Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 21, 2024

Student name:
Berke TAŞKIN

Student Number:
b2210356045

1 Problem Statement

We use many algorithms for our needs while programming. It's important to know the gives and takes for each particular algorithm. Therefore, analyzing an algorithm is the basis step in order to reach a better understanding about the performance of your program. Memory space, disk space and elapsed time are three must-known properties of an algorithm. In this Assignment, we are going to analyze these properties for certain sorting and searching algorithms. At the end, we will have a clue about the efficiencies of our algorithms and in which situation to use each one of them.

2 Sort and Search Algorithms

These algorithms are written in Java.

2.1 Sort Algorithms

2.1.1 Insertion Sort Algorithm

```
1  public class InsertionSort {
2      public static int[] sort(int A[]){
3          int i, key;
4
5          for (int j = 1; j < A.length; j++) {
6              key = A[j];
7              i = j-1;
8              while (i >= 0 && A[i] > key){
9                  A[i+1] = A[i];
10                 i = i-1;
11             }
12             A[i+1] = key;
13         }
14
15         return A;
16     }
17 }
```

2.1.2 Merge Sort Algorithm

```
18 public class MergeSort {
19     public static int[] sort(int A[]){
20         int n = A.length;
21         if (n <= 1){
22             return A;
23         }
24 
25         int[] left = new int[n/2];
26         int[] right = new int[n-(n/2)];
27 
28         for (int i = 0; i < n/2; i++){
29             left[i] = A[i];
30         }
31         for (int i = 0; i < n-(n/2); i++){
32             right[i] = A[n/2 + i];
33         }
34 
35         left = sort(left);
36         right = sort(right);
37 
38         return merge(left,right);
39     }
40 
41     public static int[] merge(int A[], int B[]){
42         int[] C = new int[A.length+B.length];
43         int i = 0, j = 0, k = 0;
44 
45         while (i < A.length && j < B.length){
46             if (A[i] > B[j]){
47                 C[k++] = B[j++];
48             } else {
49                 C[k++] = A[i++];
50             }
51         }
52 
53         while (i < A.length) {
54             C[k++] = A[i++];
55         }
56         while (j < B.length) {
57             C[k++] = B[j++];
58         }
59 
60         return C;
61     }
62 }
```

2.1.3 Counting Sort Algorithm

```
63 public class CountingSort {
64     public static int[] sort(int A[]){
65         int size = A.length;
66         int[] output = new int[A.length];
67         int k = 0;
68         for (int i = 0; i < size; i++){
69             if (A[i] > k){
70                 k = A[i];
71             }
72         }
73         int[] count = new int[k+1];
74 
75         for (int i = 0; i < size; i++){
76             int j = A[i];
77             count[j] = count[j]+1;
78         }
79         for (int i = 1; i < k+1; i++){
80             count[i] = count[i] + count[i-1];
81         }
82         for (int i = size-1; i >= 0; i--){
83             int j = A[i];
84             count[j] = count[j]-1;
85             output[count[j]] = A[i];
86         }
87 
88         return output;
89     }
90 }
```

2.2 Search Algorithms

2.2.1 Linear Search Algorithm

```
91  public class LinearSearch {
92      public static int search(int A[], int x){
93          int size = A.length;
94
95          for (int i = 0; i < size; i++){
96              if (A[i] == x){
97                  return i;
98              }
99          }
100
101         return -1;
102     }
103 }
```

2.2.2 Binary Search Algorithm

```
104 public class BinarySearch {
105     public static int search(int A[], int x){
106         int low = 0;
107         int high = A.length-1;
108
109         while (high-low > 1){
110             int mid = (high+low)/2;
111             if (A[mid] < x){
112                 low = mid+1;
113             } else {
114                 high = mid;
115             }
116         }
117
118         if (A[low] == x){
119             return low;
120         }
121
122         else if (A[high] == x){
123             return high;
124         }
125
126         return -1;
127     }
128 }
```

3 Running Time Results

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	0.2	0.1	0.3	1.8	6.8	27.4	110	439	1845	6835
Merge sort	0.1	0.1	0.3	0.3	0.3	1.1	2.1	4.3	8.9	19.2
Counting sort	217	142	133	135	132	132	132	134	152	148
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0.1	0.5
Merge sort	0.1	0	0.1	0.5	0.6	1	2.1	4.5	9.6	21.2
Counting sort	213	142	132	132	132	131	131	134	151	158
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.2	0.3	0.9	3.5	13.7	54.5	218	876	3690	13686
Merge sort	0	0	0.1	0.1	0.5	1.1	2.1	4.3	9.3	21.2
Counting sort	201	135	133	132	133	133	133	134	145	161

