Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 18, 2024

Student name:
Can Ertaş

Student Number:
b2220356088

1 Problem Definition

In this assignment we are comparing 3 different sorting algorithms and two different searching algorithms in terms of their time efficiency and auxiliary space complexity. We are using different kind of data (Random, Sorted or Reversed) to see how it behaves in different conditions so we can choose the best algorithm that suits our goals at our current situation.

2 Solution Implementation

I created two classes named Sort, Search for the implementation of sorting and searching algorithms. All the sorting and searching methods are static.

2.1 Insertion Sort Algorithm

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

In this algorithm, the main variable consuming auxiliary space is the key variable at line 3. This variable holds the value of the current element being compared to other variables. So the auxiliary space complexity of this algorithm is $O(1)$.

2.2 Merge Sort Algorithm

The method named merge has a private access, as more access will not be needed.

```
12 protected static int[] mergeSort(int [] arr){
13     int len = arr.length;
14
15     if(len <= 1){
16         return arr;
17     }
18
19     int [] left = new int[len / 2];
20     System.arraycopy(arr, 0, left, 0, len / 2);
21     left = mergeSort(left);
22
23     int [] right = new int[len - len/2];
```

```
24      System.arraycopy(arr, len / 2, right, 0, len - (len / 2));
25      right = mergeSort(right);
26
27      return merge(left, right);
28  }
29
30  private static int[] merge(int[] arr1, int[] arr2){
31      int [] result = new int [arr1.length + arr2.length];
32      int i = 0;
33      int j = 0;
34
35      int r = 0;
36      while (i < arr1.length && j < arr2.length){
37          if(arr1[i] < arr2[j]){
38              result[r++] = arr1[i++];
39          }else{
40              result[r++] = arr2[j++];
41          }
42      }
43
44      while (i < arr1.length){
45          result[r++] = arr1[i++];
46      }
47      while (j < arr2.length){
48          result[r++] = arr2[j++];
49      }
50
51      return result;
52  }
```

In the Merge Sort algorithm; at lines 19, 23, 31. Three integer arrays whose sizes correlate with size of the input arrays were created. So the auixiliary space complexity is $O(n)$.

2.3 Counting Sort Algorithm

```
54  protected static int[] countingSort(int [] input, int k){
55      int size = input.length;
56      int [] count = new int [k + 1];
57      int [] output = new int [size];
58
59      for (int j : input) {
60          count[j]++;
61      }
62
63      for(int i = 1; i <= k; i++){
64          count[i] = count[i] + count[i - 1];
65      }
```

```
66
67      for(int i = size - 1; i >= 0; i--){
68          int j = input[i];
69          count[j]--;
70          output[count[j]] = j;
71      }
72
73      return output;
74  }
```

In the Counting Sort algorithm, the lines 56, 57 clearly show us two integer arrays whose sizes are k + 1 (k : largest value in the array) and n (the length of the input array) respectively. So the auixiliary space complexity of this algorithm is $O(n + k)$.

2.4 Linear Search Algorithm

```
75  protected static int linearSearch(int [] arr, int x){
76      int size = arr.length;
77      for (int i = 0; i < size; i++){
78          if(arr[i] == x)
79              return i;
80      }
81      return -1;
82  }
```

In the Linear Search algorithm, the only memory spaces were created for size (not even needed actually) and i variables. There is no any must-create variable in the algorithm. So the auixiliary space complexity of this algorithm is $O(1)$.

2.5 Binary Search Algorithm

```
83      protected static int binarySearch(int [] arr, int x){
84          int low = 0;
85          int high = arr.length - 1;
86          while (high - low > 1){
87              int mid = (high + low) / 2;
88              if(x > arr[mid]){
89                  low = mid + 1;
90              }else{
91                  high = mid;
92              }
93          }
94          if(arr[low] == x){
95              return low;
96          }else if(arr[high] == x){
97              return high;
```

```
98              }
99
100             return -1;
101         }
```

In the Binary Search algorithm, at lines 84, 85 two necessary main integer variables were created, other than these there are no significant variables created. So the auixiliary space complexity of this algorithm is $O(1)$.

3 Results, Analysis, Discussion

When we have a chunk of Random Data, Insertion Sort is absolutely a complete time wasting algorithm for large data whereas counting sort does a much better job it takes almost constant amount of time (I belive because the data size even 250000 is small for counting sort algorithm to look like an $O(n+k)$ algorithm, so it looks like $O(1)$). And finally Merge Sort is undoubtedly the best for this kind of data with $O(nlogn)$ time complexity as seen from the table.

