Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Abdulkadir Parlak

Student Number:
b2210765025

1 Problem Definition

In this assignment, we are asked to analyze some of the basic sorting and searching algorithms such as merge sort, insertion sort or binary search. Also we compare their running times on a number of inputs with changing sizes and record the results. Finally we find out their time and auxiliary space complexities and graph the results at the end.

2 Solution Implementation

2.1 Insertion Sort

Insertion Sort implemented in Java11:

```
1  public static void insertionSort(int[] arr){
2      for (int i = 1; i < arr.length; i++) {
3          int keyValue = arr[i];
4          int j = i - 1;
5          while(j >= 0 && arr[j] > keyValue){
6              arr[j + 1] = arr[j];
7              j--;
8          }
9          arr[j + 1] = keyValue;
10     }
11 }
```

2.2 Merge Sort

Merge Sort implemented in Java11:

```
12 public static int[] mergeSort(int[] arr){
13     int n = arr.length;
14     if(n <= 1){
15         return arr;
16     }
17     int midIndex = n / 2;
18     int[] leftArray = new int[midIndex];
19     int[] rightArray = new int[n - midIndex];
20
21     for (int i = 0; i < midIndex; i++) {
22         leftArray[i] = arr[i];
23     }
24     for(int i = midIndex; i < n; i++){
25         rightArray[i - midIndex] = arr[i];
26     }
27
28     leftArray = mergeSort(leftArray);
29     rightArray = mergeSort(rightArray);
```

```
30
31      return mergeArrays(leftArray, rightArray);
32  }
33  private static int[] mergeArrays(int[] leftArray, int[] rightArray){
34      int leftSize = leftArray.length;
35      int rightSize = rightArray.length;
36
37      int[] finalArr = new int[leftSize + rightSize];
38
39      int i = 0, j = 0, k = 0;
40      while(i < leftSize && j < rightSize){
41          if (leftArray[i] <= rightArray[j]) {
42              finalArr[k] = leftArray[i];
43              i++;
44          }
45          else {
46              finalArr[k] = rightArray[j];
47              j++;
48          }
49          k++;
50      }
51      while(i < leftSize){
52          finalArr[k] = leftArray[i];
53          i++;
54          k++;
55      }
56      while(j < rightSize){
57          finalArr[k] = rightArray[j];
58          j++;
59          k++;
60      }
61      return finalArr;
62  }
```

2.3 Counting Sort

Counting Sort implemented in Java11:

```
public static int[] countingSort(int[] input) {
    int k = getMax(input, input.length);

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

    for (int i = 0; i < input.length; i++) {
        int j = input[i];
        count[j]++;
    }
    for (int i = 1; i <= k; i++) {
        count[i] += count[i - 1];
    }
    for (int i = input.length - 1; i >= 0; i--) {
        int j = input[i];
        count[j]--;
        output[count[j]] = input[i];
    }
    return output;
}
private static int getMax(int arr[], int n)
{
    int res = arr[0];
    for (int i = 1; i < n; i++)
        res = Math.max(res, arr[i]);
    return res;
}
```

2.4 Linear Search

Linear Search implemented in Java11:

```
public static int linearSearch(int[] input, int key){
    for (int i = 0; i < input.length; i++) {
        if(input[i] == key){
            return i;
        }
    }
    return -1;
}
```

