Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 30, 2023

Student name:
Ataberk ASAR

Student Number:
b2210356135

1 Problem Definition

Aim of this programming assignment is compare and contrast various sorting and searching algorithms. Additionally, theoretical asymptotes of algorithms tested with practical experiment setups.

2 Solution Implementation

2.1 Sorting Algorithms

2.1.1 Selection Sort

```java
package Sorter;

public class SelectionSorter implements Sorter {

    @Override
    public void sort(int[] space) {
        for (int i = 0; i < space.length; i++) {
            int min = i;
            for (int j = i + 1; j < space.length; j++) {
                if (Sorter.lt(space[j], space[min]))
                    min = j;
            }
            Sorter.swap(space, i, min);
        }
    }

}
```

2.1.2 Quick Sort

```java
package Sorter;

public class QuickSorter implements Sorter{

    @Override
    public void sort(int[] space) {
        quickSort(space, 0, space.length - 1);
    }

    private static void quickSort(int[] space, int lo, int hi) {
        int stackSize = hi - lo + 1;
        int[] stack = new int[stackSize];
        int top = -1;

        stack[++top] = lo;
        stack[++top] = hi;

        while (top >= 0) {
            hi = stack[top--];
            lo = stack[top--];
            int pivot = partition(space, lo, hi);

            if (pivot -1 > lo) {
                stack[++top] = lo;
                stack[++top] = pivot - 1;
            }
            if (pivot + 1 < hi) {
                stack[++top] = pivot + 1;
                stack[++top] = hi;
            }
        }

    }

    private static int partition(int[] space, int lo, int hi) {
        int pivot = space[hi];
        int i = lo - 1;

        for(int j = lo; j < hi; ++j)
            if (space[j] <= pivot)
                Sorter.swap(space, ++i, j);
        Sorter.swap(space, ++i, hi);
        return i;
    }

}
```

2.1.3 Bucket Sort

```java
package Sorter;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class BucketSorter implements Sorter{

    @SuppressWarnings("all")
    @Override
    public void sort(int[] space) {
        int bucketNum = (int) Math.ceil(Math.sqrt(space.length));
        List<Integer>[] buckets = new ArrayList[bucketNum];
        for (int i = 0; i < bucketNum; ++i)
            buckets[i] = new ArrayList<>();
        int max = max(space);

        for (int i : space)
            buckets[hash(i, max, bucketNum)].add(i);
        for (List<Integer> bucket : buckets)
            Collections.sort(bucket);

        int i = 0;
        for(List<Integer> bucket : buckets)
            for (int item : bucket)
                space[i++] = item;
    }

    private static int max(int[] space) {
        int max = Integer.MIN_VALUE;
        for (int i : space)
            if (i > max)
                max = i;

        return max;
    }

    private static int hash (int i, int max, int bucketNum) {
        return (int) (1.0 * i / max * (bucketNum - 1));
    }

}
```

2.2 Search Algorithms

2.2.1 Linear Search

```java
package Searcher;

public class LinearSearcher implements Searcher{

    @Override
    public int search(int[] space, int key) {
        for (int i = 0; i < space.length; ++i)
            if (space[i] == key)
                return i;
        return -1;
    }
}
```

2.2.2 Binary Search

```java
package Searcher;

public class BinarySearcher implements Searcher{

    @Override
    public int search(int[] space, int key) {
        int lo = 0;
        int hi = space.length - 1;

        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (space[mid] == key)
                return mid;

            if (space[mid] > key)
                hi = mid - 1;
            else
                lo = mid + 1;
        }

        return -1;
    }
}
```

3 Results, Analysis, Discussion

Your explanations, results, plots go in this section...