When we have Sorted Data, by the design of Insertion Sort it only compares every value in the array to the previous value so it just takes $O(n)$ time complexity and is very efficient. Merge Sort performs very well with $O(nlogn)$ time complexity. Counting Sort also performs very well but there is a sudden increase whose reason I still have not been able to understand frankly.

When we have Reversed Data (In descending order). Approximately, the time Insertion Sort takes is doubled which makes it much worse, on the contrary the time Merge Sort takes is almost halved. Counting Sort looks once again almost constant. (So again my opinion is that data sizes are so small for counting sort algorithm to look $O(n+k)$, and that is why it looks like constant $O(1)$ algorithm)

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	0	0	1	3	16	58	226	90	3909
Merge sort	0	0	0	1	0	1	2	6	15	24
Counting sort	106	69	67	67	70	74	67	66	67	67
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	0
Merge sort	0	0	0	0	0	1	2	5	6	10
Counting sort	0	0	0	0	0	0	0	0	0	74
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	7	29	116	481	1940	7498
Merge sort	0	0	0	0	0	0	1	2	5	10
Counting sort	67	67	67	69	70	68	70	80	75	70

Below, We can see the running times the two searching algorithms take. Of course with $O(logn)$ time complexity Binary Search performs very well. Linear Search, as expected, behaves with $O(n)$ time complexity for both random and sorted data.

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)	1504	220	361	375	970	1111	1989	3701	12425	24745
Linear search (sorted data)	119	166	237	364	685	1182	2215	4316	8712	17945
Binary search (sorted data)	188	144	135	128	171	196	234	312	536	1719

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	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

How these Auxiliary space complexities were obtained were explained above under each code segment of each algorithm.