2.5 Binary Search

Binary Search implemented in Java11:

```
public static int binarySearch(int[] sortedArr, int key){
    int left = 0;
    int right = sortedArr.length - 1;
    while(left <= right){
        int mid = (left + right) / 2;
        if(sortedArr[mid] == key){
            return mid;
        }
        if(sortedArr[mid] < key){
            left = mid + 1;
        }
        else{
            right = mid - 1;
        }
    }
    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									
Insertion sort	0.1	0.2	0.2	0.9	1.6	1.5	5.5	22.1	85.1	349.4
Merge sort	0.2	0.1	0.1	0.3	0.7	1.1	2.6	4.7	9.8	19.9
Counting sort	129.8	91.9	91.9	91.9	91.6	92.0	92.0	92.4	92.8	98.3
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.1	0.3
Merge sort	0.0	0.0	0.0	0.1	0.3	0.5	1.1	2.1	5.1	8.2
Counting sort	92.4	92.1	92.0	91.8	92.7	92.0	92.7	92.6	93.0	93.8
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.1	0.0	0.2	0.7	2.5	9.9	40.7	160.3	611.5
Merge sort	0.0	0.0	0.1	0.1	0.3	0.5	1.1	1.9	3.8	8.3
Counting sort	109.8	92.2	92.3	92.0	92.4	92.2	92.4	93.1	93.5	93.8

