Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 21, 2024

Student name:
Barış Yildiz

Student Number:
b2210356107

1 Problem Definition

There are many algorithms designed for optimizing the sorting and searching problems common in computer science. Some of these algorithms are more efficient at sorting/searching on input arrays of different characteristics while some are far more efficient than others on all kinds of input arrays. In this assignment, the sorting algorithms Insertion Sort, Merge Sort and Counting Sort; the searching algorithms Linear Search and Binary Search are tested on input arrays of different characteristics and their execution times are recorded.

2 Solution Implementation

All algorithms are tested using integer input arrays of sizes 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 and 250000. The sorting algorithms were executed 10 times for each input array of different size while the searching algorithms were executed 1000 times, each time searching for a different value. The average execution time was calculated for each algorithm, for each input array and plotted using the XChart library. The implementations of the sorting and searching algorithms subject to the assignment are provided below.

2.1 Insertion Sort

```
1  public 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
6          while(i>=0 && arr[i] > key) {
7              arr[i+1] = arr[i];
8              i--;
9          }
10
11         arr[i+1] = key;
12     }
13 }
```

2.2 Merge Sort

```java
public static int[] mergeSort(int[] arr) {
    if (arr.length <= 1) {
        return arr;
    }

    int[] left = new int[arr.length/2];
    System.arraycopy(arr, 0, left, 0, left.length);

    int[] right = new int[arr.length - arr.length/2];
    System.arraycopy(arr, arr.length/2, right, 0, right.length);

    left = mergeSort(left);
    right = mergeSort(right);
    return merge(left, right);
}

private static int[] merge(int[] left, int[] right) {
    int[] merged = new int[left.length + right.length];

    int leftPointer = 0, rightPointer = 0, mergedPointer = 0;

    while (leftPointer < left.length && rightPointer < right.length) {
        merged[mergedPointer++] = (left[leftPointer] < right[rightPointer]) ?
            left[leftPointer++] : right[rightPointer++];
    }

    while (leftPointer < left.length) {
        merged[mergedPointer++] = left[leftPointer++];
    }

    while(rightPointer < right.length) {
        merged[mergedPointer++] = right[rightPointer++];
    }

    return merged;
}
```

2.3 Counting Sort

```
public static int[] countingSort(int[] arr) {

    int max = arr[0];

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    int[] countArr = new int[max+1];
    int[] outputArr = new int[arr.length];

    for (int i = 0; i < arr.length; i++) {
        countArr[arr[i]]++;
    }

    for (int i = 1; i < countArr.length; i++) {
        countArr[i] += countArr[i-1];
    }

    for (int i = arr.length-1; i > -1; i--) {
        int j = arr[i];
        countArr[j]--;
        outputArr[countArr[j]] = arr[i];
    }

    return outputArr;
}
```

2.4 Linear Search

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

2.5 Binary Search

```
86  public static int binarySearch(int[] arr, int x) {
87      int low = 0;
88      int high = arr.length-1;
89
90      while (low <= high) {
91          int mid = low + (high-low)/2;
92          int midValue = arr[mid];
93
94          if (midValue == x) {
95              return mid;
96          } else if (midValue < x) {
97              low = mid+1;
98          } else {
99              high = mid-1;
100         }
101     }
102
103     return -1;
104 }
```

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	0.0	0.0	0.0	0.0	0.5	1.4	6.6	28.6	123.7	512.7
Merge sort	0.0	0.0	0.0	0.0	0.0	0.2	0.5	1.7	4.7	10.3
Counting sort	161.0	118.2	109.4	109.1	109.3	109.4	109.2	110.0	122.2	112.6
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Merge sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	3.0	7.0
Counting sort	109.1	108.8	117.1	108.4	110.3	119.7	118.1	112.8	105.5	107.1
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	1.0	5.5	31.2	119.5	508.7	2202.4	8426.4
Merge sort	0.0	0.0	0.0	0.0	0.0	0.1	0.2	1.1	3.1	7.2
Counting sort	103.2	104.6	104.5	104.9	104.4	114.2	113.3	106.5	107.3	112.1

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)	1168.8	999.8	149.0	244.4	435.7	877.3	1485.1	3628.0	6080.5	10491.4
Linear search (sorted data)	448.6	91.8	155.7	294.7	527.8	1167.1	2398.4	4601.4	9779.1	16782.9
Binary search (sorted data)	156.5	86.7	97.5	75.8	68.7	77.0	74.2	82.5	91.6	97.1

From Table 1, several conclusions about the sorting algorithms can be made.