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	**2.8**	**3.4**	**11.0**	**47.4**	**152.0**	**434.0**	**1735.2**	**6907.0**	**27889.2**	**110850.8**
Quick sort	**0.0**	**0.0**	**0.2**	**0.2**	**0.8**	**0.8**	**3.0**	**10.2**	**29.8**	**51.8**
Bucket sort	**0.2**	**0.4**	**0.6**	**1.0**	**1.4**	**2.0**	**3.2**	**5.8**	**12.4**	**28.2**
	Sorted Input Data Timing Results in ms									
Selection sort	**0.8**	**1.8**	**7.0**	**58.0**	**125.4**	**447.6**	**1679.4**	**6716.0**	**27571.2**	**109135.8**
Quick sort	**0.2**	**0.2**	**1.4**	**5.8**	**22.0**	**89.0**	**359.8**	**1442.8**	**5728.2**	**21815.2**
Bucket sort	**0.0**	**0.0**	**0.0**	**0.0**	**0.2**	**0.0**	**0.6**	**0.8**	**1.4**	**3.0**
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	**0.6**	**1.6**	**10.0**	**40.6**	**131.8**	**516.4**	**1993.8**	**7966.4**	**32283.4**	**129087.8**
Quick sort	**0.0**	**0.0**	**0.4**	**2.0**	**6.2**	**11.4**	**37.2**	**161.8**	**618.6**	**1308.2**
Bucket sort	**0.0**	**0.0**	**0.2**	**0.0**	**0.0**	**0.2**	**0.6**	**1.2**	**2.2**	**4.8**

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)	**3030.8**	**624.2**	**619.8**	**936.2**	**1551.2**	**2719.6**	**4894.8**	**9964.6**	**17430.6**	**28237.8**
Linear search (sorted data)	**330.4**	**555.4**	**540.0**	**742.6**	**1230.8**	**2392.6**	**3875.4**	**7001.8**	**13903.8**	**26831.4**
Binary search (sorted data)	**2146.4**	**445.2**	**529.8**	**621.8**	**810.6**	**950.6**	**1105.8**	**1142.0**	**1188.6**	**1263.4**

3.1 Analysis of Bucket Sort

Best Case Analysis: Best case for bucket sort algorithm is that given array is already sorted in desired order, where elements inserted into buckets in order and uniformly. Thus there is $\sqrt{n}$ elements, in $\sqrt{n}$ buckets. Max function on line 29 has $\Omega(n)$ time complexity. Note that Java's default sort algorithm on line 21 uses Tim Sort algorithm, which has $\Omega(\sqrt{n})$ time complexity for already sorted array (also its best case). Overall, bucket sort algorithm has $\Omega(n)$ time complexity for best case.
Average Case Analysis: Average case for bucket sort algorithm is that array elements inserted into buckets uniformly. Thus there is $\sqrt{n}$ elements, in $\sqrt{n}$ buckets. Max function on line 29 has $\Theta(n)$ time complexity. Note that Java's default sort algorithm on line 21 uses Tim Sort algorithm, which has $\Theta(\sqrt{n}\log\sqrt{n})$ time complexity for its average case. Overall, bucket sort algorithm has $\Theta(n)$ time complexity for average case.
Worst Case Analysis: Worst case for bucket sort algorithm is that array elements inserted into buckets non uniformly, ie. every element inserted into same bucket. Max function on line 29 has $O(n)$ worst time complexity. Note that Java's default sort algorithm on line 21 uses Tim Sort algorithm, which has $O(n\log n)$ time complexity for its worst case. Overall, bucket sort algorithm has $O(n\log n)$ time complexity for average case.
Auxiliary Space Complexity Analysis The space complexity of bucket sort is $O(n)$, since every element also stored into buckets.

3.2 Analysis of Linear Search

Best Case Analysis: Best case for linear search algorithm is that search element is in the first index. Thus, linear search algorithm has $\Omega(1)$ time complexity for best case.
Average Case Analysis: An item can be in n + 1 different places (n different index, and not present at all). Thus, average number of comparisons = $(\sum x)/(n+1) = (n*(n+1))/2(n+1) = n/2$. Overall, linear search algorithm has $\Theta(n)$ time complexity for average case.
Worst Case Analysis: Worst case for linear search algorithm is that search element is in the last index. Thus, linear search algorithm has $\Omega(n)$ time complexity for worst case.
Auxiliary Space Complexity Analysis The space complexity of linear search is $O(1)$, since no extra space is used.

3.3 Analysis of Binary Search

Best Case Analysis: Best case for binary search algorithm is that search element is in the middle index. Thus, binary search algorithm has $\Omega(1)$ time complexity for best case.

