Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Yüksel Çağrı ARSLAN

Student Number:
b2210765031

1 Problem Definition

The goal of this assignment is to demonstrate how the time it takes for different algorithms to run relates to their theoretical complexity. You'll need to implement these algorithms using the provided pseudocode and then run experiments on specific datasets. The aim is to show that the actual running times match up with the expected growth rates of the algorithms. To achieve this, you'll need to figure out ways to minimize any randomness in your timing measurements. Finally, you'll plot your results to visually display and analyze how the algorithms' complexities behave.

2 Solution Implementation

This solution aims to compare how fast different ways of sorting and finding things work. It uses graphs made with a tool called XChart. First, it opens a file with data and reads the part we want into lists. Then, it uses special ways of sorting and finding things, like putting numbers in order or searching for a specific number. These methods are written in Java based on example instructions.

During testing, it tries out these methods many times using loops, making sure not to repeat unnecessary steps. It records how long each method takes to run and stores these times in a special table for making graphs with XChart.

The sorting methods used are insertion sort, merge sort, and counting sort, each following their own set of rules for how fast they should be. For finding things, it uses linear search (looking at each item one by one) and binary search (cutting the search area in half each time).

The results from these tests are used to make graphs showing how fast each method works in different situations, like when the data is random, already sorted, or reversed. These graphs help to see which method is faster and how well it handles different types of data. This helps to understand and compare the performance of each method.

2.1 Sorting Algorithm 1 : Insertion sort

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

2.2 Sorting Algorithm 2 : Merge sort

```
12 public static Integer[] mergeSort(Integer[] a) {
13         int n = a.length;
14         if (n <= 1) {
15             return a;
16         }
17         Integer[] left = Arrays.copyOfRange(a, 0, n / 2);
18         Integer[] right = Arrays.copyOfRange(a, n / 2, n);
19 
20         left = mergeSort(left);
21         right = mergeSort(right);
22 
23         return mergeSort(left, right);
24 
25     }
26 
27     public static Integer[] mergeSort(Integer[] A, Integer[] B) {
28         Integer[] C = new Integer[A.length + B.length];
29         int i = 0, j = 0, k = 0;
30 
31         while (i < A.length && j < B.length) {
32             if (A[i] > B[j]) {
33                 C[k++] = B[j++];
34             } else {
35                 C[k++] = A[i++];
36             }
37         }
38 
39         while (i < A.length) {
40             C[k++] = A[i++];
41         }
42         while (j < B.length) {
43             C[k++] = B[j++];
44         }
45 
46         return C;
47     }
```

2.3 Sorting Algorithm 3 : Counting sort

```
48 public static int[] countingSort(Integer[] a, Integer k) {
49         int[] count = new int[k + 1];
50         int[] output = new int[a.length];
51         int size = a.length;
52 
53         for (int i = 0; i < size; i++) {
54             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];
            count[j]--;
            output[count[j]] = a[i];
        }
        return output;

    }
```

2.4 Searching Algorithm 1 : Linear Search

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

2.5 Searching Algorithm 2 : Binary Search

```
public static int binarySearch(Integer[] a, Integer number) {
        int low = 0;
        int high = a.length - 1;
        while (high - low > 1){
            int mid = (high + low) / 2;

            if (a[mid] < number) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        if (a[low] == number){
            return low;
        } else if (a[high] == number){
```

```
94              return high;
95          }
96          return -1;
97      }
```

3 Results, Analysis, Discussion

A set of time measurements is given for three sorting algorithms and searching algorithms that are being worked on. These measurements cover various sizes of input data. 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	0.9	0.6	0.6	2.4	9.5	43.4	167.7	684.5	3090.1	14305.8
Merge sort	0.6	0.7	0.2	0.4	1.1	2.6	5.4	8.6	17.9	34.4
Counting sort	216.7	113.8	113.7	113.8	113.9	113.9	114.4	114.7	116.2	118.0
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.4	0.6
Merge sort	0.1	0.0	0.1	0.2	0.4	0.9	1.7	3.1	16.8	17.3
Counting sort	0.1	0.0	0.0	0.0	0.1	0.1	0.2	0.5	1.1	116.5
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.1	0.4	1.2	5.6	19.4	78.1	307.3	1243.3	5005.5	19509.5
Merge sort	0.1	0.1	0.1	0.4	0.7	1.4	3.1	9.2	19.3	21.6
Counting sort	114.9	113.7	113.5	113.6	113.9	114.0	114.5	115.4	116.5	117.2

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)	8152.246	3039.276	1687.883	914.612	1747.989	3430.278	6740.196	13017.101	25326.67	51152.331
Linear search (sorted data)	99.442	103.993	129.663	1222.828	1672.175	3406.641	4685.929	9260.507	23562.791	52342.97
Binary search (sorted data)	508.41	91.843	105.594	105.67	311.012	352.065	603.878	1975.288	944.599	1257.825