Insertion Sort: The best case occurs when the input array is already sorted. The algorithm scans every element once and determines that it is sorted. On the average case, which is when elements in the input array are randomized, Insertion Sort scans every element and for each element, scans some the previous elements again. The worst case for Insertion Sort is sorting a reversely sorted input array. In this case, for each scanned element in the array, all previous elements are scanned again. This results in Insertion Sort having $O(n)$ time complexity in the best case, and $O(n^2)$ time complexity on the average and worst case.(3) One of the advantages of Insertion Sort is that it has $O(1)$ auxiliary space complexity. The sorting is done in-place without needing additional memory space.(4)

Merge Sort: According to the Merge Sort data in Table 1, there is not much difference between the execution times in the best, average and worst case. However, on input arrays of larger sizes, the difference might be more clear as data suggests that execution times increase as input size gets bigger. Analyzing the algorithm, it can be seen that Merge Sort divides the original array into sub arrays $log(n)$ times and scans every sub array element once. This evaluates to $O(nlogn)$

time complexity(3). The main disadvantage of Merge Sort is that it uses $O(n)$ auxiliary space.(4) Sorting is done using auxiliary arrays (at lines 19, 22 and 31) holding approximately 2n elements.

Counting Sort: The data displays that there is not much difference between the execution times, but unlike Merge Sort, the execution times are not always increasing with the input size. The reason for this is that Counting Sort depends on not only the input size, but also the maximum element in the input array, so there is no best and worst case determined by input array type.(3) The algorithm creates an auxiliary array of size equal to the maximum element and stores frequency of all elements in there. After that, the algorithm goes through every frequency stored and every element of the original array again, resulting in $O(n+k)$ time complexity where k is the maximum element. Counting Sort also runs with $O(n+k)$ space complexity(4) since it creates two additional arrays of sizes k and n (at lines 59 and 60).

From Table 2, several conclusions about the searching algorithms can be made.

Linear Search: It is observed that the execution times of Linear Search are higher on sorted data. The reason for this might be that on sorted data, approximately half of the time the algorithm searches past the middle and half of the time before it, while on unsorted data, the search usually doesn't go much further past the middle. This fact becomes more apparent as size grows. The best case for Linear Search is when the algorithm finds the target element at index 0, without needing to check more than 1 element. The average case is when the algorithm needs to check half of the elements, and the worst case is when the algorithm checks every element. Therefore, the time complexity for the best, average and worst cases are $O(1)$, $O(n)$ and $O(n)$.(3) Linear Search searches in-place, using only $O(1)$ auxiliary space.(4)

Binary Search: The Binary Search algorithm is observed to be performing much more efficiently than Linear Search, with the downside of not working on random datasets. Binary Search searches the array by dividing it into halves, therefore having only $O(logn)$ time complexity. However, the target might be in the middle, which results in the algorithm checking only that element and terminating in $O(1)$ time complexity.(3) Similar to Linear Search, Binary Search also searches in-place with $O(1)$ space complexity.(4)

The time and auxiliary space complexity tables Table 3 and 4 are provided below.

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

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	$O(1)$

In general, data confirms theoretical information with some exceptions such as:

- Insertion Sort seems to be running in $O(1)$ time complexity on sorted data. This is probably due to precision issues.
- Data on Merge Sort is not clear as Merge Sort is very efficient at sorting provided input arrays.

4 Plots

Plots of data in tables 1 and 2 are provided below.

