Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 29, 2023

Student name:
Doğukan Aytekin

Student Number:
b2200356003

1 Problem Definition

Analyze different sorting and searching algorithms and compare their running times on a number of inputs with changing sizes.

2 Solution Implementation

All of my sorting and searching implementations, also even though I don't include them here I have one measurement functions for every implementation to test them.

2.1 Selection Sort

```java
public class SelectionSort {
    public static void selectionSort(int[] flowDuration, int size) {
        for (int i = 0; i < size - 1; i++) {
            int index = i;
            for (int j = i + 1; j < size; j++) {
                if (flowDuration[j] < flowDuration[index]) {
                    index = j;//searching for lowest index
                }
            }
            int smallerNumber = flowDuration[index];
            flowDuration[index] = flowDuration[i];
            flowDuration[i] = smallerNumber;
        }
    }
```

2.2 Quick Sort

```java
public class QuickSort {
    public static void quickSort(int[] flowDuration,int low,int high){
        int stackSize = high-low+1;
        int[] stack = new int[stackSize];
        int top = -1;
        stack[++top]=low;
        stack[++top]=high;
        while (top>=0){
            high=stack[top--];
            low=stack[top--];
            int pivot = Partition(flowDuration,low,high);
            if (pivot-1>low){
                stack[++top]=low;
                stack[++top]=pivot-1;
            }
            if (pivot+1<high){
```

```
                stack[++top]=pivot+1;
                stack[++top]=high;
            }
        }
    }
```

2.3 Bucket Sort

```
public class BucketSort {
    public static int[] bucketSort(int[] flowDuration){

        int numberOfBuckets = (int) Math.sqrt(flowDuration.length);
        ArrayList[] buckets = new ArrayList[numberOfBuckets];

        int max = flowDuration[0];
        for (int i = 0; i < flowDuration.length; i++) {
            if (flowDuration[i]>max){
                max = flowDuration[i];
            }
        }

        for (int i = 0; i < numberOfBuckets; i++) {
            buckets[i] = new ArrayList();
        }

        for(int i : flowDuration){
            buckets[hash(i,max,numberOfBuckets)].add(i);
        }

        for (ArrayList bucket : buckets){
            Collections.sort(bucket);
        }

        ArrayList<Integer> sortedArrayList = new ArrayList<>();
        for(ArrayList bucket : buckets){
            for (int i = 0; i < bucket.size(); i++) {
                sortedArrayList.add((Integer) bucket.get(i));
            }
        }

        for (int i = 0; i < flowDuration.length; i++) {
            flowDuration[i] = sortedArrayList.get(i);
        }


        return flowDuration;
    }
```

```java

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

2.4 Linear Search

```java
public class LinearSearch {
    public static int linearSearch(int[] flowDuration,int x){
        int size = flowDuration.length;
        for (int i = 0; i < size; i++) {
            if (flowDuration[i]==x){
                return i;
            }
        }
        return -1;
    }
```

2.5 Binary Search