Average Case Analysis: An item can be in n + 1 different places (n different index, and not present at all). Also, number of comparisons can be at most $\log n$ for a search item, where 1 element requires 1 comparison (middle element), 2 elements require 2 comparisons (elements at n/4 and 3n/4), 4 elements require 3 comparisons, ..., $2^{\log n-1}$ elements require $\log n$ comparisons. Thus, average number of comparisons $(\sum_{x=1}^{\log n} 2^{x-1} * x)/(n+1) = (2^{\log n} * \log(n-1)+1)/(n+1) < (n * \log(n-1)+1)/(n+1) = \log n$. Thus, binary search algorithm has $\Theta(\log n)$ time complexity for average case.

Worst Case Analysis: Worst case for linear search algorithm is that search element is in the last index that algorithm will look. Thus, binary search algorithm has $\Omega(\log n)$ time complexity for worst case.

Auxiliary Space Complexity Analysis The space complexity of binary search is $O(1)$, since no extra space is used.

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	$\Omega(n)$	$\Theta(n)$	$O(n \log n)$
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**
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	$O(n)$
Linear Search	$O(1)$
Binary Search	$O(1)$

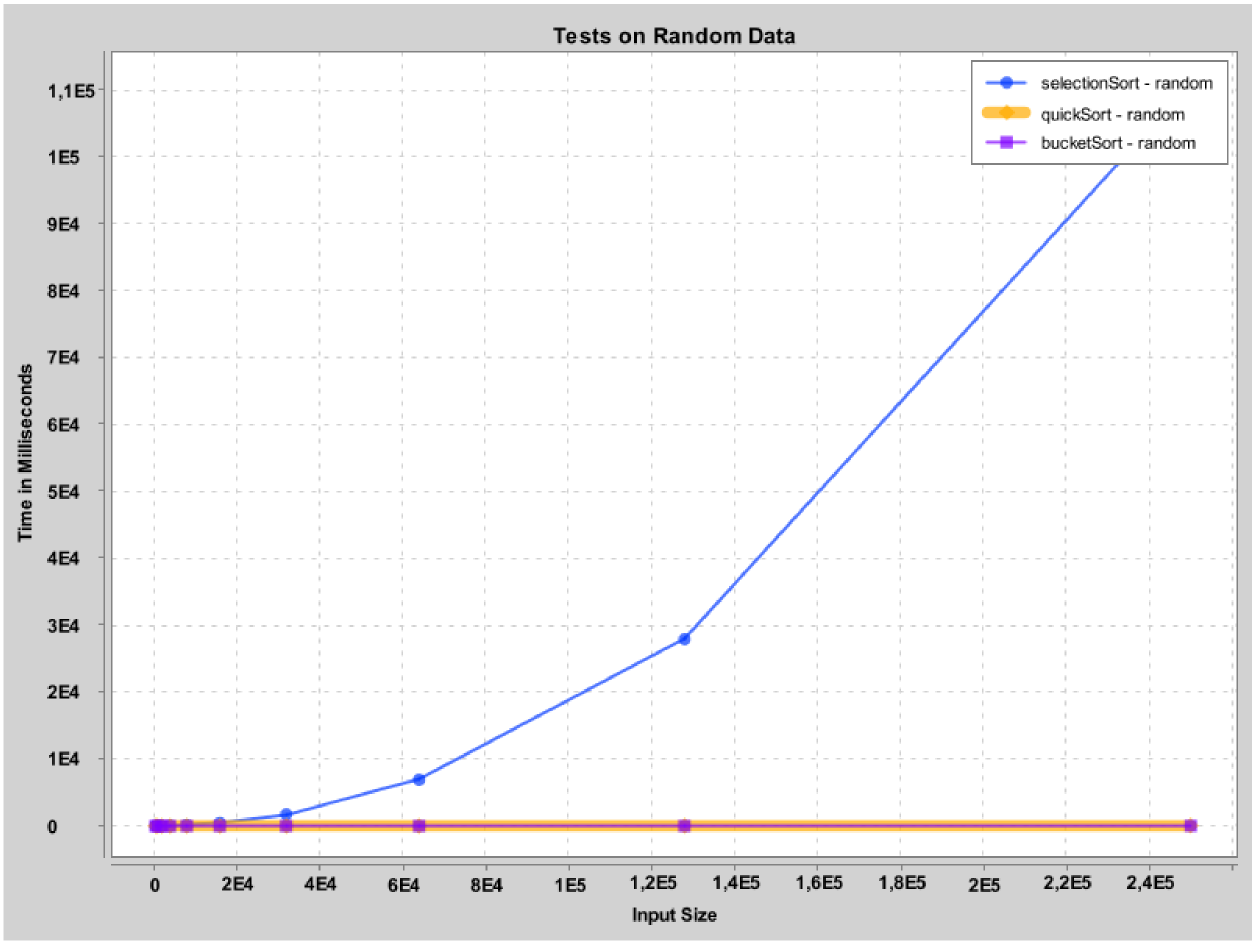

Figure 1: Plot of sorting on random array.