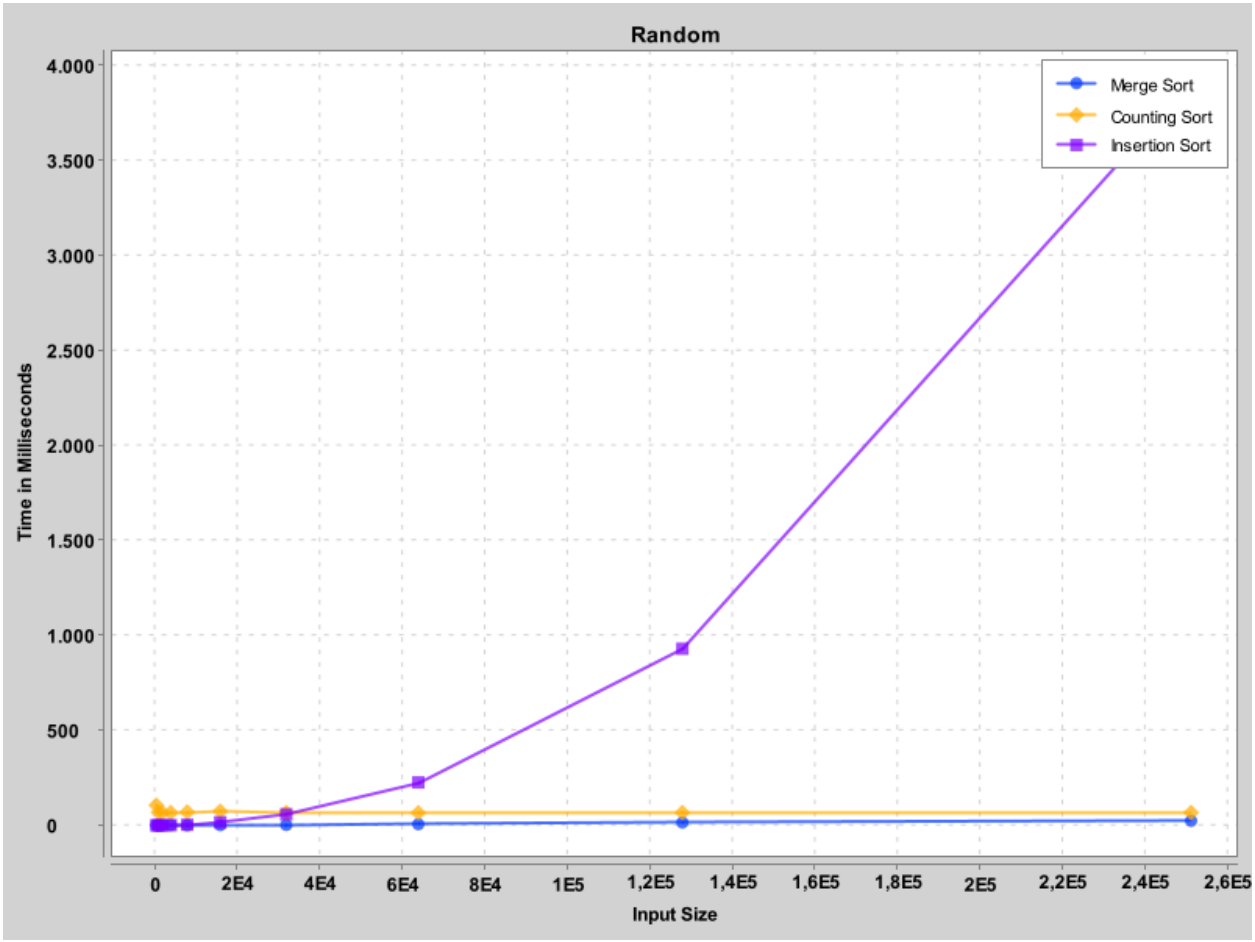

Figure 1: Time - Data Size graph of Sorting Algorithms with Random Data

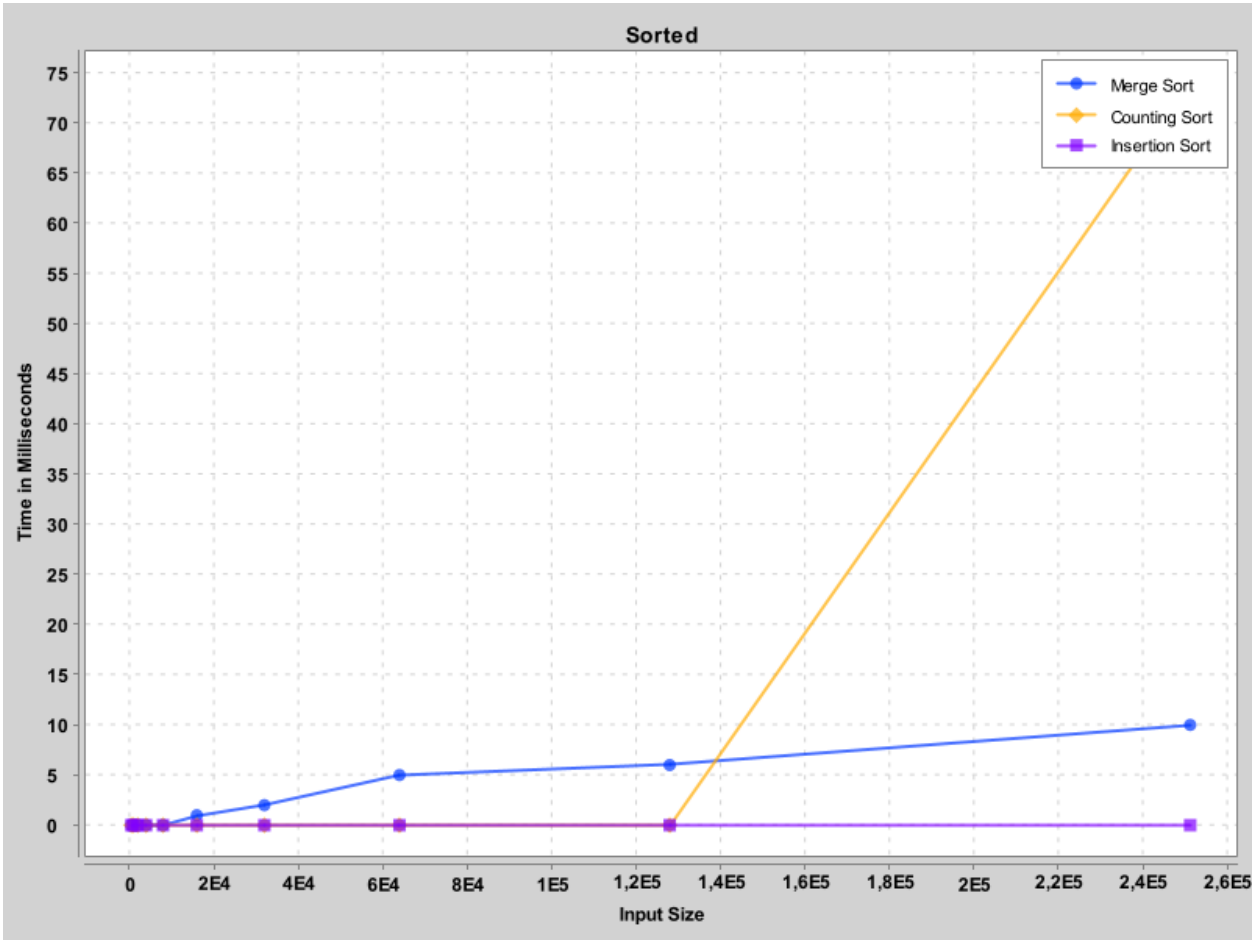

