Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Mert ERGÜN

Student Number:
b2220356062

1 Problem Definition

In a realm where time is of the essence and memory is a treasure, our intrepid data wranglers face a conundrum of epic proportions. They embark on a quest to tame the wild beasts known as sorting and searching algorithms. With a trove of data at their fingertips, they must harness the power of Java to bend these algorithms to their will, comparing their voracious appetites for time and space. Will our heroes triumph, optimizing the efficiency of their digital domain, or will they be ensnared in a labyrinth of computational complexity? Join them as they unravel the mysteries of algorithmic performance, where each step closer to optimization is a victory against the clockwork chaos of the digital cosmos.

2 Solution Implementation

In the echoing chambers of the digital fortress, our valiant heroes, armed with the mystical scrolls of Java, perform ancient rituals such as Insertion sort, Merge sort and Counting sort. Each spell, meticulously written in the sacred language of code, is a testament to their search for order in the chaos of data.

In the dark cellar they had entered, they held three spells they could use to sort the scattered piles of data. Each one was more powerful than the other. Some were less taxing on the mage, but took longer to materialize, while others were much, much faster, but drained the mage's strength in the process. Our heroes had to make very wise choices. They rolled up their sleeves and opened their ancient minds to the wisdom of the ancients.

2.1 Insertion Sort

First, they channeled the Insertion Sort, a meticulous ritual where they would pick one piece of data at a time, comparing it to those already sorted, and placing it in its rightful position. Though not the swiftest, this spell's simplicity was its beauty, ideal for sorting smaller, less chaotic data realms.

```java
public class InsertionSort {
    public static int[] sort(int[] array) {
        for (int i = 1; i < array.length; i++) {
            int key = array[i];
            int j = i - 1;
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = key;
        }
        return array;
    }
}
```

2.2 Merge Sort

Next, they summoned the mighty Merge Sort, invoking a powerful divide-and-conquer strategy. They split the data pile into smaller fragments, sorted each fragment, and then magically merged them into a harmonious order. This spell, while more complex, dazzled with its efficiency, especially in larger, more tangled data realms.

```
public class MergeSort {
    public static int[] sort(int[] array) {
        if (array.length > 1) {
            int mid = array.length / 2;
            int[] left = new int[mid];
            int[] right = new int[array.length - mid];

            for (int i = 0; i < mid; i++) {
                left[i] = array[i];
            }
            for (int i = mid; i < array.length; i++) {
                right[i - mid] = array[i];
            }

            sort(left);
            sort(right);

            int i = 0;
            int j = 0;
            int k = 0;

            while (i < left.length && j < right.length) {
                if (left[i] < j) {
                    array[k] = left[i];
                    i++;
                } else {
                    array[k] = right[j];
                    j++;
                }
                k++;
            }

            while (i < left.length) {
                array[k] = left[i];
                i++;
                k++;
            }

            while (j < right.length) {
                array[k] = right[j];
                j++;
```

```
                    k++;
                }
            }
            return array;
        }
}
```

2.3 Counting Sort

Lastly, the Counting Sort was called upon, a spell best suited for numerical data with a known range. Instead of comparisons, this enchantment involved tallying the occurrences of each number and then reconstructing the array. A swift and potent spell, but its power was confined to specific situations where the data's nature was well-understood.

```
public class CountingSort {
    public static int[] sort(int[] array) {
        int k = 0;

        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] > k) {
                k = array[i];
            }
        }

        int[] count = new int[k + 1];
        int[] output = new int[array.length];

        for (int i = 0; i < size; i++) {
            count[array[i]]++;
        }

        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }

        for (int i = size - 1; i >= 0; i--) {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }

        return output;
    }
}
```

Through these ancient rituals, our heroes sought to bring order to the chaos, their choices reflecting a deep understanding of their arsenal's strengths and limitations.

But it wasn't enough for them to organize the masses of data that had emerged from the horrible void in front of them! If they wanted to advance in these terrifying dungeons, they had to find the ancient sacred knowledge they were looking for in the masses of data they would encounter.

