Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 21, 2024

Student name:
Mustafa Ege

Student Number:
b2210356088

1 Problem Definition

Efficient sorting is key to improving the performance of other algorithms like search and merge operations, which rely on sorted input data. In today's world of abundant information accessible through computing and the internet, being able to search through data quickly is essential. Testing a sorting algorithm's efficiency involves sorting datasets of various sizes and characteristics to see how well it performs in different scenarios.
Our aim here is to analyze different sorting and searching algorithms and compare their running times on a number of inputs with changing sizes.

2 Solution Implementation

Sorting algorithms used are 2.1 insertion sort, 2.2 merge sort and 2.3 counting sort. Searching algorithms used are 2.4 linear search and 2.5 binary search. Each implementation shows the algorithms in Java.

2.1 Insertion Sort

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

2.2 Merge Sort

```java
public class MergeSort {
    public static int[] mergeSort(int[] A) {
        int n = A.length;
        if (n <= 1)
            return A;
        int[] left = new int[n / 2];
        int[] right = new int[n - n / 2];
        System.arraycopy(A, 0, left, 0, n / 2);
        System.arraycopy(A, n / 2, right, 0, n - n / 2);
        left = mergeSort(left);
```

```
        right = mergeSort(right);
        return merge(left, right);
    }
    public static int[] merge(int[] A, int[] B) {
        int[] C = new int[A.length + B.length];
        int i = 0, j = 0, k = 0;
        while (i < A.length && j < B.length) {
            if (A[i] > B[j])
                C[k++] = B[j++];
            else
                C[k++] = A[i++];
        }
        while (i < A.length)
            C[k++] = A[i++];
        while (j < B.length)
            C[k++] = B[j++];
        return C;
    }
}
```

2.3 Counting Sort

```
public class CountingSort {
    public static int[] countingSort(int[] A) {
        int size = A.length;
        int k = 0;
        for (int i = 0; i < size; i++) {
            k = Math.max(k, A[i]);
        }
        int[] count = new int[k + 1];
        int[] output = new int[size];
        for (int i = 0; i < size; i++) {
            int j = A[i];
            count[j]++;
        }
        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }
        for (int i = size - 1; i >= 0; i--) {
            int j = A[i];
            output[count[j] - 1] = A[i];
            count[j]--;
        }
        return output;
    }
}
```

2.4 Linear Search

```
67 public class Search {
68     public static int linearSearch(int[] A, int x) {
69         int size = A.length;
70         for (int i = 0; i < size; i++) {
71             if (A[i] == x)
72                 return i;
73         }
74         return -1;
75     }
76 }
```

2.5 Binary Search

```
77 public class Search {
78     public static int binarySearch(int[] A, int x) {
79         int low = 0;
80         int high = A.length - 1;
81         while (high - low > 1) {
82             int mid = (high + low) / 2;
83             if (A[mid] < x)
84                 low = mid + 1;
85             else
86                 high = mid;
87         }
88         if (A[low] == x)
89             return low;
90         else if (A[high] == x)
91             return high;
92         return -1;
93     }
94 }
```

3 Results, Analysis, Discussion

Average running time test results for sorting algorithms are given in milliseconds in Table 1, for search algorithms are given nanoseconds in Table 2. Complexity analysis tables showing computational complexities and auxiliary space complexities are given in Table 3 and Table 4.
Normally, computational times obtained have fractional parts in milliseconds and nanoseconds, but these fractional parts of the obtained results has been deducted to increase readability.