Experiment results supports theoretical complexity analyses. This experiment setup can be considered as average case for sorting algorithms.

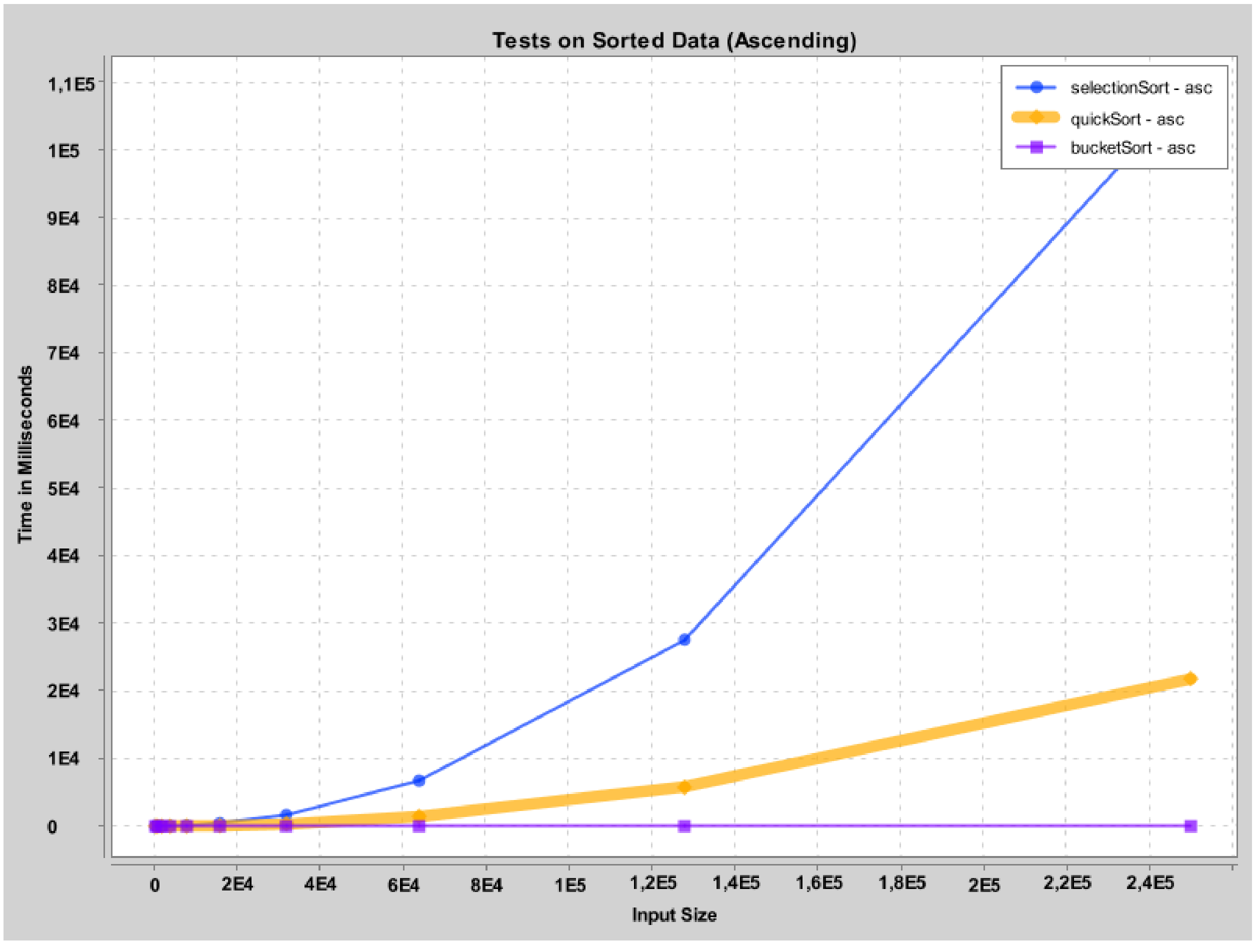

Figure 2: Plot of sorting on ascending array.