```java
public class BinarySearch {
    public static int binarySearch(int[] flowDuration,int x){
        int low = 0;
        int high = flowDuration.length-1;
        while ((high-low)>1){
            int mid = (high+low)/2;
            if (flowDuration[mid]<x){
                low = mid+1;
            }
            else {
                high=mid;
            }
        }
        if (flowDuration[low]==x){
            return low;
        }
        else if (flowDuration[high]==x){
            return high;
        }
        return -1;
    }
```

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									
Selection sort	0.1	0.2	0.8	3.3	13.1	51.7	208.5	830.2	3286.7	12465.5
Quick sort	0.1	0.1	0.1	0.2	0.3	0.7	2.6	9.7	28.5	50.2
Bucket sort	0.3	0.3	0.4	0.5	1.0	2.1	4.0	8.8	17.1	35.6
	Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.2	0.7	3.1	12.7	52.9	209.5	842.9	3310.8	12672.6
Quick sort	0.1	0.3	1.3	5.4	22.0	89.3	355.6	1404.1	5639.1	22513.5
Bucket sort	0.1	0.1	0.1	0.1	0.2	0.3	0.5	1.0	2.0	4.3
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.3	1.1	4.5	17.3	78.9	314.7	1253.3	5182.3	22762.3
Quick sort	0.1	0.2	0.5	1.9	5.7	10.9	36.4	159.8	604.2	1282.1
Bucket sort	0.0	0.0	0.1	0.1	0.2	0.4	0.8	1.4	2.7	6.0

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)	881.2	1170.6	171.0	281.8	514.7	1524.6	1637.6	3423.6	6835.5	15295.1
Linear search (sorted data)	61.2	98.9	168.8	401.0	777.6	1581.4	3109.8	6530.1	13333.1	25525.1
Binary search (sorted data)	363.3	164.1	235.3	97.0	129.3	157.6	210.9	145.9	121.7	111.8