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)	921	783	308	419	1228	667	359	1235	10860	56073
Linear search (sorted data)	495	653	94	270	1021	3563	1976	2743	17064	65727
Binary search (sorted data)	879	131	141	616	168	89	88	85	104	75

4 Complexity Analysis

Complexity Analysis Tables (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	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Counting Sort: The best case is when the array is already sorted, the average case is when the numbers in array are mixed randomly, and the worst case is when the array is reversely sorted. For each of these cases, Counting Sort should have a time complexity of $O(n + k)$. If the array length (n) or maximum number in that array (k) are very high, that also makes running time that much higher. This means both n and k are equally important for the time complexity of this sort. In some situations, if k is hugely greater than n, it is hard to analyze the effect of n for counting sort.

Linear Search: The best case is when we search the first item in array, and the worst case is when we search the last item in array. Other item positions are mostly in average case. For the best case scenario, we don't have to traverse the entire array as it's the first item when we enter the array. Therefore it has $\Omega(1)$ time complexity. Oppositely, we have to traverse the entire array for the worst case scenario. Running time is depending on the length of array, thus it has $O(n)$ time complexity. In most of the scenarios, we are searching a number in the middle of array. Since middle part is also depending on the full length of array, average case should also have $\Theta(n)$ time complexity.

Binary Search: The best case is when we search the middle item in array, and the worst case is when we search the first or last item in array. If the item we are searching is exactly in the middle, it has only $\Omega(1)$ time complexity as it's the first item we see when traversing the array. For the worst case scenario, we have to traverse the array until we have only one element left. Since we search in binary (halving the array in each time) and not linearly, it only has $O(\log n)$ time complexity. In most of the scenarios, our item is not in the middle and we have to half the array each time until we find it. This makes average case also have a $\Theta(\log n)$ time complexity.

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	$O(1)$

Counting Sort: We are creating two new arrays (which are named count and output in the algorithm) with size of array length (n) and maximum number (k). Therefore, auxiliary space complexity becomes $O(n + k)$ for Counting Sort.

Linear Search: We are not creating a new array in the algorithm, so Linear Search has auxiliary space complexity of $O(1)$.

Binary Search: We are not creating a new array in the algorithm, so Binary Search has auxiliary space complexity of $O(1)$.

5 Plotting Graphs

5.1 Tests on Random Data

Fig. 1.

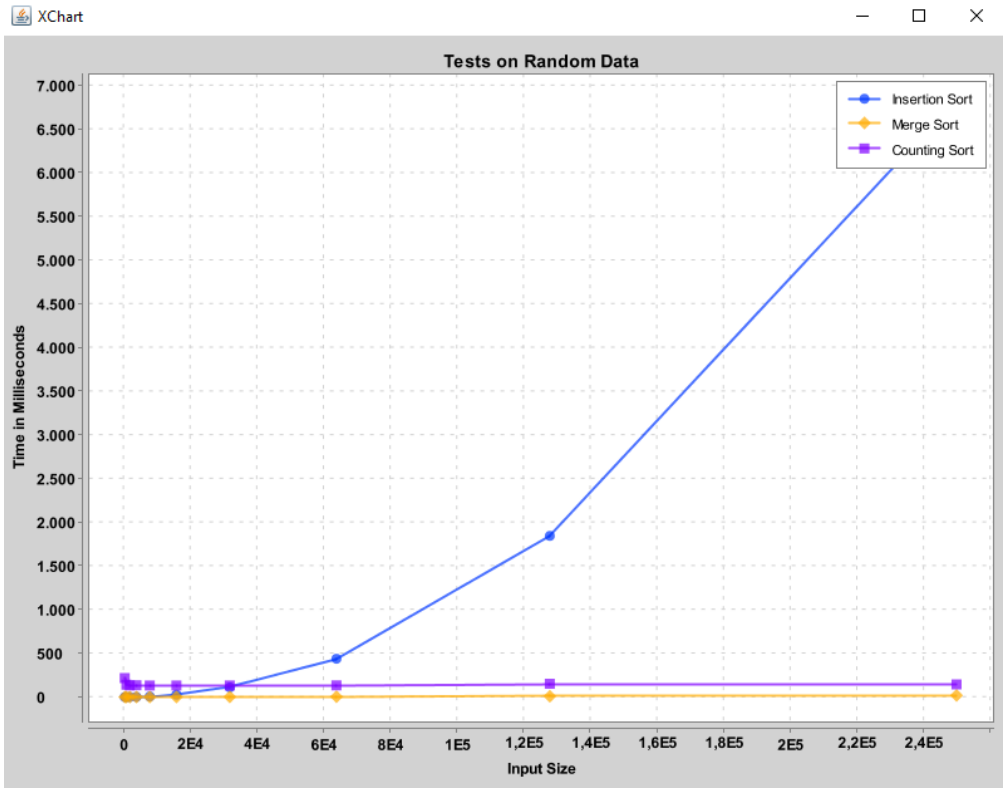

Figure 1: Plot of the Sorting Algorithms (Random Data)