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	0	0	1	6	24	88	356	1442	5925
Merge sort	0	0	0	0	1	2	4	8	16	36
Counting sort	230	158	158	158	158	158	158	159	174	160
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	0
Merge sort	0	0	0	0	1	2	3	6	11	24
Counting sort	0	0	0	0	0	0	1	1	2	171
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	2	6	24	94	352	1459	5991
Merge sort	0	0	0	0	1	2	4	9	17	35
Counting sort	233	158	158	158	158	158	158	159	160	159

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)	2120	1553	674	1112	426	3224	6247	12563	24439	49479
Linear search (sorted data)	3067	678	1122	1990	3341	6244	12962	24614	48695	59351
Binary search (sorted data)	469	278	294	260	230	232	238	244	249	551

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

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

Examining the best, average, and worst cases for the algorithms, and checking if obtained results match their theoretical asymptotic complexities while also proving theoretical computational and auxiliary space complexities of the given algorithms:

- Insertion Sort. Best case for insertion sort is the sorted data, only iterating over the list, making no changes. From the table 1 it can be seen that all running times are near zero. Moreover, average and worst case are the random and reversely sorted data respectively, as it means making a lot of swapping operations on the data. Both of them have quadratic computational complexities as seen in 3 and the test results are increasing quadratically. The space complexity is O(1), since it is done in-place, no auxiliary.

- Merge Sort. The best case for merge sort would happen when the elements are sorted, since already sorted elements would decrease the number of comparisons, the worst case would happen when elements are reversely sorted, leaving average case to be the randomly positioned. However, the effect would be so low since for all cases computational complexity is $(n \log n)$ as seen from 3. Also, in the 1 it can be seen that theoretical complexity match with the results, since with the huge amount of data, test time is not changing much, only increasing logarithmically. Only downside is the space complexity, with O(n), as merge sort uses an auxiliary array.

- Counting Sort. The best case for counting sort occurs when all items are in the same range, or when k is equal to 1, counting the occurrences of each element is constant time and finding the correct index value of each element in the sorted output array takes n time, so O(n+1) which is O(n). The worst case happens with the skewed data, when the largest element is much larger than the other elements, resulting in O(n+k) complexity. For big values of k like n^2 or n^3 and greater k dominates and it results in O(k). For the average case, we can compute it with a fixed n and k differing from 0 to infinity then varying n as well. The result would become O(n+k) for the average case. In the test, our data is formed out of both small and big numbers, resulting a big k, especially when the data is sorted. It can be seen from the the table 1, but more clearly in the graph 2 we can see that when the data is sorted, the big numbers are accumulated at the end, resulting a huge k, also n is quite big since the input size is increased at the right side, therefore the graph for counting sort rises dramatically. We can also conclude from the table and graph that theoretical complexity and tested results match with each other. Space complexity is O(k), since an auxiliary array with the size of the biggest number (k) is used,

In search operations, in order to show the most average case, the element being searched is chosen as the element in the middle of the data. For instance, with the random data of 500, the search element is the element at the index 250 or with the data of 64000, the search element is at index 32000. Note that we only get the middle number from the random data, not the sorted. This is the reason why there are some deviations of the computational times between linear search of random and sorted data.

- Linear Search. The best case for linear search occurs when the element being searched is found in the first position, resulting in constant $\Omega(1)$ time. The average case occurs when the element being searched in the middle so making n/2 comparisons, and resulting $\Theta(n/2)$ complexity. Lastly, worst case is similar to average with the element being at the end, making n comparisons, resulting O(n) complexity. It can be seen from 4 that linear search with random data has the average complexity of n/2, also linear search with sorted data has a complexity near to the average complexity. However, probably because of the data in the middle is rather greater than the rest of the data, the computational time increases. We can also observe that complexities match from the table 2 especially after input size 16000. Before that with the random data, the data might be encountering the same number before the searched middle number. Auxiliary space complexity is O(1) since no additional space used, done in place.

- Binary Search. The best case for binary search is similar with the linear search, only difference is instead of first position, the search number needs to be in the central index of the data, resulting in $\Omega(1)$ time. Average case is whenever the search number is not the number in the middle, and using only half of the data every time, resulting in $\Theta(\log n)$ complexity. The worst case happens when the number to be searched is in the beginning or at the end of the data, in other words if it is smallest or the greatest element in the list, but the complexity is still $\Theta(\log n)$, meaning that the computational time does not increase as much with n like the linear search. In table 2 it can be seen that the time results are very similar with each other with all input sizes, proving the computational complexity of logn, and in graph 4 we can see that binary search's computational time is very low at the bottom, a fast working algorithm.