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

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	$O(1)$

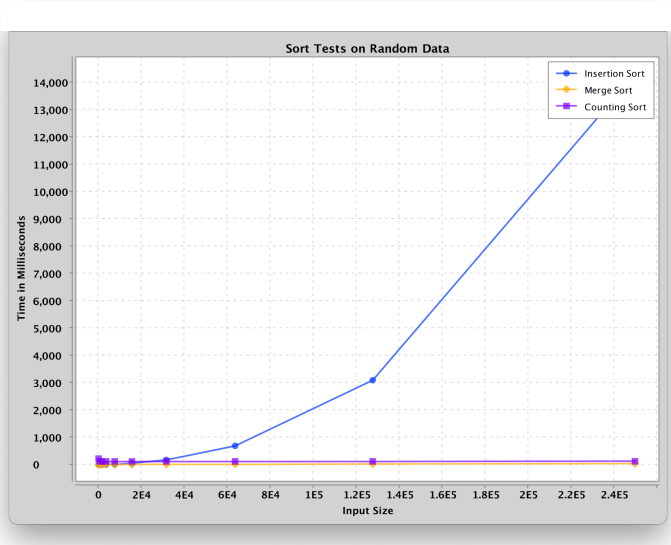

Figure 1: Plot of the Sort Tests on Random Data.

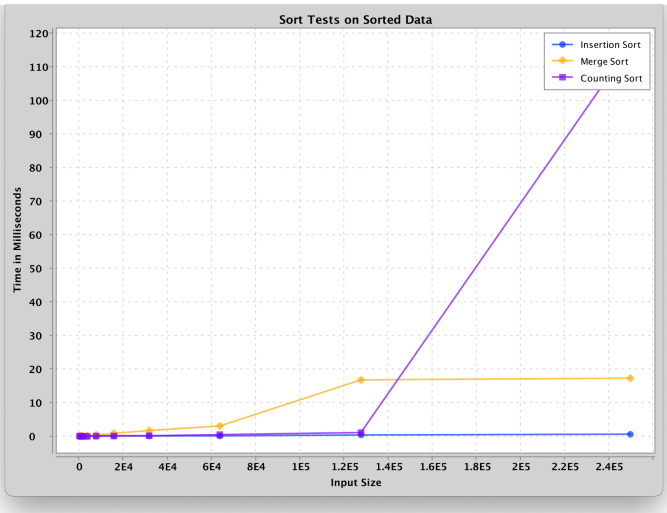

Figure 2: Plot of the Sort Tests on Sorted Data.

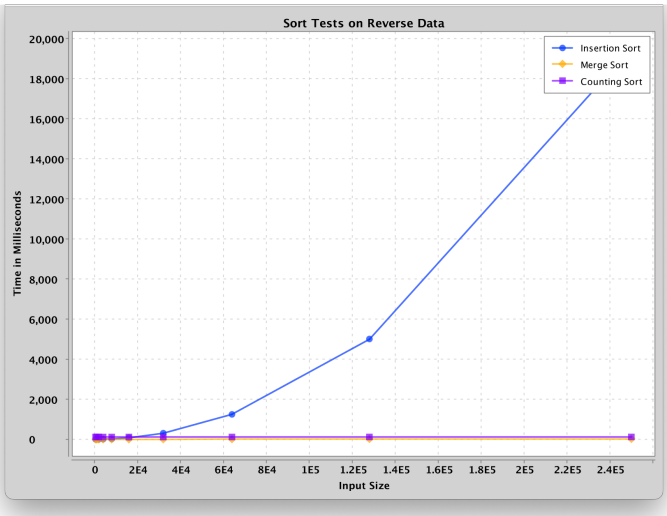

Figure 3: Plot of the Sort Tests on Reverse Data.

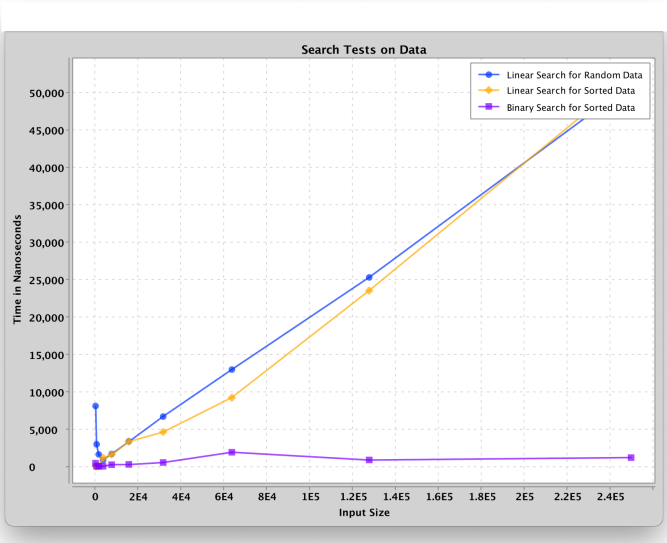

Figure 4: Plot of the Search Tests on Data.

4 Notes

In summary, looking at graphs of how well sorting algorithms work gives us a good idea of how efficient they are. Algorithms like Merge Sort, Insertion Sort, and Counting Sort show different behaviors in these graphs. Merge Sort usually works consistently well with a time complexity of O(nlogn), as we see in our graphs. Insertion Sort, though easy to understand, tends to slow down a lot with bigger sets of data, showing a time complexity of O(n2) in our graphs. Counting Sort, on the other hand, works efficiently, especially when dealing with data that has a limited range of values.

By studying the shapes of these graphs, we can see clear patterns and decide which sorting method works best for different tasks and sizes of data. So, combining the code we write with these visual graphs helps us understand how well Merge Sort, Insertion Sort, and Counting Sort perform, which helps us make better choices when picking a sorting method for our needs.

References

- https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/counting-sort/

- https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/