Figure 2: Time - Data Size graph of Sorting Algorithms with Sorted Data

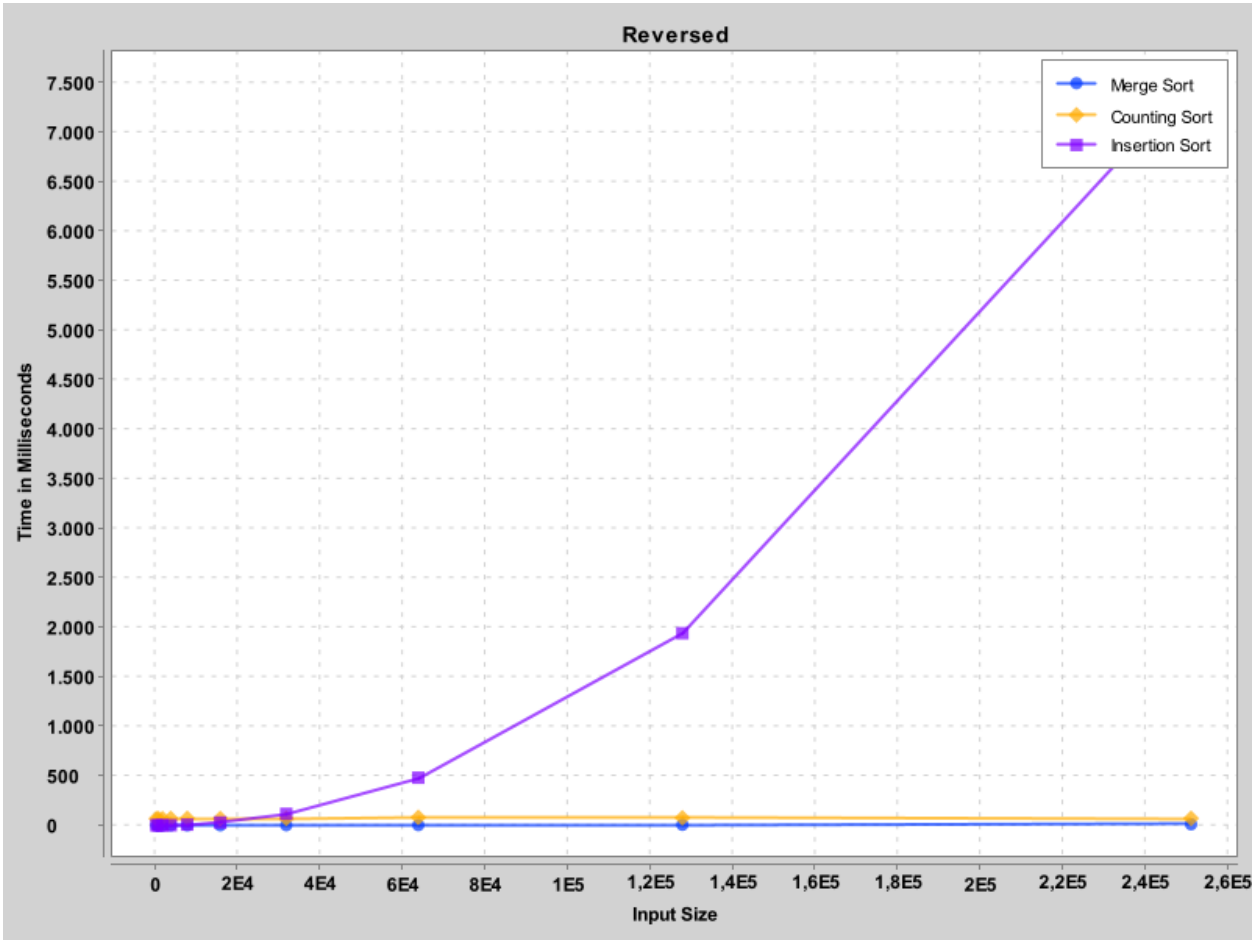

Figure 3: Time - Data Size