These results of random data are mostly the average case. When we look at the graph, we see that Insertion Sort has much higher running time than other two sorting algorithms. This is normal as it should have $O(n^2)$ time complexity on paper. So our results seems true for this test. Also, when we look at Merge Sort and Counting Sort on graph, we see that they are pretty close to each other. But there is a small difference between them. Merge Sort starts with 0 ms and then doubles its time for each set. On the other hand, Counting Sort always stays the same around 100-200 ms. Merge Sort never crosses Counting, and the reason for that is k value in Counting Sort. Since its maximum number in each array is a huge number, its time is also significantly huge than usual. This also shows us that we cannot reach certain conclusions about which sorting algorithm works better than the others for every situation. But in this situation, we can easily say that Merge Sort is the best sorting algorithm for average case scenarios.

5.2 Tests on Sorted Data

Fig. 2.

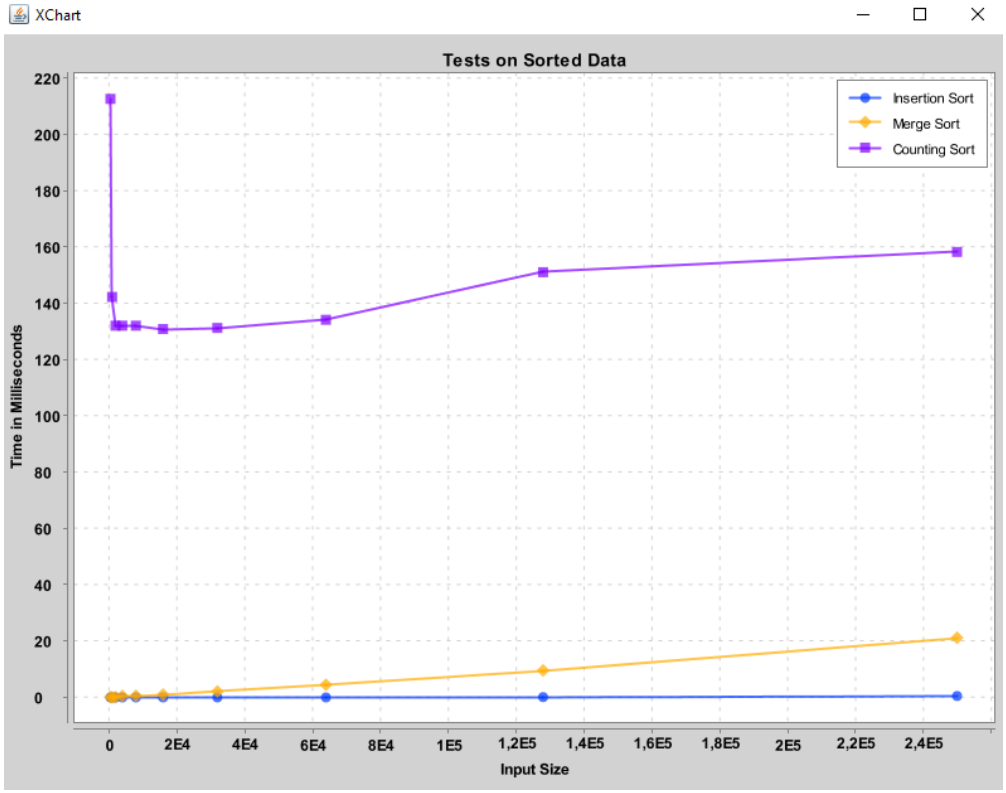

Figure 2: Plot of the Sorting Algorithms (Sorted Data)

These results of sorted data are certainly the best case scenarios. When considering the graph of sorted data, we have a better sight of difference between Counting and Merge Sort. That's because Insertion Sort has $O(1)$ time complexity now as the array is already sorted. Its times are mostly zeroes (which confirms our complexity table), so the graph has better scaling to see lower times. Here, we actually see the result of k number being huge really well. Normally, Counting Sort should have lower running times than Merge Sort, but not this time. So, using Counting Sort for best case scenarios seems very inefficient. Even though Insertion Sort was so much worse in average case scenarios, its the most efficient sorting algorithm here.