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**
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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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	$O(1)$

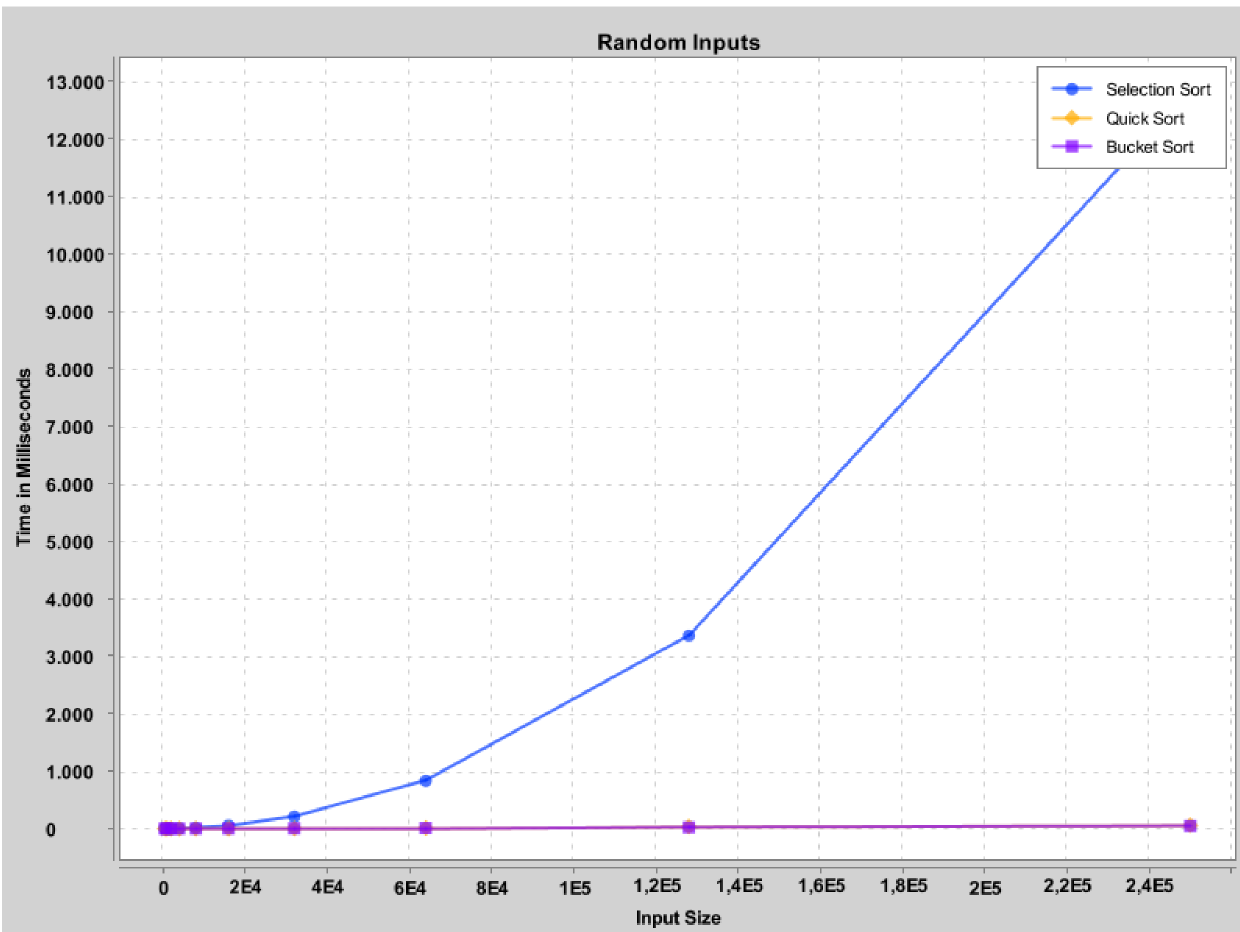

Figure 1: Sorting experiment with random inputs.

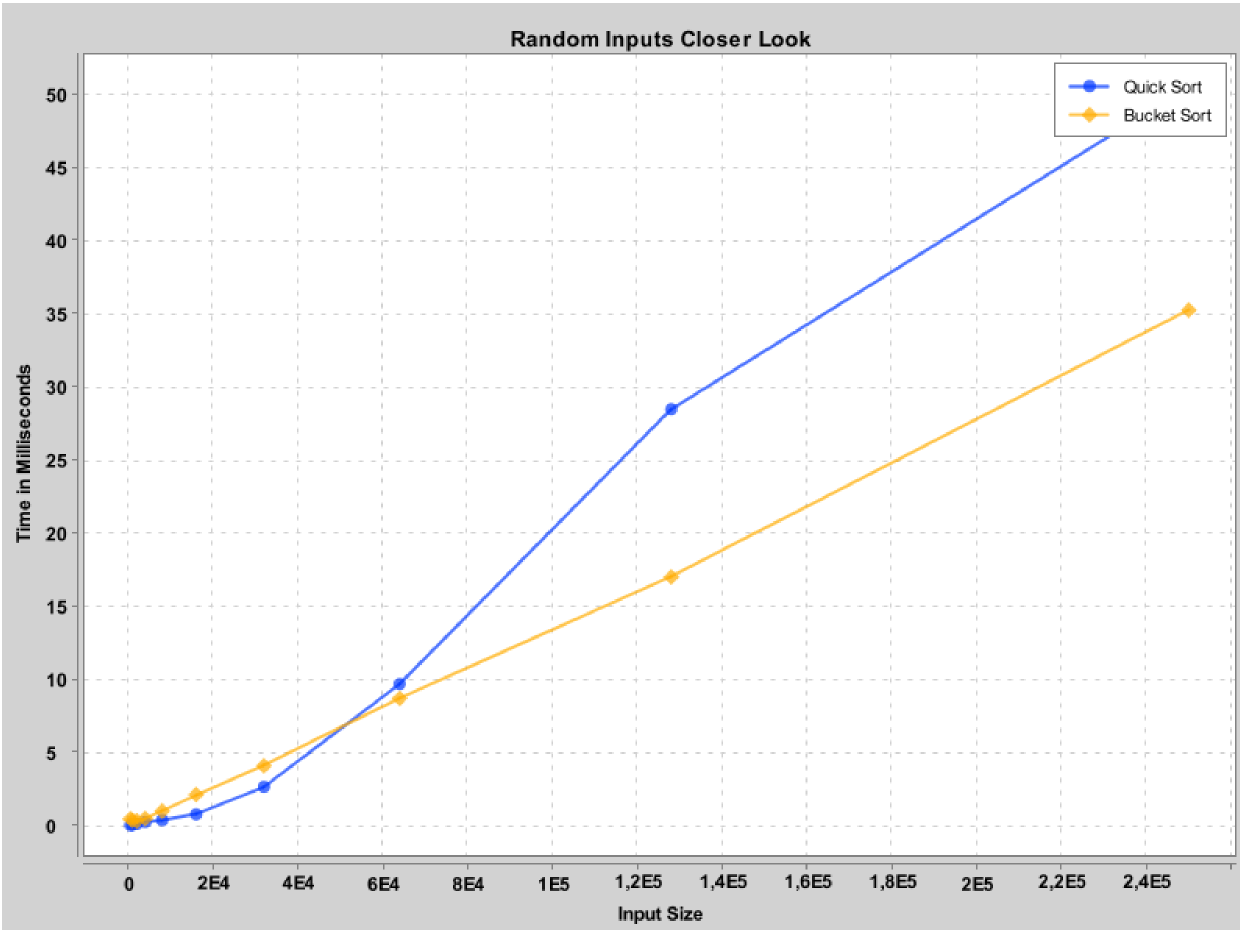

Figure 2: Closer look on Random Inputs (QuickSort and BucketSort).

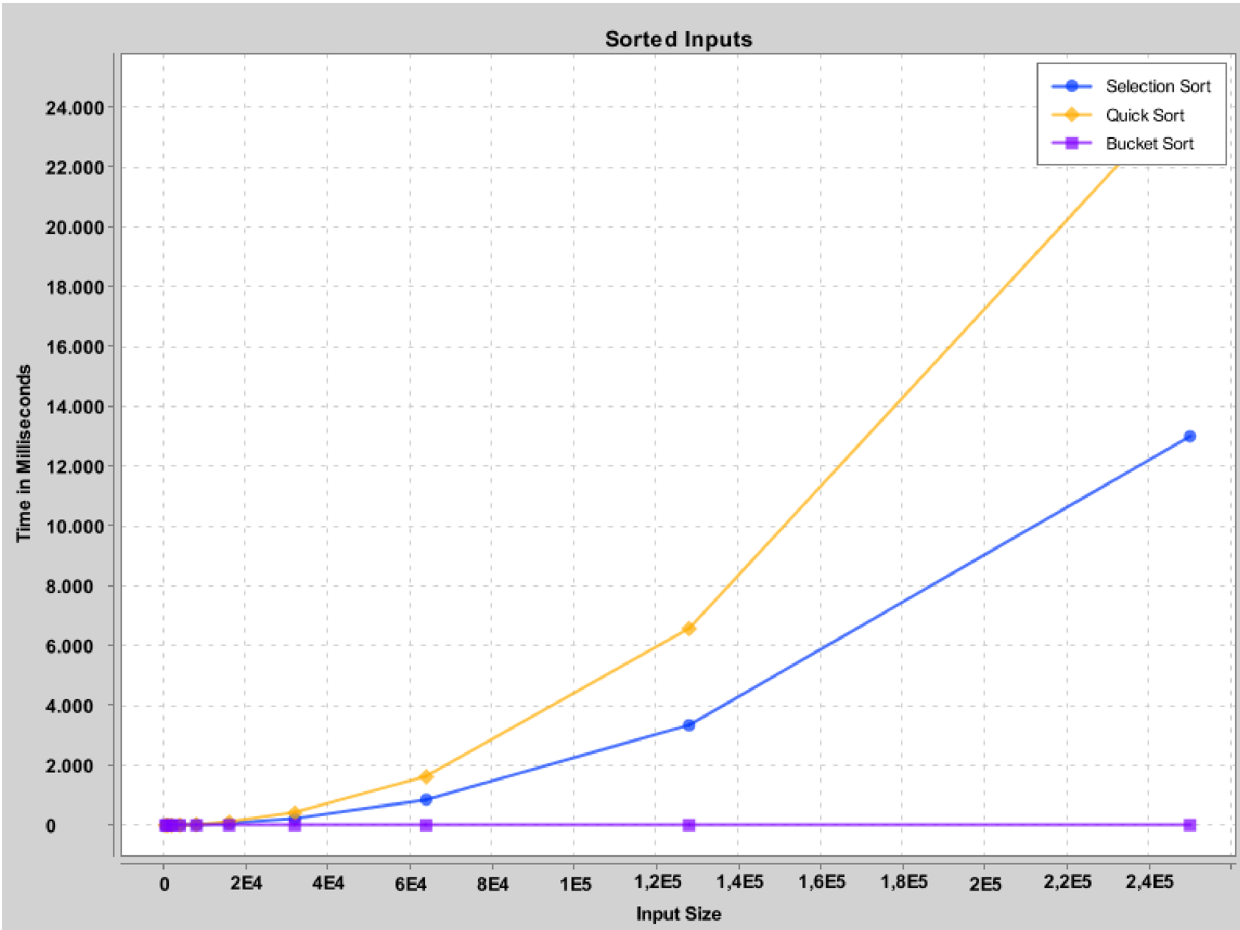