Experiment results supports theoretical complexity analyses. Note that this experiment setup considered as worst case for quick sort, and best case for bucket sort.

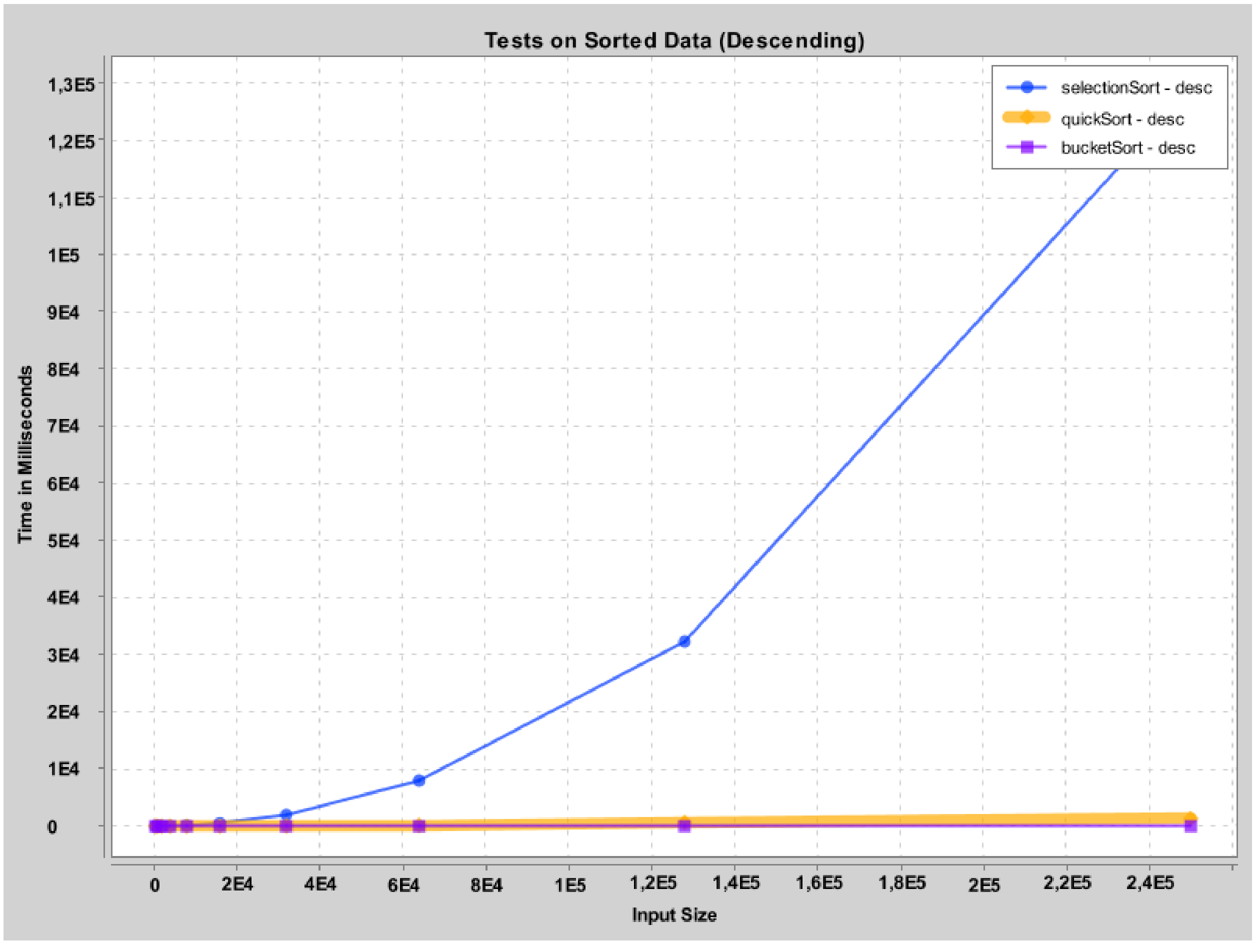

Figure 3: Plot of sorting on descending array.

Experiment results supports theoretical complexity analyses.