graph of Sorting Algorithms with Reversed Data

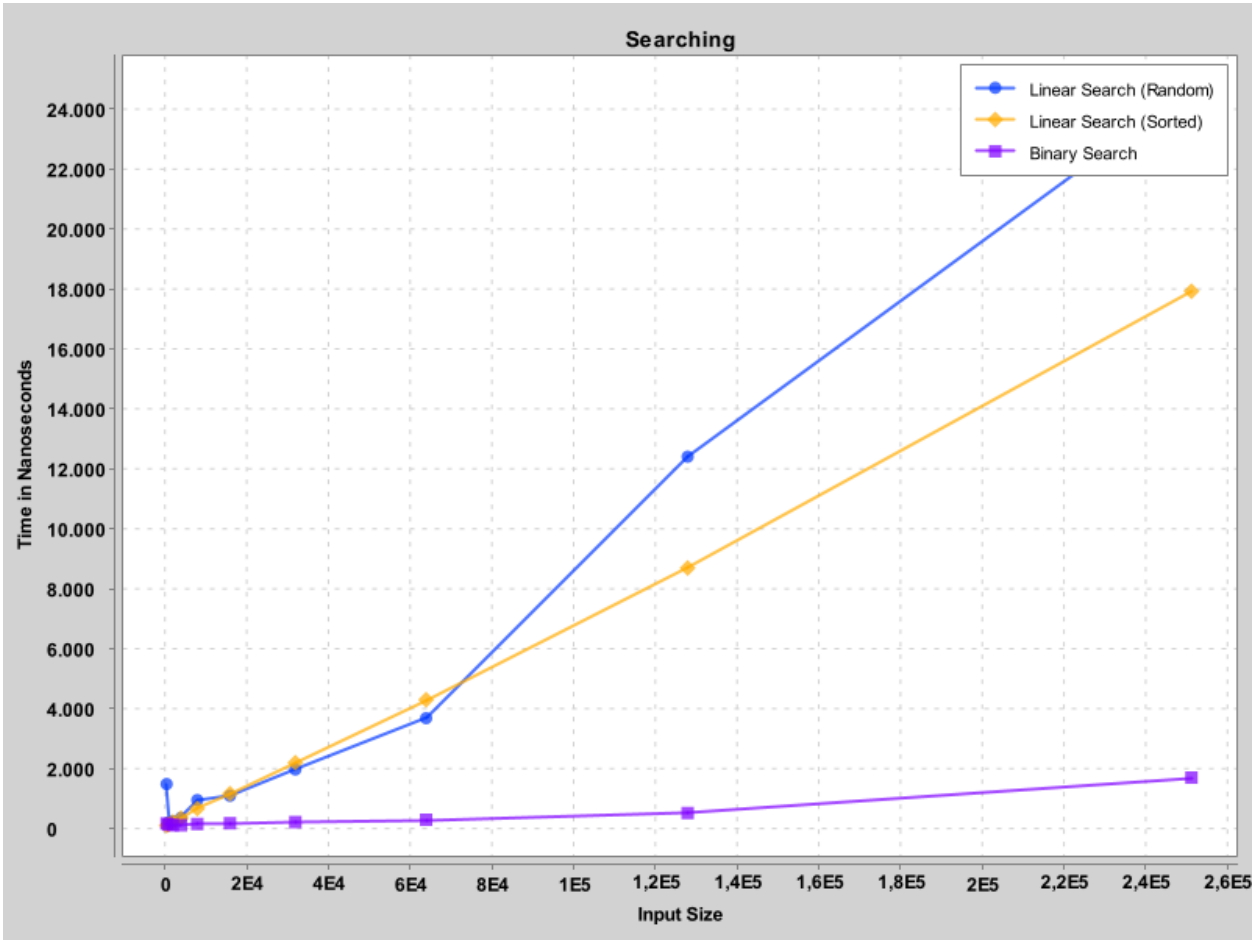

Figure 4: Time - Data Size graph of Searching Algorithms