Figure 3: Sorting experiment with sorted inputs.

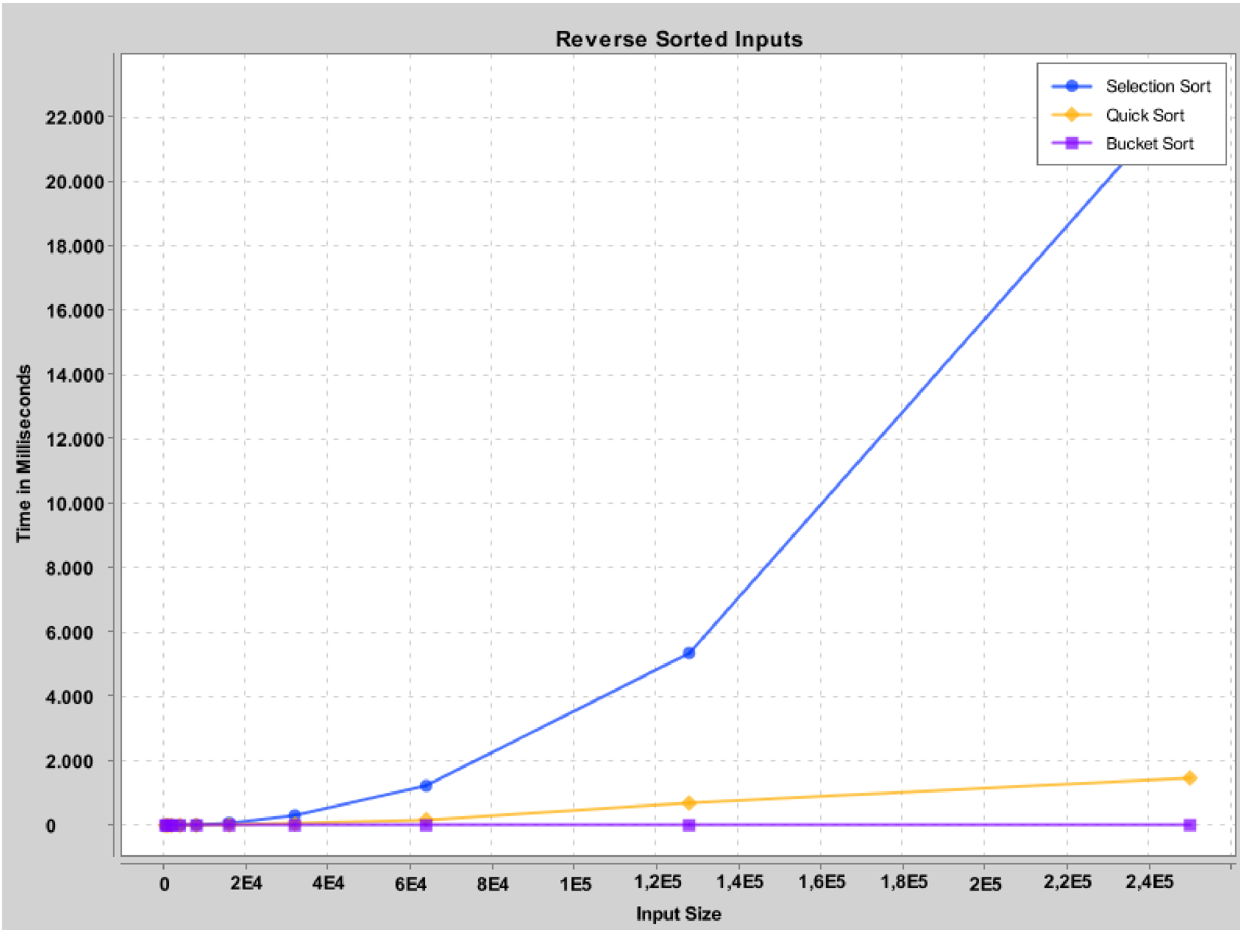

Figure 4: Sorting experiment with reverse sorted inputs.

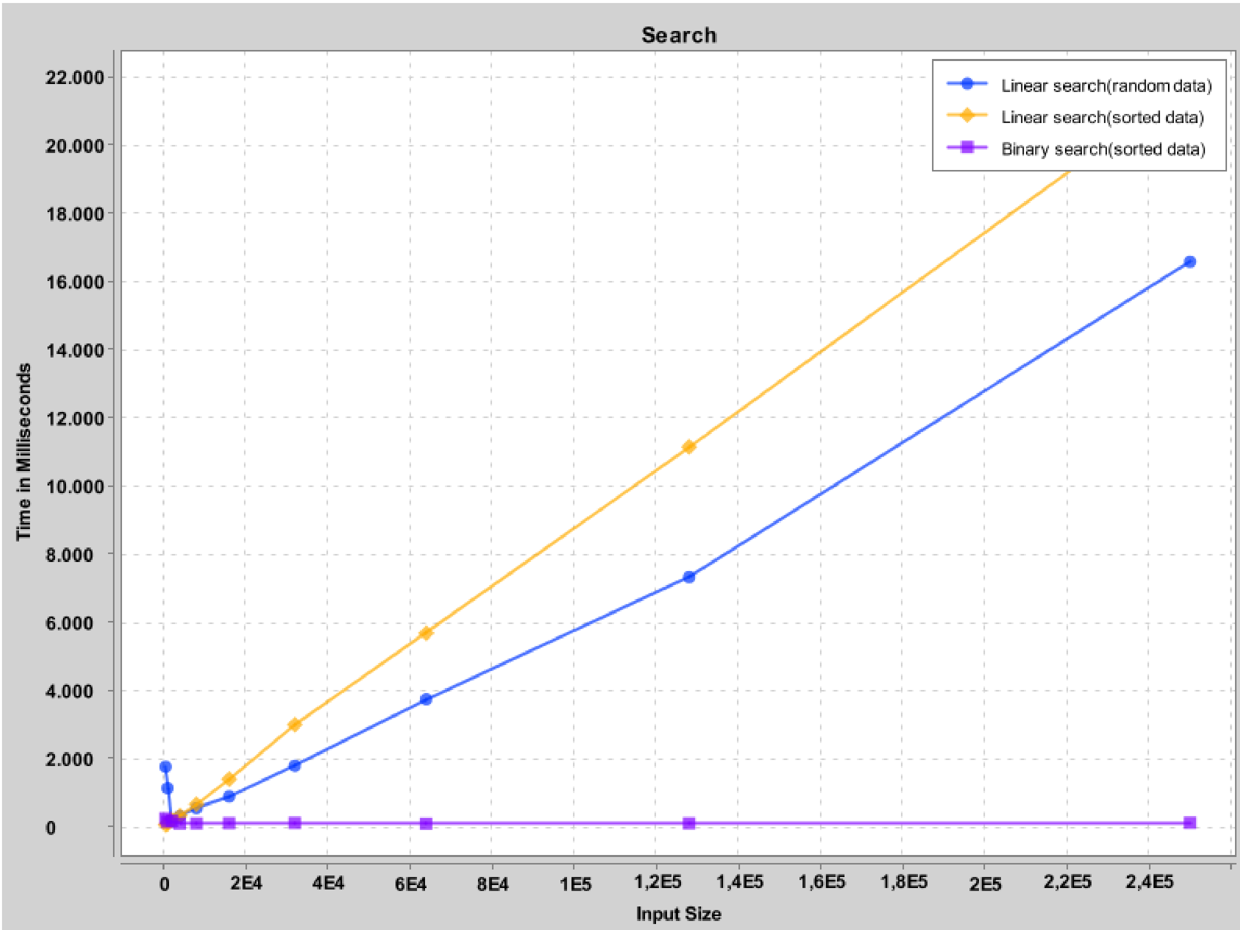

Figure 5: Searching experiments.

4 Notes

For searching algorithms , because of choosing searching element random everytime I run the code the graph has small differences. I add the chart of Random Inputs with (QuickSort and BubbleSort) because they look like same in our graph but in real they have difference.