Our heroes must make their way through the pitch-black corridors of the dreaded dungeons, where they must face the twin guardians of ancient knowledge: Linear Search and Binary Search!

2.4 Linear Search

In the dimly lit corridors of the dungeon, our data knights first summon the spirit of Linear Search. Like a tireless sentinel, it marches forth, scrutinizing each element in turn, undeterred by the murk and mire of unsorted data. It's a testament to perseverance, for it leaves no stone unturned, no clue unchecked until it finds its quarry or exhausts the trail.

Linear Search follows no order. It starts at the beginning of the data, visiting all the data without stopping until it finds what it is looking for. It doesn't tire it out, but if you are waiting in the hope that it will find something, the waiting can take centuries in crowded rooms.

```
92  public class LinearSearch {
93      public static int search(int[] arr, int target) {
94          for (int i = 0; i < arr.length; i++) {
95              if (arr[i] == target) {
96                  return i;
97              }
98          }
99          return -1;
100     }
101 }
```

2.5 Binary Search

But the true test of their mettle comes with the invocation of Binary Search. This potent spell slices through the sorted data with the precision of a blade, dividing its domain by half with each incantation. It's a dance of divide and conquer, where each step brings our heroes closer to their elusive target, provided their world is ordered and their minds are clear.

By splitting each data set right down the middle, it tries to find the data in much, much less time than it should. However, there is one feature that Binary Search requires in the data set it is looking for, the set must be ordered. Otherwise it will not survive and its humble soul will have to leave our world.

```
102 public class BinarySearch {
103     public static int search(int[] arr, int target) {
104         int left = 0;
105         int right = arr.length - 1;
106         while (left <= right) {
```

```
            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {
                return mid;
            }

            if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
}
```

3 Results, Analysis, Discussion

Let's delve deeper into hidden secrets, blending the challenges of analysis with our ongoing saga.

As our heroes studied the mystical Table of Running Times, they realized the changing powers of sorting spells. Similar to organizing individual scrolls in a library, Add Sort revealed its time-consuming nature, especially as the library grew, growing in complexity like a quadratic spell ($O(n^2)$) and quadrupling its time with every doubling of data.

In contrast, the Merge Sort, which splits data into smaller areas to capture it more efficiently, demonstrated its power by scaling gracefully with the growth of the data, a testament to its logarithmic fit (O(n log n)), doubling its speed with each jump in data size.

Within the algorithmic arsenal, our heroes discovered the Counting Sort spell, a technique that does not depend on comparisons, but counts the occurrences of each element. This magic shines in realms where the data is not so vast, working in linear time (O(n)) under the right conditions. When they applied this method, they realized its efficiency in sorting their numerical artifacts, especially when the numbers are clustered in a manageable range. Unlike the fragmented power of laborious Insertion Sort or Merge Sort, Counting Sort categorizes data quickly, offering a beacon of efficiency in the chaos of the digital maze. But like any powerful spell, it had a downside: you had to know in advance the mightiest, greatest value in the data. The other two sorting spells had no such restriction, making the Counting Sort spell available to us only in certain circumstances.

Turning their gaze to the search guards, they noticed the steady but slow progress of the Linear Search, scanning every nook and cranny with a time directly proportional to the data (O(n)). In contrast, Binary Search, with its clever divide-and-conquer strategy, sliced and diced the sorted data with logarithmic precision (O(log n)), unaffected by the mere doubling of the data.

In this grand analysis, our heroes revealed the intrinsic nature of their algorithms, guiding them to choose wisely in their future endeavors and using their knowledge as both shield and sword in the ongoing quest through the data realm.

They jotted down the output of their spells in the inks of purple squid that roam the choppy

seas, on ageless paper made from trees felled from massive grove forests. This way they could test the accuracy of the information they had heard from their ancestors, from libraries and other bards, and when these spells had worked. Their experiments did not surprise them. Our adventurers had indeed obtained the values mentioned in the tests with data of different sizes. Which reminded them again: "You can't go back on what the ancestors said".