As seen on the graph above, insertion sort performs quite well almost better than merge sort for small-sized datasets. But, when the dataset size increases, the merge sort outperforms the insertion sort. Since insertion sort has $O(n^2)$ time complexity it is not a suitable choice if you have big datasets in question. On the other hand, counting sort performs similar results as the dataset gets larger, since it highly depends on the range of the elements in the dataset. In the

sorted case, insertion sort performs almost instant. Because there is not any inversion in the data. Yet, merge sort performed similar to the randomly distributed case. Because merge sort does not have anything to do with the order of the elements. It will continue to divide-conquer the problem. Since counting sort is a non-comparison based sort algorithm, it does not get affected by the order of the elements in the dataset.

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)	2000.0	1000.0	1000.0	1000.0	1000.0	1000.0	3000.0	4000.0	7000.0	13000.0
Linear search (sorted data)	0.0	0.0	0.0	0.0	1000.0	2000.0	2000.0	6000.0	11000.0	21000.0
Binary search (sorted data)	1000.0	1000.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Linear search is basically iterating over the elements one by one regardless of the situation. So, it took similar time to apply linear search on a random data or sorted data. The fluctuations on the table is caused by the random key selection. So, in each test there is a random key is selected from the dataset and the tests are repeated 1000 times. There may be some lucky selections caused larger dataset searchs finished earlier than the small dataset searchs. In the case of binary search, because of the characteristics of the binary search ($O(\log n)$ time complexity), almost every search ended nearly instant. The zeros in the table means "the search ended quite fast (almost instant)".

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

Insertion sort has variable performance depending on the initial order of elements (best case is the sorted case), merge sort consistently performs efficiently across all cases (no matter how the data is distributed), and counting sort is highly efficient when the range of input elements is small. In case of search algorithms, linear search is basically iterating over the data one by one from the beginning till the end. So, it has O(n) time complexity on average case. However, the binary search is quite efficient than the linear search since it cuts the problem into half at each iteration. This operation makes the time complexity O(logn) on average case.