5.3 Tests on Reversely Sorted Data

Fig. 3.

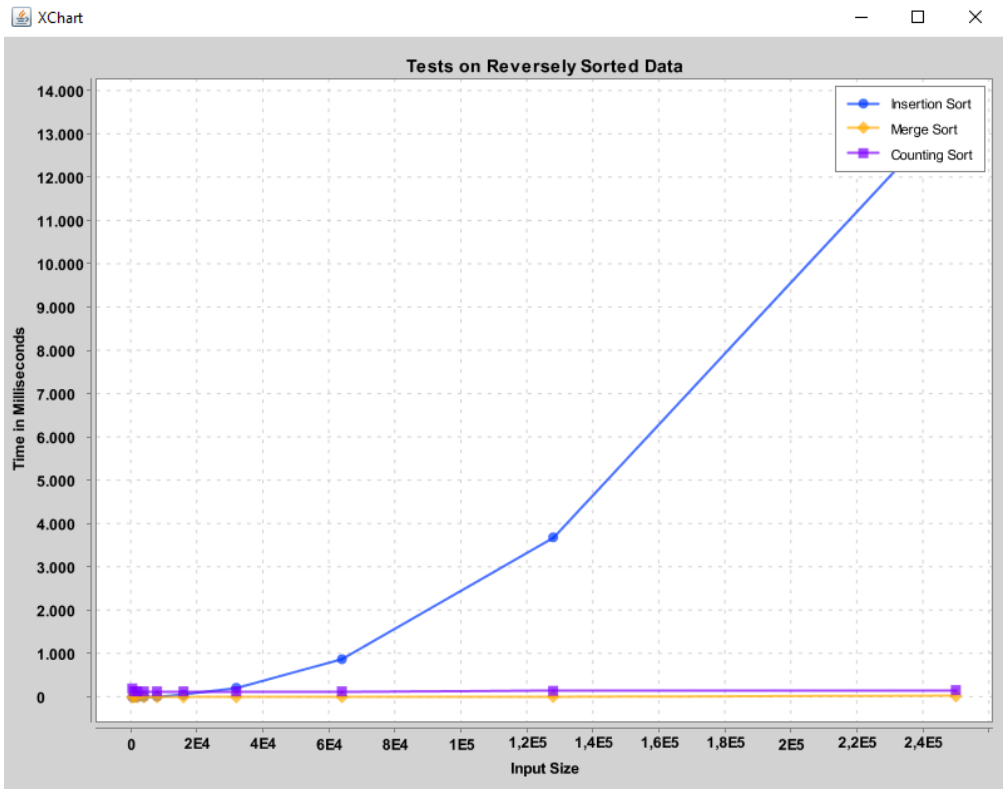

Figure 3: Plot of the Sorting Algorithms (Reversely Sorted Data)

These results of sorted data are certainly the worst case scenarios. But even in the worst case, Merge and Counting Sort have the same running times with the previous tests. Unfortunately, running times of Insertion Sort have doubled themselves. For input size of 250000, 6.5k ms becomes 13k ms here which is nearly twice as much. The results say that Insertion Sort is absolutely the least efficient among three of them for both average and worst case scenarios. Our complexity tables also confirm this information. Similarly to average case scenarios, Merge Sort is also the most efficient sorting algorithm here. So, we can reach a conclusion of worst cases being very close to average cases.

5.4 Tests on Searching

Fig. 4.