4 Notes

It might be best to do the all experiments individually to get the most reliable results and graphs, since running all the experiments all at the same time may possibly result in exceeding the limits of the computer. With the data used in this experiment, it doesn't make a huge difference, but I think running individually shows the results better, so I obtained the the results and graphs this way.

References

- https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm
- https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/

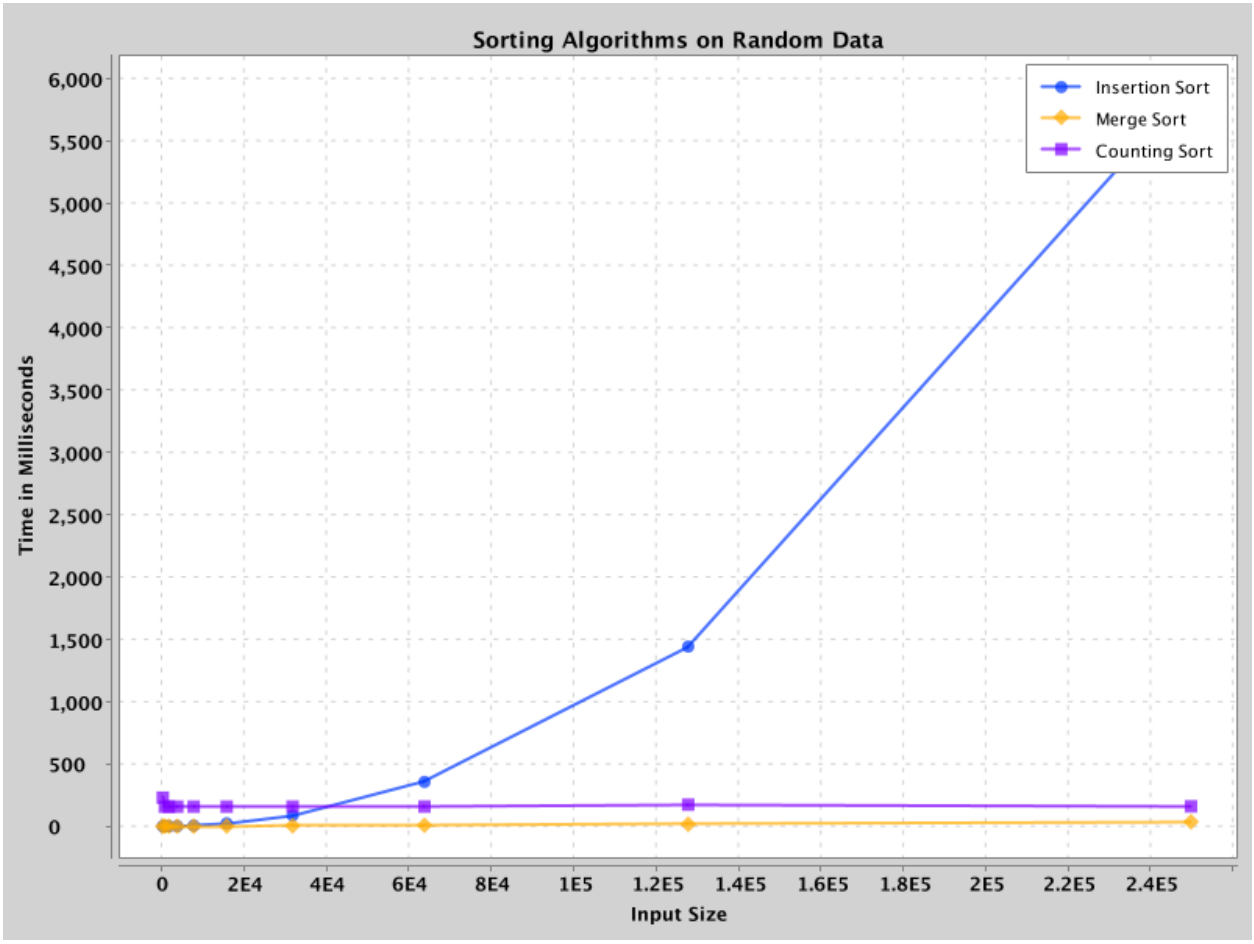

Figure 1: Sorting Algorithms' Running Times on Random Data

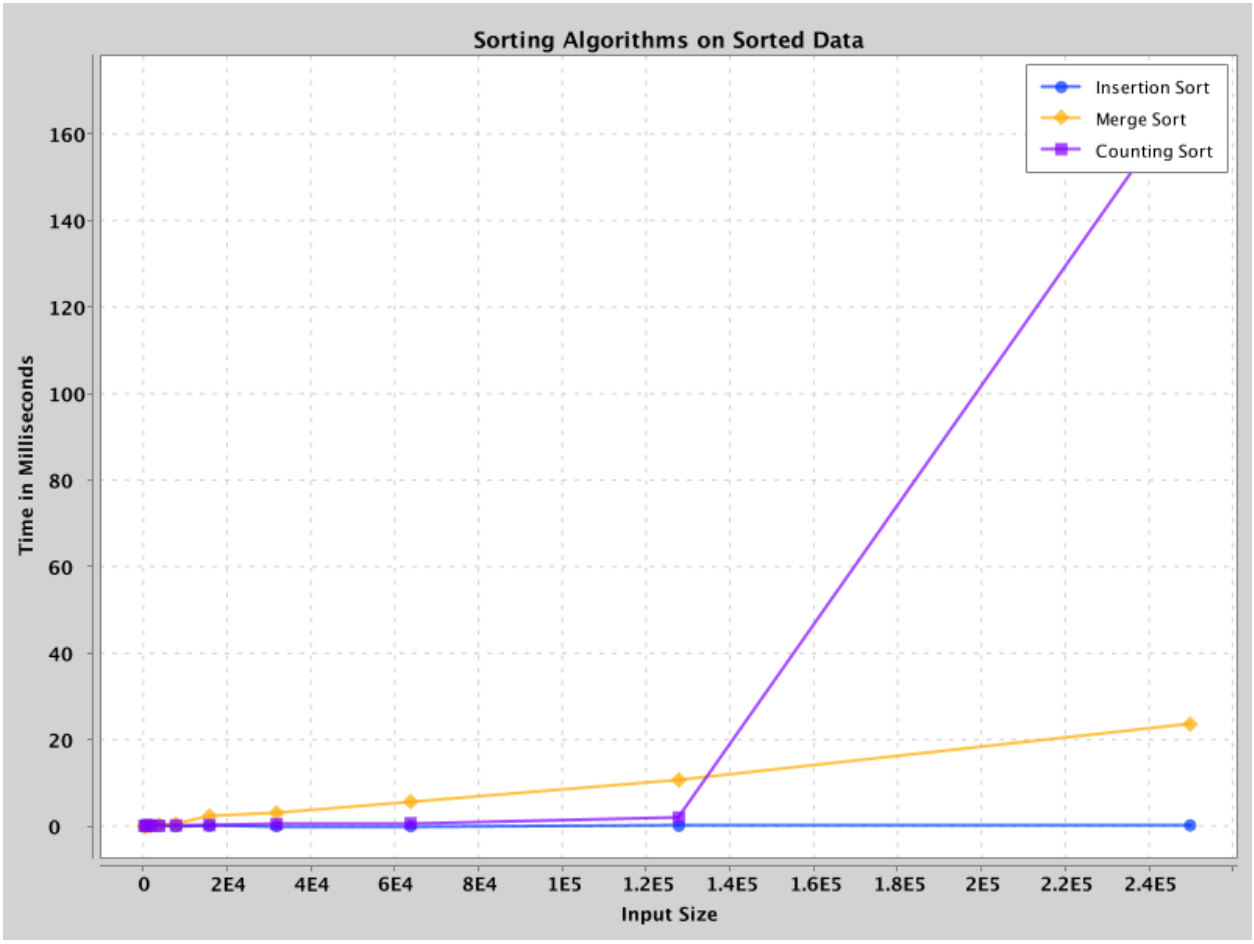