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	$O(1)$

Insertion sort is an in-place sorting algorithm so it doesn't need additional space. However, the merge sort is not in-place, it needs additional space in order to store the sorted elements before merging them back into the original array (line 37). The total space complexity of merge sort is the sum of the space required for the recursive stack (line 18) and the temporary storage for merging. Counting sort creates an auxiliary array of size k+1, see the line 66, to store the counts of each element, and it also requires an output array of the same size as the input array to store the sorted elements (line 67). Search algorithms do not require any additional space.

Here are some plots that are obtained by analysis:

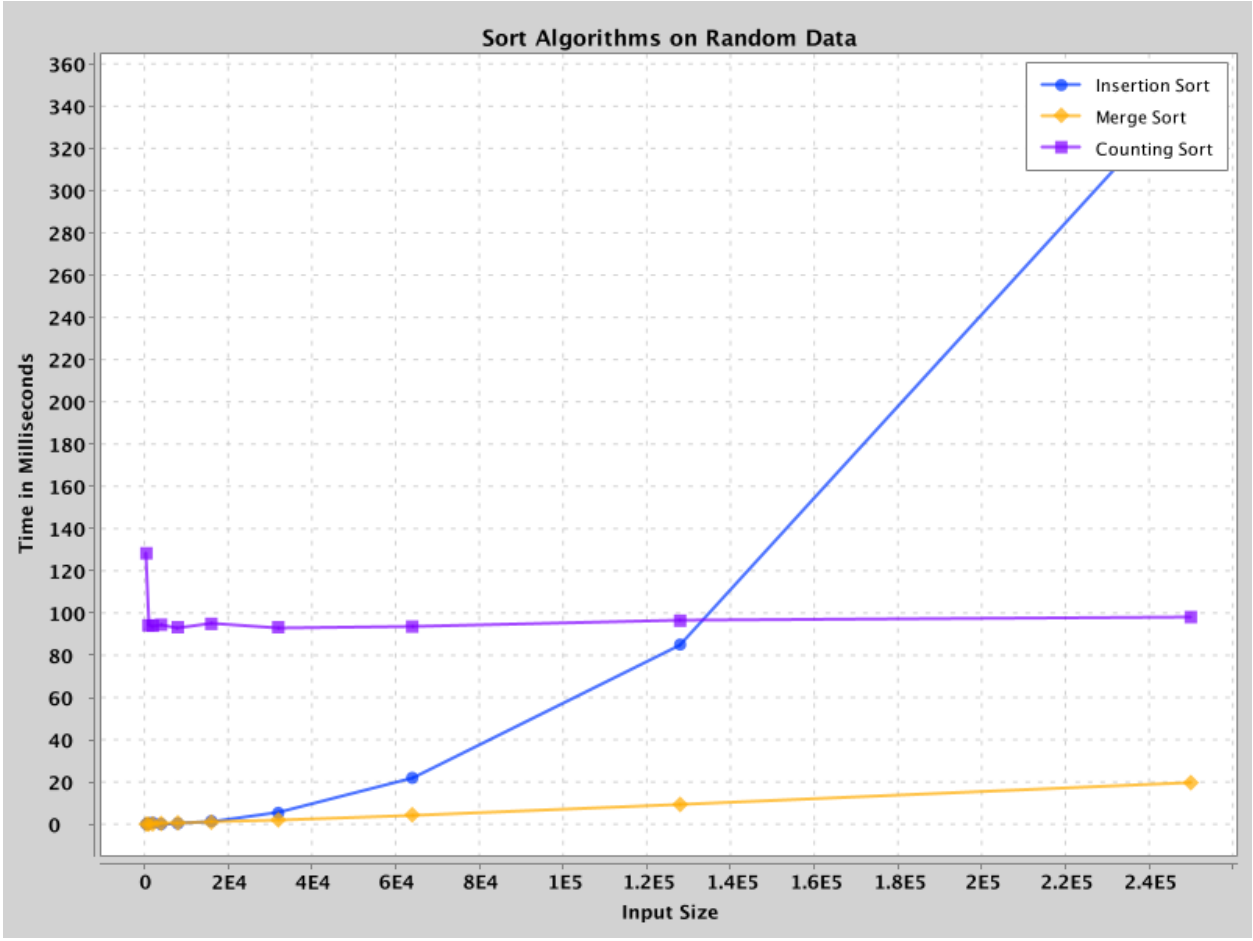

Figure 1: Sort Algorithms Performance on Randomly Distributed Data

On randomly distributed dataset, the insertion sort performed exponential, merge sort performed quite well, $O(n \log n)$ time complexity is observed. And since the counting sort is a non-comparison based algorithm, it performed linear and the dataset size didn't effect its performance.