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	0.2	0.4	0.2	0.7	2.6	10.2	39.8	173.2	664.8	2538.6
Merge sort	0.2	0.4	0.1	0.1	0.2	0.4	0.8	1.8	3.6	8.9
Counting sort	64.8	64.7	63.9	63.1	63.7	63.5	66.9	70.1	67.2	72.9
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1
Merge sort	0.0	0.1	0.0	0.0	0.2	0.4	0.7	1.6	3.2	6.7
Counting sort	65.7	66.3	70.6	65.7	65.3	65.9	68.4	66.5	66.1	69.7
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.1	0.4	1.3	5.1	22.2	78.6	326.8	1311.0	4919.0
Merge sort	0.0	0.0	0.0	0.1	0.2	0.4	0.8	1.7	3.2	6.7
Counting sort	65.4	65.5	66.0	65.4	65.2	65.4	66.4	73.7	66.4	66.8

In this mysterious library, our heroes were confused by only one thing. The Counting Sort spell did not work the way it was written in the ancient scrolls. While the ancient books said that it should increase in direct proportion to the size of the data set, that is, it should be of O(n+k) complexity, the results of our heroes' experiments showed them that they had a spell that seemed to be independent of the size of the data, or, in magic parlance, of O(1) complexity. No matter how many times they repeated the results of their experiments, they could not find anything different. No matter how big the data set got, Counting Sort hovered around the same neighbourhood, not even doubling. Pondering over this and continuing to search the ancient scriptures, the powerful sorcerer came to the conclusion that the K value was so much larger than the length of the array, N, that no matter how much N increases, the dominant expression is the K value, so that the duration of the Counting Sort almost never changes.

And the elven rogue in the field realised this with his sly eyes: The spell called insertion sort already worked almost flawlessly and instantly when it came to ordered lists, but much, much slower when it came to unordered lists. The team's wizard explained this with these ancient words: "Insertion Sort compares each element and replaces the out-of-place ones one by one in order to replace them. If the list is already sorted, it doesn't need to do anything but compare. If the list is in reverse order, it has to replace them all!"

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

In our enchanted narrative, as the data knights transitioned to their search enchantments, they faced new trials. They realized that not all spells could be cast upon every tome of data. The Binary Search spell, for instance, required a lexicon ordered as meticulously as the stars in the sky, limiting its use to such arranged archives. Initial tests with smaller datasets revealed unexpected

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)	71.026	92.989	156.065	268.046	497.237	919.787	1826.257	3392.313	6420.476	12751.129
Linear search (sorted data)	67.504	103.184	201.718	334.577	611.95	1270.389	2402.778	4858.325	9739.745	19349.129
Binary search (sorted data)	62.837	65.323	66.51	75.31	78.328	80.99	84.471	76.137	83.244	94.251

time extensions, a mystery that puzzled our heroes. Despite this anomaly, Linear Search's time grew with the data's expanse, true to its O(N) nature. Binary Search, however, remained steadfast, its duration hardly wavering, embodying the essence of O(log N) efficiency, a constant in the ever-expanding universe of data.

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	$\Omega(1)$	$\Theta(n/2)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

In our narrative, the enchanted table reveals the essence of each algorithm's power. Insertion Sort, a meticulous spell, varies from linear to quadratic intensity in its performance. Merge Sort, a spell of division and unity, consistently operates in a realm of n log n complexity, showcasing its efficiency. Counting Sort's magic aligns with n+k, hinting at its dependence on the data's range. Linear Search, simple yet steadfast, ranges from instantaneous discovery to a full traversal. Binary Search, a model of efficiency, maintains its logarithmic pace, unaffected by the data's scale, illustrating the profound wisdom embedded in each algorithm's core.

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	$O(1)$

The table of Auxiliary Space Complexity stands as a testament to the additional memory required by our valiant algorithms beyond their primary data. Insertion Sort, akin to a solitary wizard, requires merely a staff—a symbol of its $O(1)$ complexity, signifying its minimal reliance on

additional space. It performs all the sorting operations in-place. Merge Sort, a gathering of mages, calls upon a great hall—$O(n)$ space—to orchestrate its magic, reflecting its need for a proportional expanse to host its divided elements. In Merge Sort example code, line 19 and 20 indicates two new arrays that sums up to our beginning array's space. Counting Sort, with its reliance on the range k of the input, summons a variable-sized chamber, $O(n + k)$, tailored to the data's spread. As everyone can see on line 73 and 74, it creates two new arrays, one with k+1 elements and other with first array's element count elements. The search spells—Linear and Binary—stand firm with only their wands, $O(1)$, a testament to their efficiency and precision, requiring no more than their own essence to sift through the data realms.