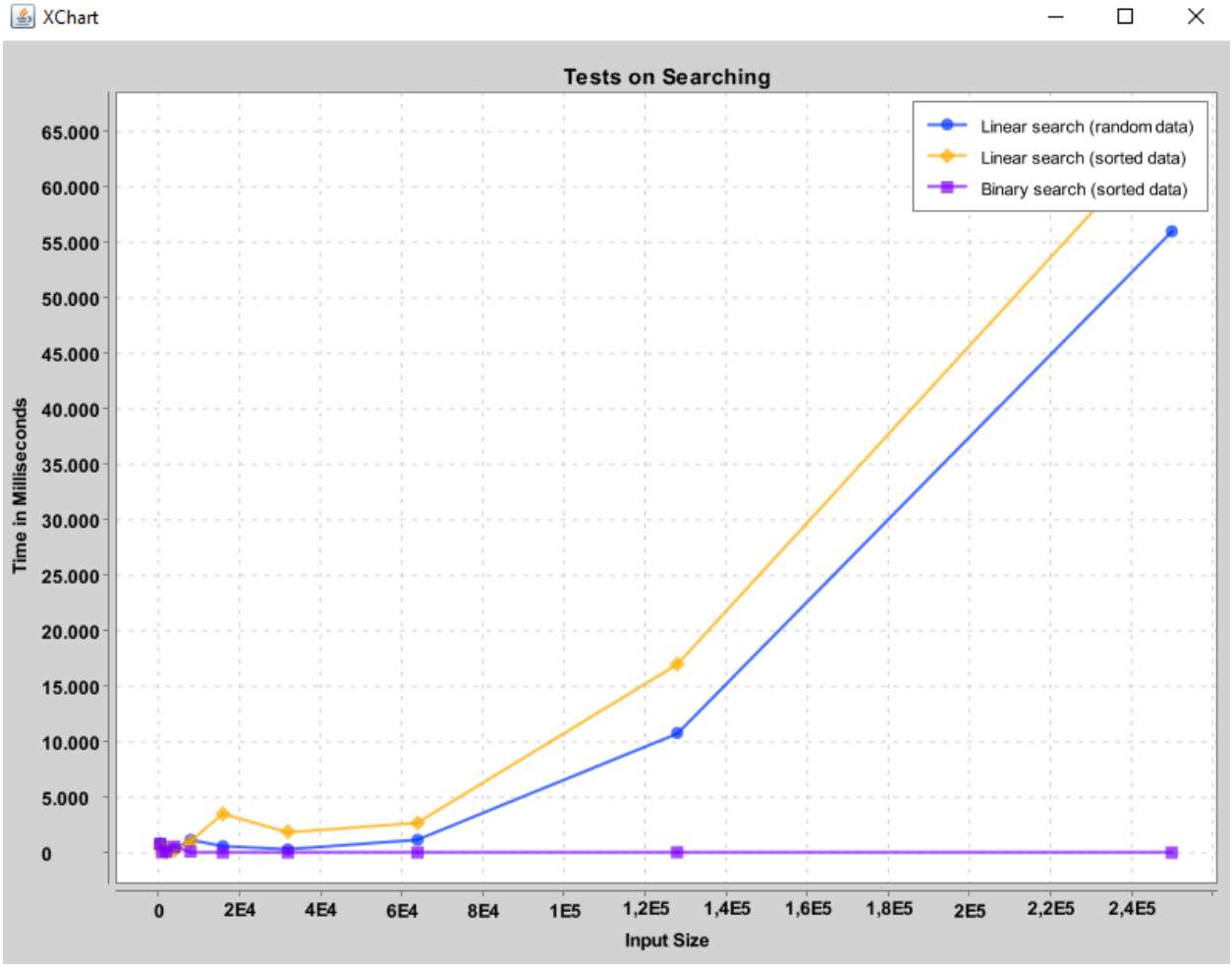

Figure 4: Plot of the Search Algorithms

Since the random number we pick can be in every position of the array, our results might have some randomness. But overall, we see that binary search is significantly more efficient than linear search in every situation. This is because it has a time complexity of $O(\log n)$ instead of $O(n)$. Also, sorted data doesn't mean it is the best case scenario in this test, because being best case is depending only on index position. So, the difference between random data and sorted data has also some randomness in itself. Maybe the number we are searching is the maximum number in the sorted array, but it was in the first position of shuffled array. One is best case, and the other is worst case. But in our tests, it was seen that linear search is slightly more time efficient in sorted data. But it is hard to tell if this gives us real information or if it is just luck.

6 Questions

6.1 What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted/searched?

For sorting algorithms, the best case is always the sorted array (Sorted Data). Because there is nothing to sort, array is already sorted. And the worst case is reversely sorted array as everything is in the furthest place (Reversely Sorted Data). Lastly, the average case is when every item in array is in a random place (Random Data).

For Linear Search, the best case is when item to search is in the first index of array. And the worst case is when item to search is in the last index of array. Lastly, anywhere in between is the average case.

For Binary Search, the best case is when item to search is in the middle index of array. And the worst is when item to search is in the first or last index of array. Lastly, anywhere in between first/middle and middle/last is the average case.

6.2 Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?

Most of our obtained results matched their theoretical asymptotic complexities, but there were just one exception. Counting Sort should have taken less running time than Merge Sort theoretically. But in our experiment it was quite the opposite. At least, it made sense because k value was so high than n value Merge Sort was using. Other than this, every sorting and searching algorithm has matched what was on paper. We have successfully showed Insertion Sort having higher running times as its complexity is $O(n^2)$. We have also showed Binary Search having lower running times than Linear Search as its complexity is $O(\log n)$ and not $O(n)$. Overall, we can say that this experiment was a success.

References

- javatpoint.com (Studying sorting and searching algorithms)
- overleaf.com (Writing the report)