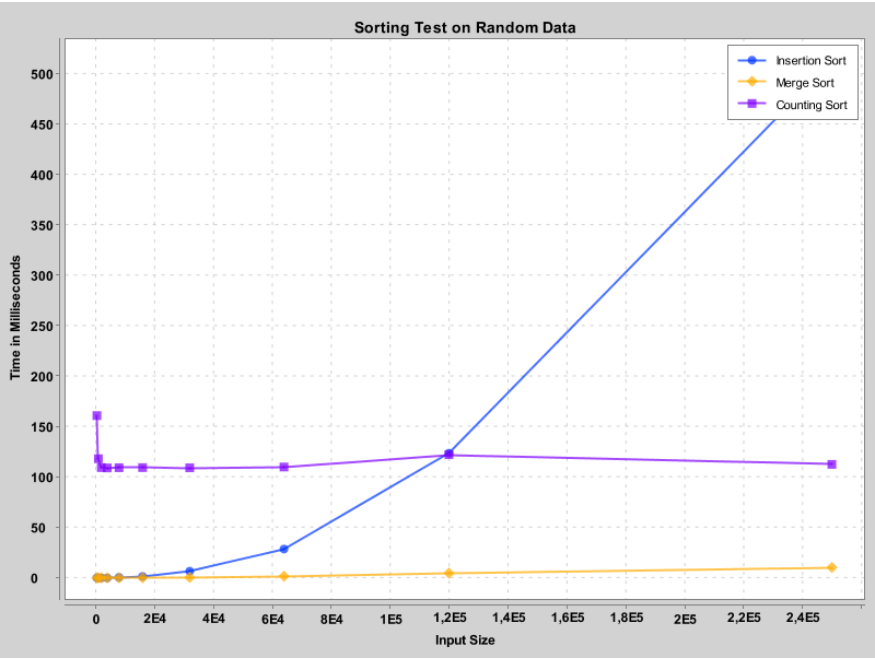

Figure 1: Plot of the sorting algorithms on random data.

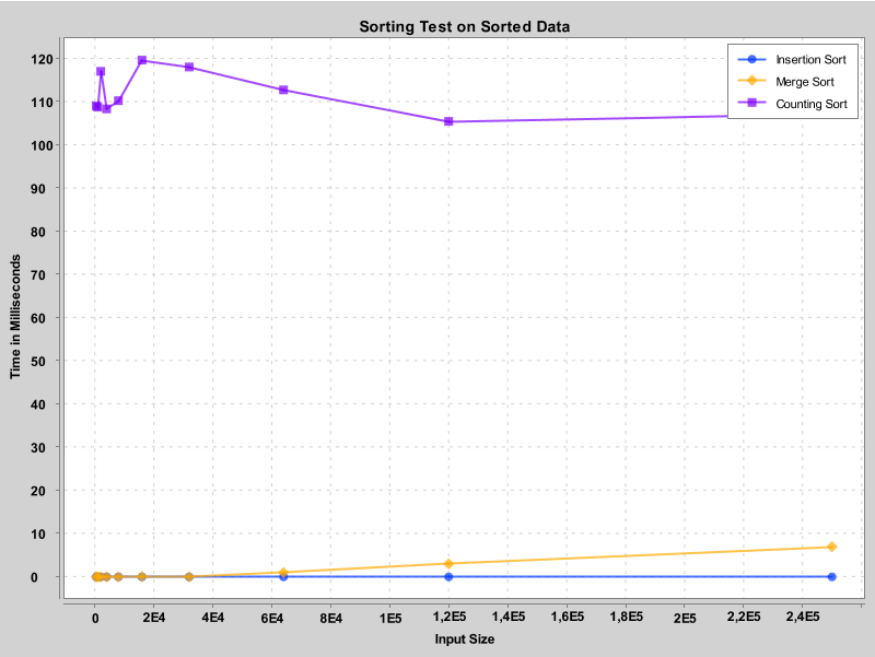

Figure 2: Plot of the sorting algorithms on sorted data.