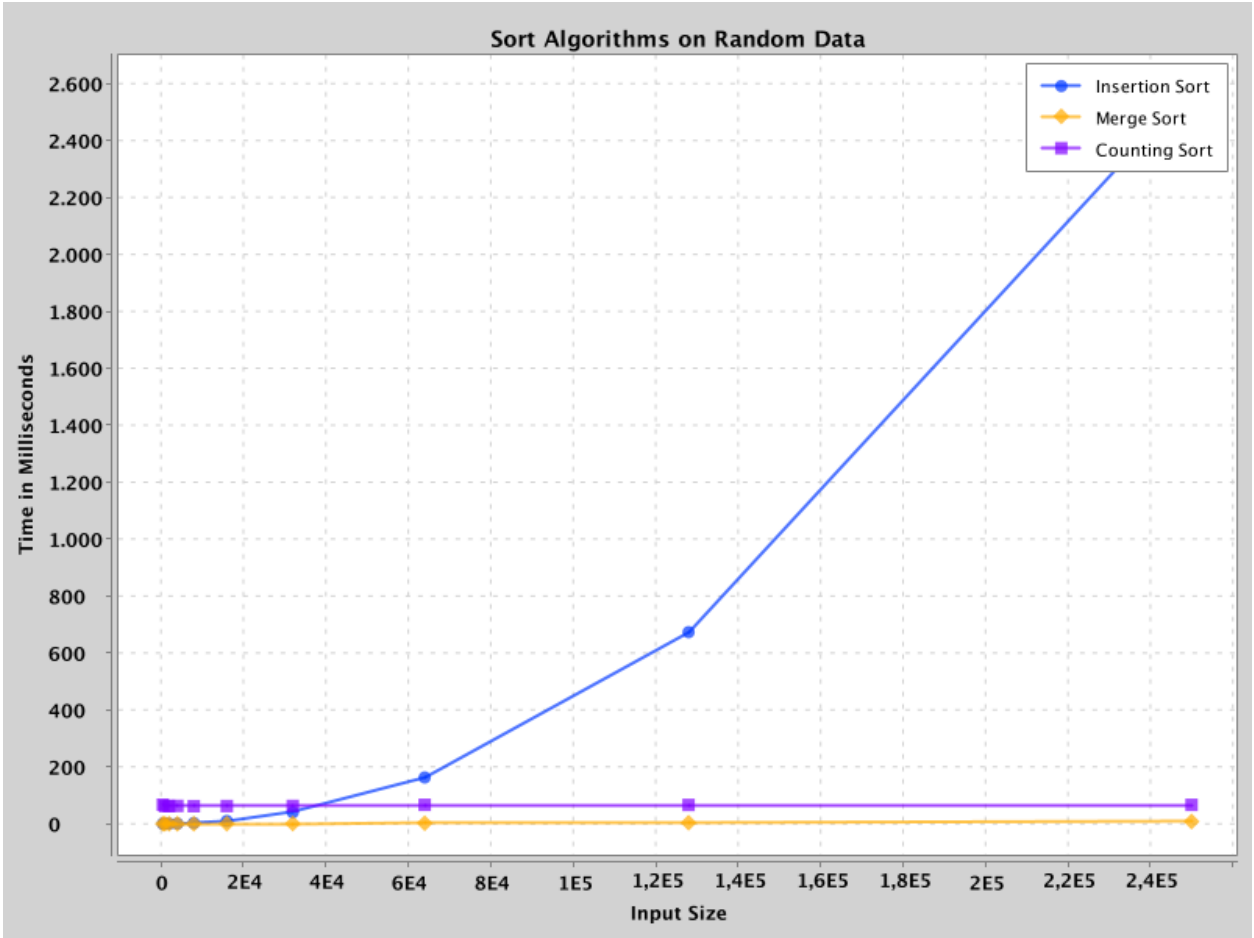

Figure 1: Sorting algorithms on Random Data

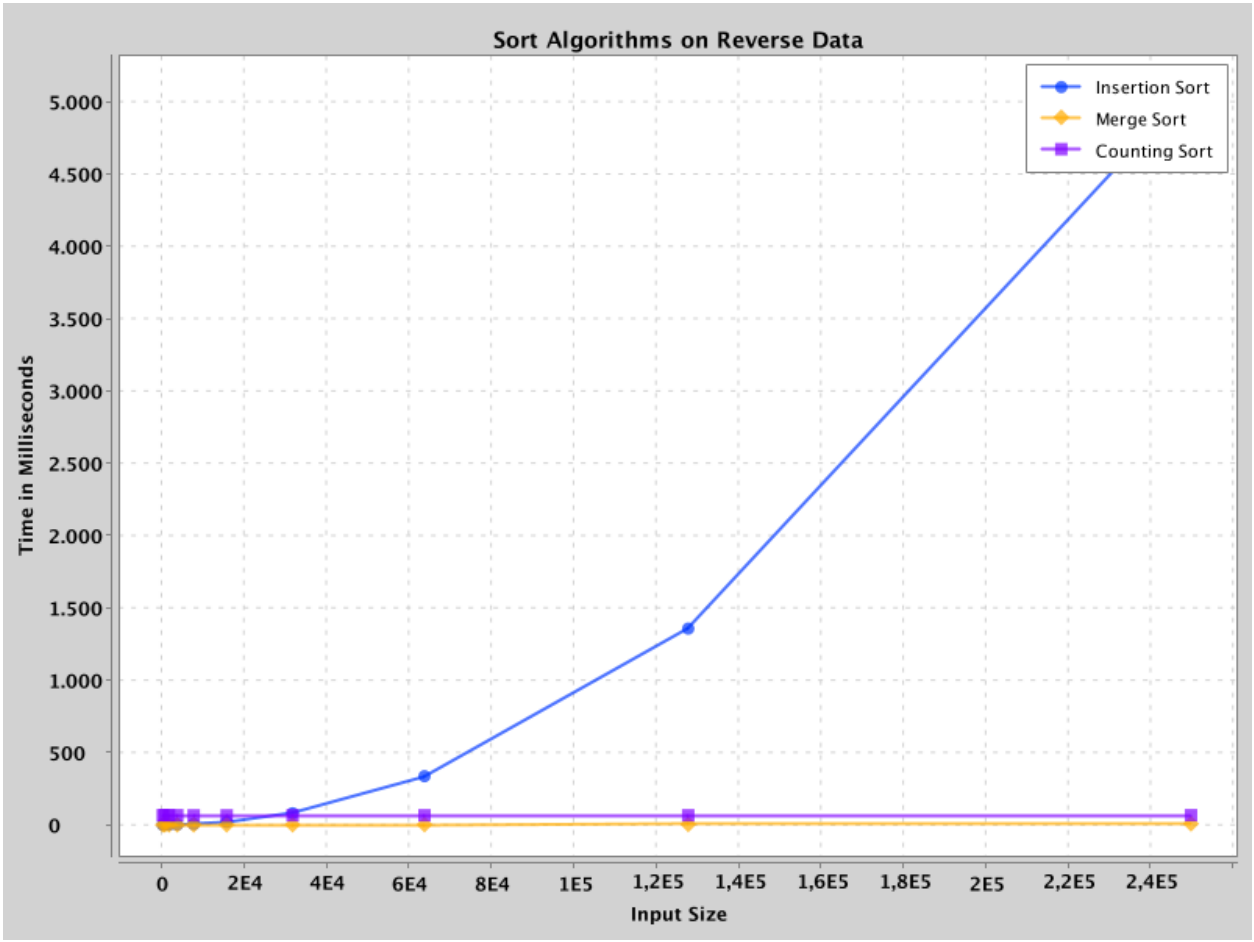

Figure 2: Sorting algorithms on Reverse Sorted Data

Here, it is clearly seen how much the time taken by Insertion Sort has increased, unlike the other graph. We also saw a quadratic increase on Random Data, but here the numbers are much larger because the number of operations to be performed is much larger.

