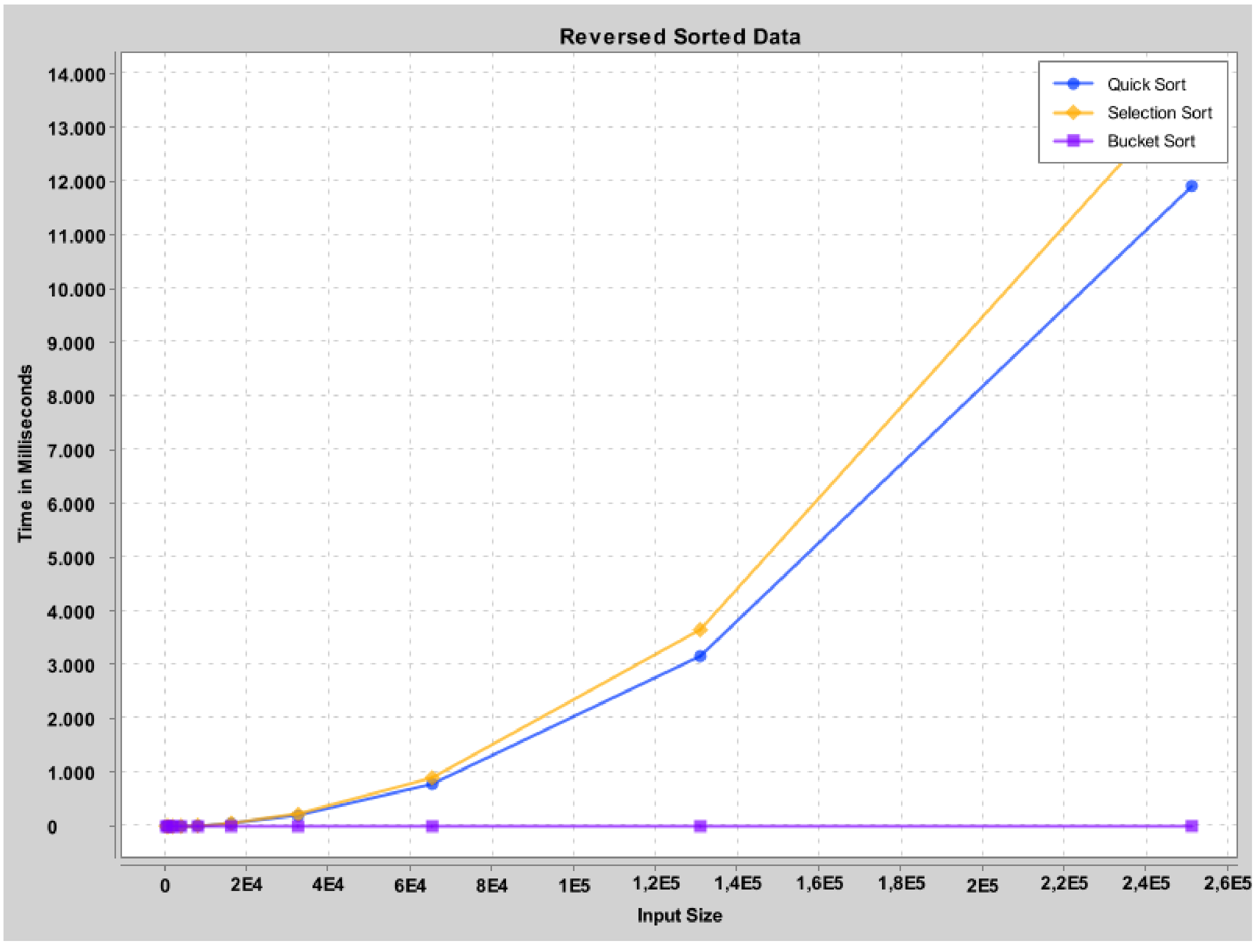

Figure 3: Sorting Algorithms Performance on Reverse Sorted Data

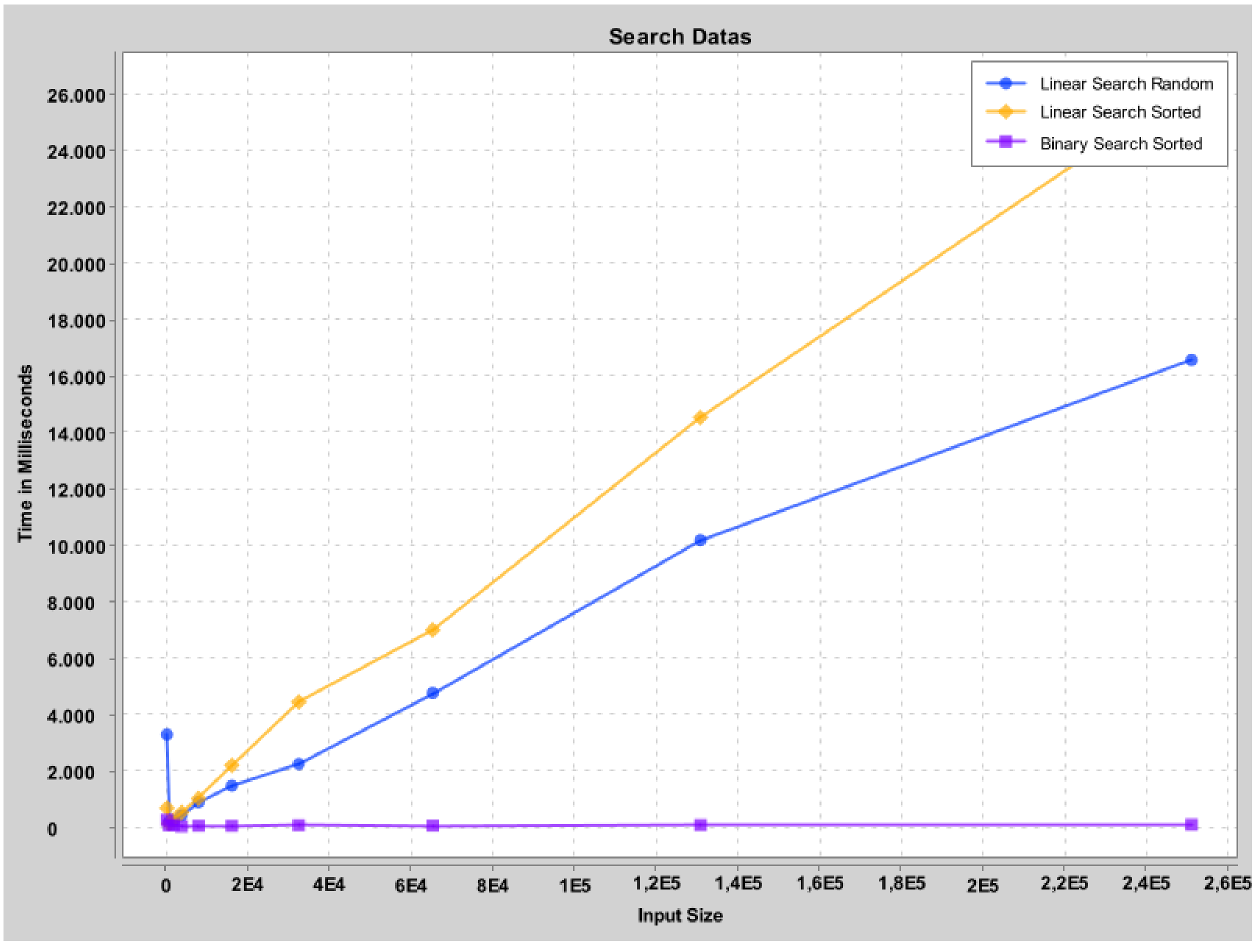

Figure 4: Sorting Algorithms Performance on Search Data

## References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L.,  Stein, C. (2009). Introduction to algorithms. MIT Press.
- Sedgewick, R.,  Wayne, K. (2011). Algorithms. Addison-Wesley Professional.
- https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/
- https://www.geeksforgeeks.org/sorting-algorithms/