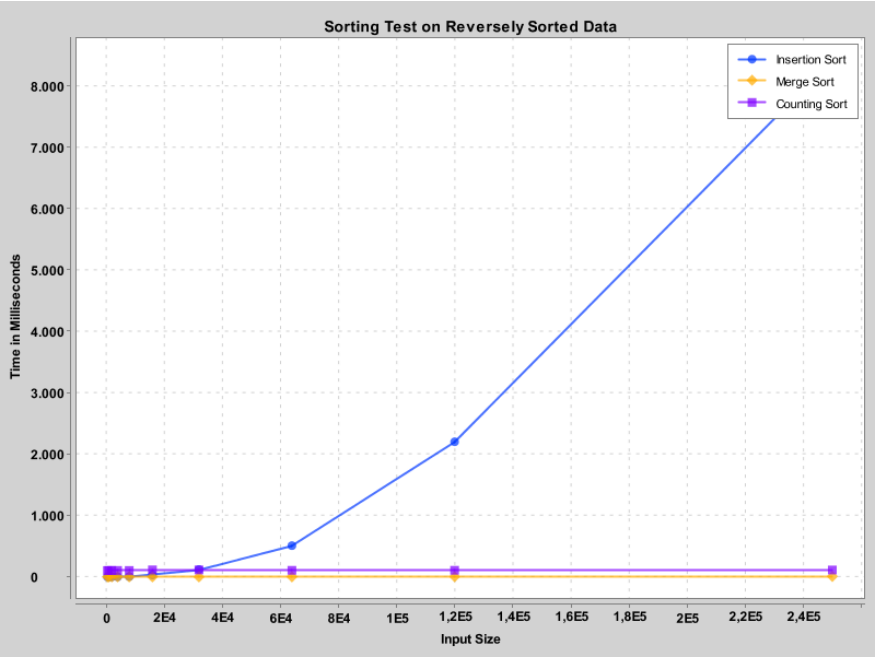

Figure 3: Plot of the sorting algorithms on reversely sorted data (sorted by descending order).

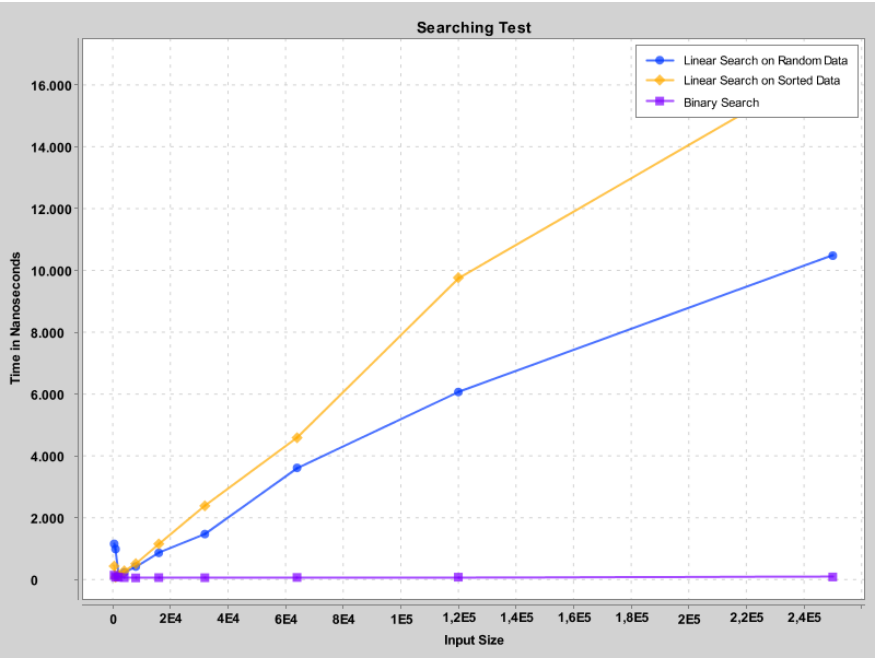

Figure 4: Plot of the searching algorithms.