In fact, Merge Sort is also increasing by doubling continuously, but the time it takes is so small that this increase cannot be fully observed in the graph.

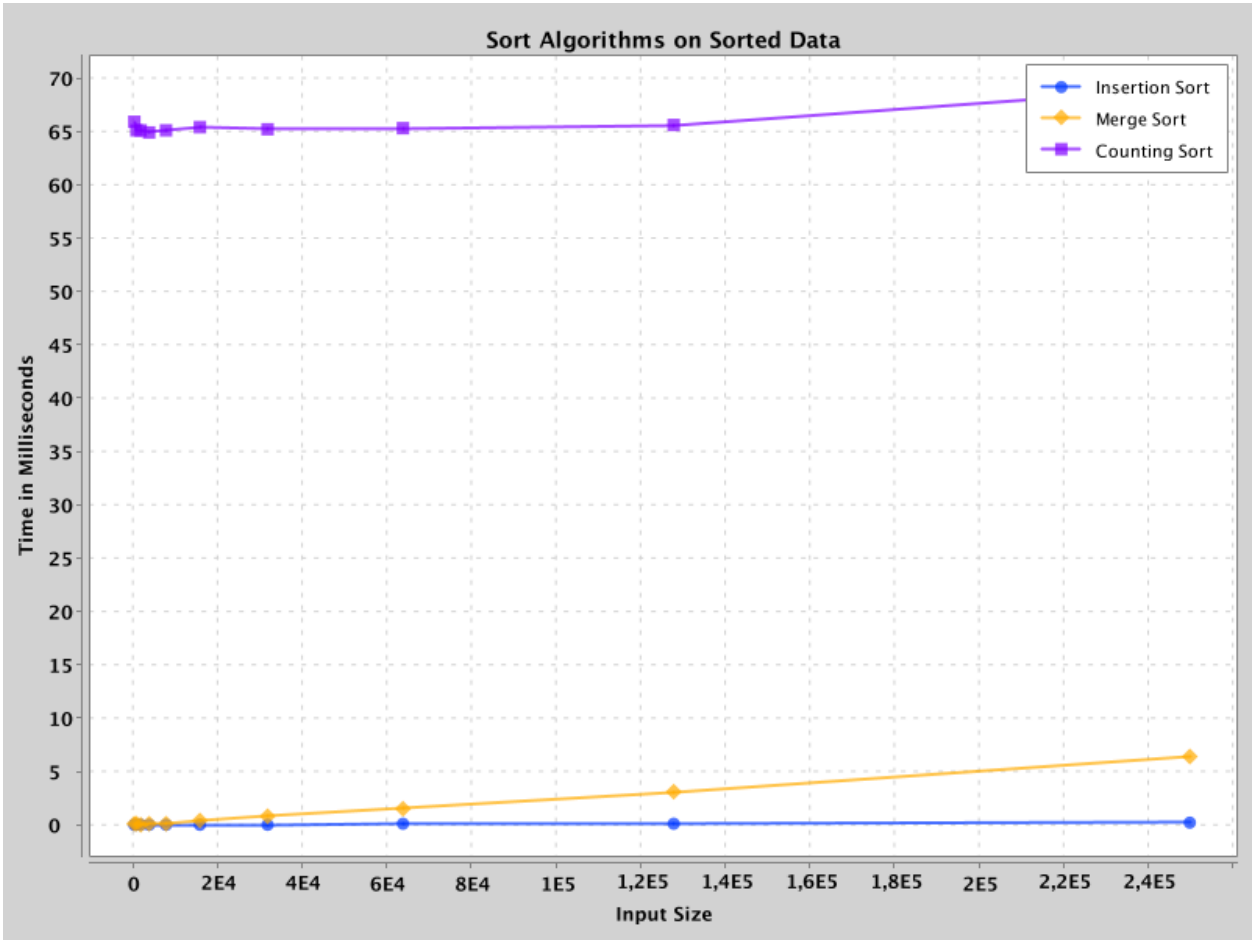

Figure 3: Sorting algorithms on Already Sorted Data

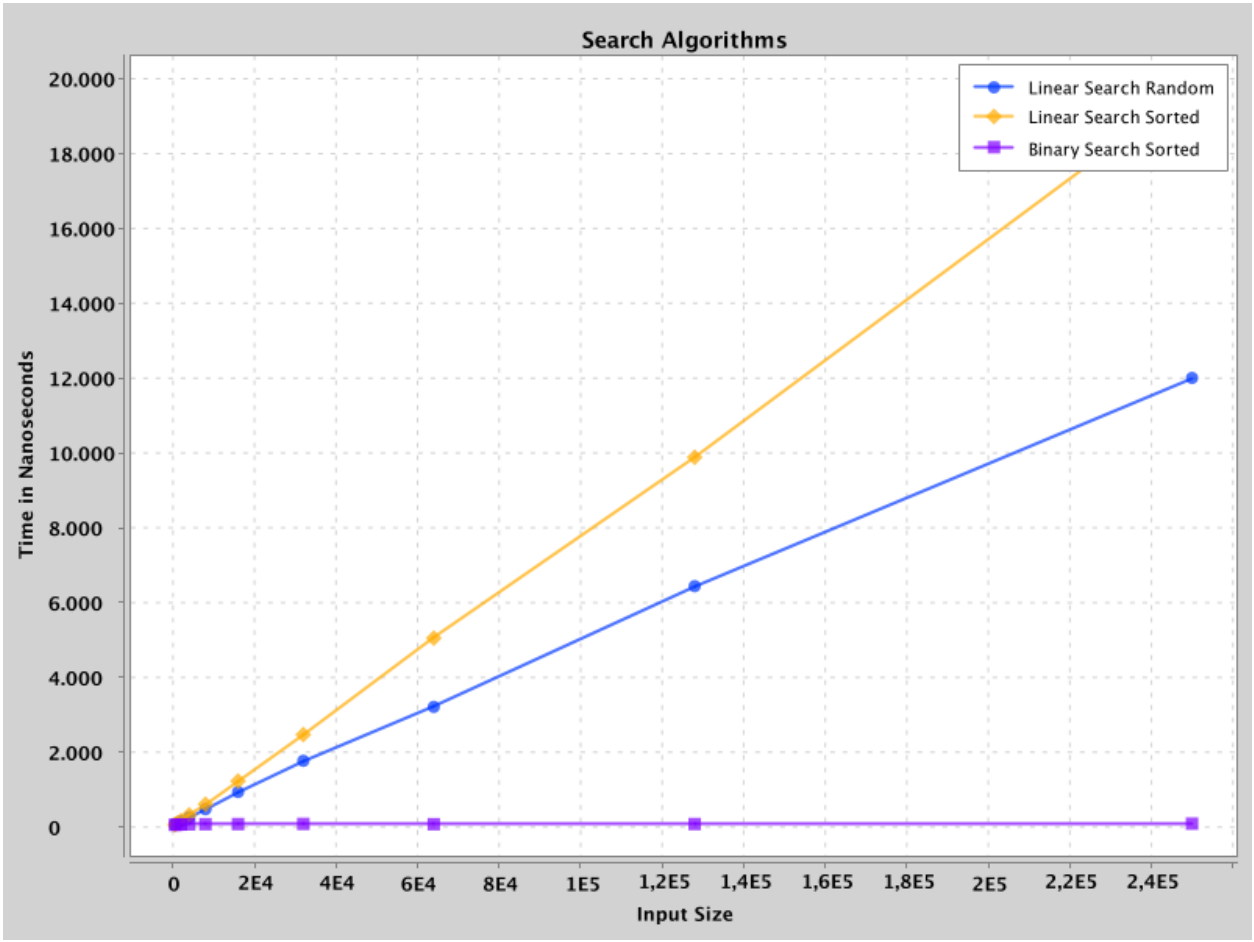

Figure 4: Searching Algorithms on Various Data