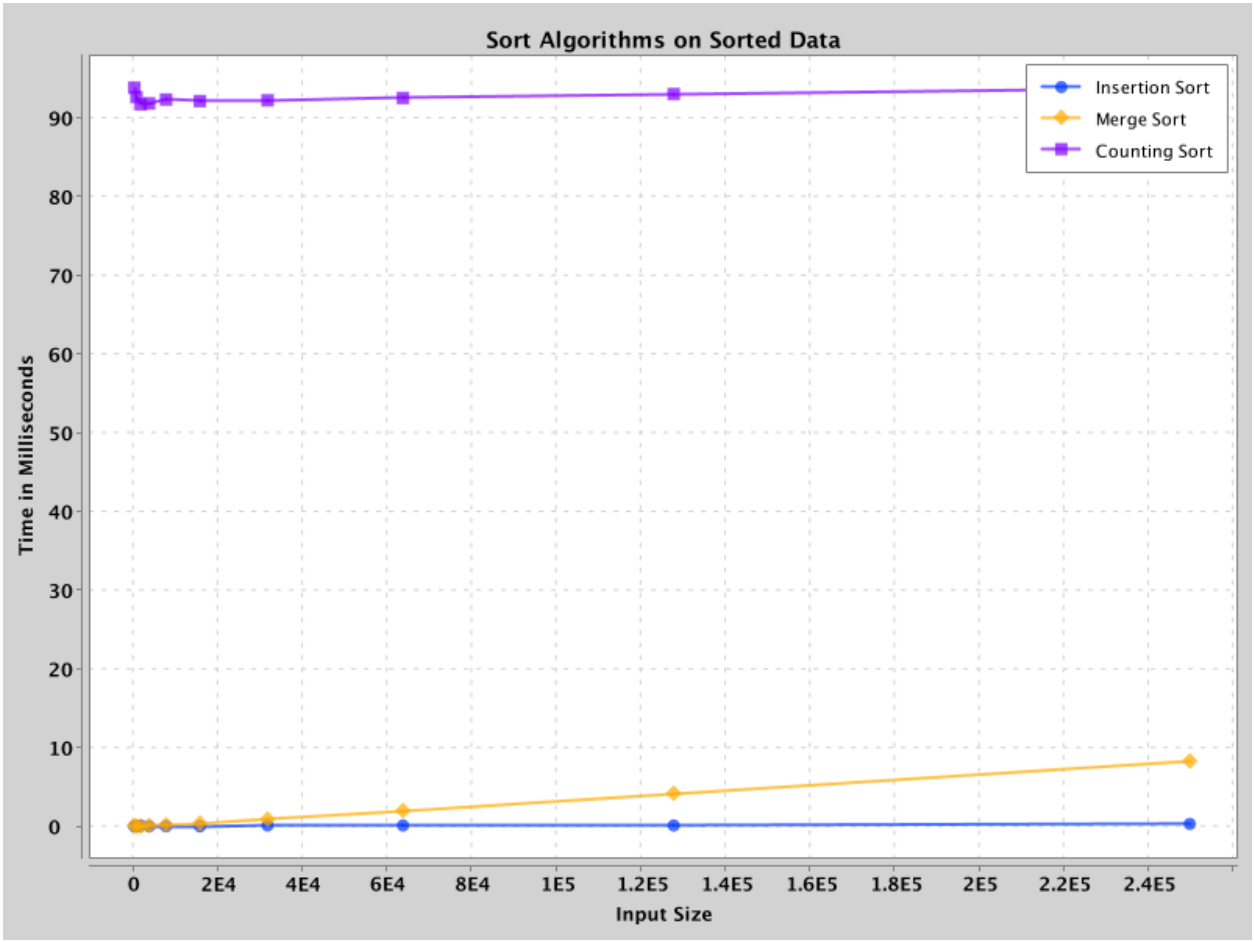

Figure 2: Sort Algorithms Performance on Sorted Data

Here the dataset is sorted. So, the insertion sort complexity became linear $O(n)$. Merge sort and counting sort are not affected by the order of the elements.

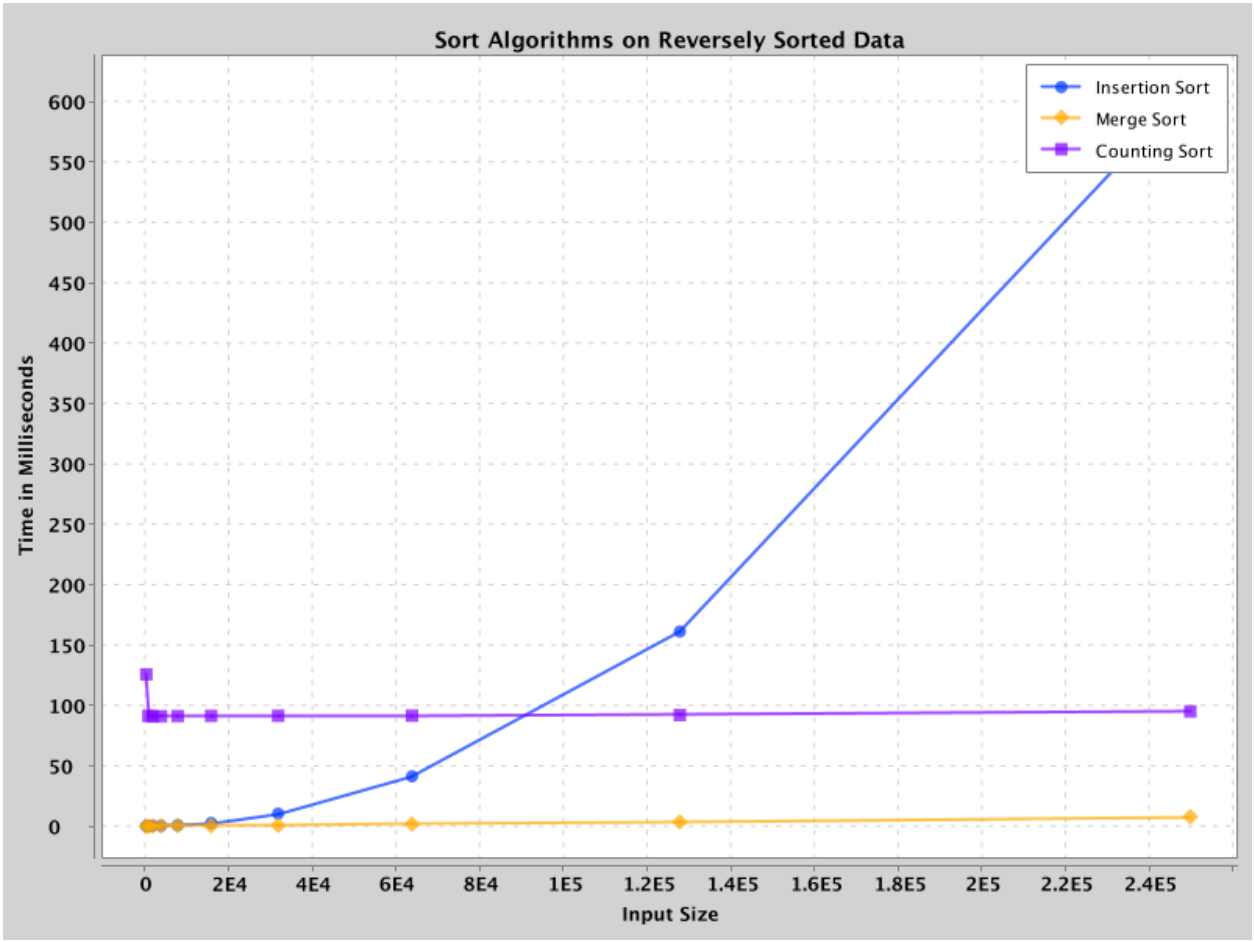

Figure 3: Sort Algorithms Performance on Reversely Sorted Data

In case of reversely sorted dataset, the insertion sort performed worse than the randomly distributed case. Because each element pair is inversion and they should be moved to the beginning of the dataset. Merge sort and counting sort are not affected again.

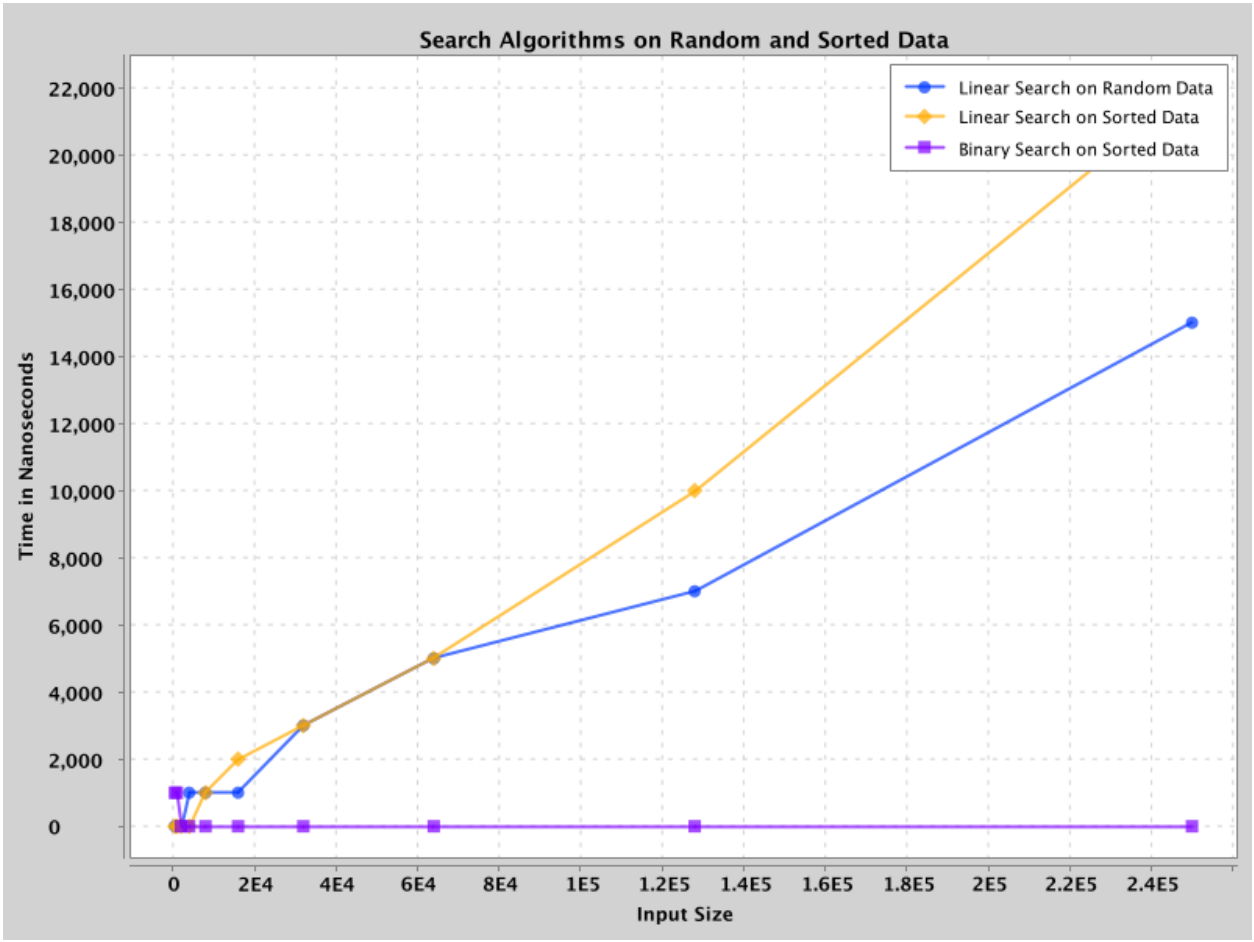

Figure 4: Search Algorithms Performance on Random and Sorted Data

Linear search is not affected by the order of the dataset. Binary search is quite fast as seen in the graph, almost instant.

4 Notes

The sort algorithms are tested 10 times in order to get optimal results. The search algorithms are tested 1000 times in order to get optimal results. At each iteration, a randomly selected element in the dataset is obtained. In this way, the effects of selecting a lucky element and misleading search time analysis is prevented.