Figure 2: Sorting Algorithms' Running Times on Sorted Data

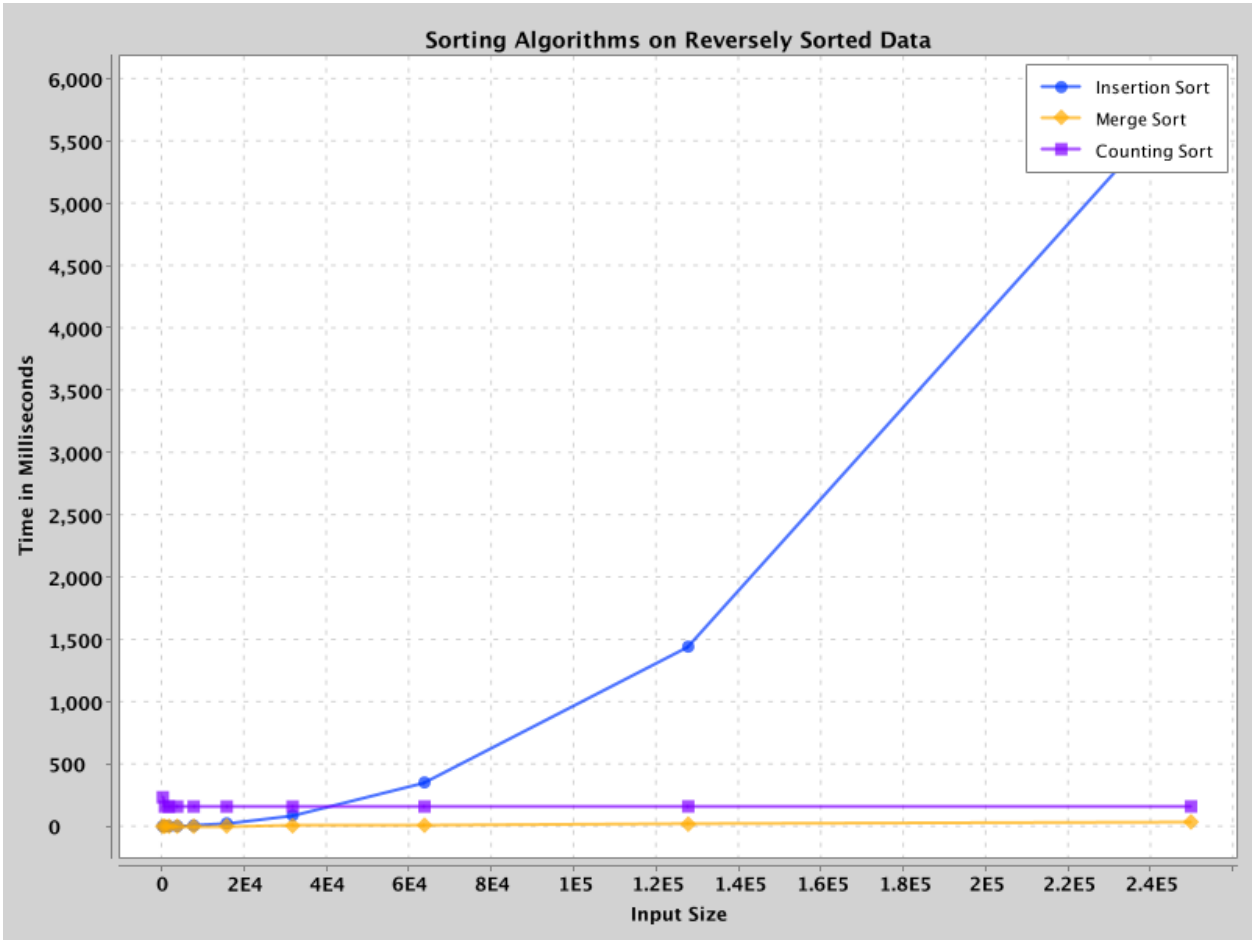

Figure 3: Sorting Algorithms' Running Times on Reversely Sorted Data

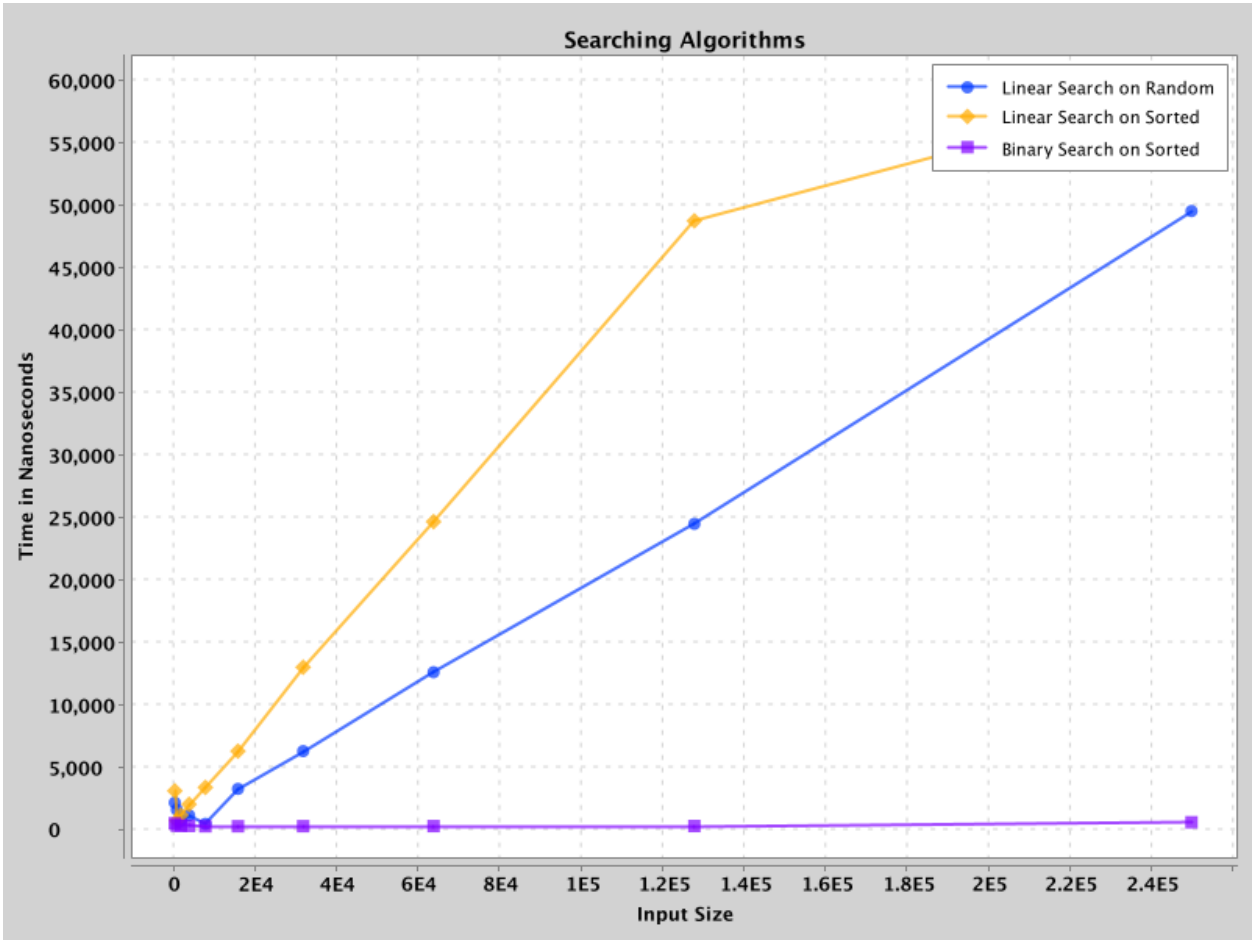

Figure 4: Searching Algorithms' Running Times