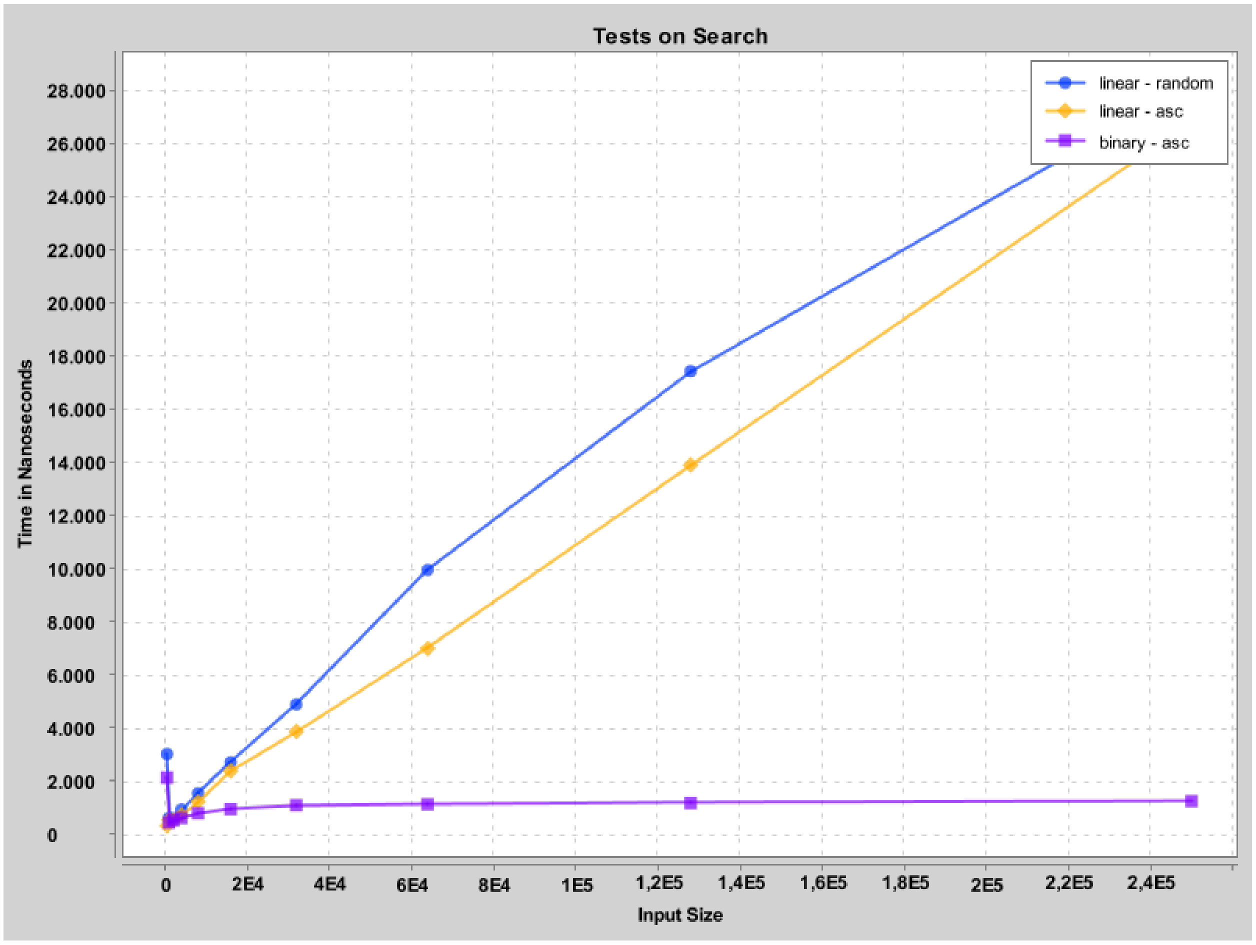

Figure 4: Plot of searching algorithms.

Experiment results mostly supports theoretical complexity analyses. For linear search on random array and binary search on sorted array, there are unusual peaks for small number of input sizes. Issue assumed to be caused by compulsory cache misses. Experiment number increased (1000 for default experiment) in order to solve compulsory cache misses and experiments repeated, but problem seems to persists.

4 Notes

I divided search and sort algorithm classes into packages. Thus compilation of the project should be implemented considering them.
Ex:
Compilation:

```
javac -cp *.jar Searcher\*.java Sorter\*.java *.java -d .
```

Execution:

```
java -cp .;* Main TrafficFlowDataset.csv
```