Hacettepe University
Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 17, 2024

Student Name:
Oktay Kaplan

Student Number:
b2210356010

Contents

1 Problem Definition **3**

2 Solution Implementation **3**

2.1 Insertion Sort Algorithm 3

2.2 MergeSort Algorithm 4

2.3 Counting Sort Algorithm 5

2.4 Linear Search Algorithm 6

2.5 Binary Search Algorithm 6

3 Results, Analysis, Discussion **7**

1 Problem Definition

We've conducted a thorough analysis of sorting and searching algorithms, evaluating their efficiency across diverse datasets. Our study focused on three sorting algorithms (insertion, merge, and counting) and two searching algorithms (linear and binary).

Using milliseconds for sorting algorithms and nanoseconds for searching algorithms, we measured their execution times across different data sizes. Additionally, we analyzed the computational and auxiliary space complexities associated with each algorithm.

To present our findings effectively, we created graphs for each dataset, offering a visual representation of algorithmic efficiency. Our study provides valuable insights into the performance and scalability of these algorithms, aiding in informed decision-making for practical applications and algorithmic optimizations.

2 Solution Implementation

2.1 Insertion Sort Algorithm

```
public void sort(int[] arr) {
        for (int j = 1; j < arr.length; j++) {
            int key = arr[j];
            int i = j - 1;
            while (i >= 0 && arr[i] > key) {
                arr[i + 1] = arr[i];
                i--;
            }
            arr[i + 1] = key;
        }
}
```

2.2 MergeSort Algorithm

```java
public void sort(int[] arr) {
        mergeSort(arr);
    }

public static int[] mergeSort(int[] A) {
    int n = A.length;
    if (n <= 1) {
        return A;
    }
    int mid = n / 2;
    int[] left = Arrays.copyOfRange(A, 0, mid);
    int[] right = Arrays.copyOfRange(A, mid, n);

    left = mergeSort(left);
    right = mergeSort(right);

    return merge(left, right);
}

private static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int i = 0, j = 0, k = 0;

    while (i < A.length && j < B.length) {
        if (A[i] <= B[j]) {
            C[k] = A[i];
            i++;
        } else {
            C[k] = B[j];
            j++;
        }
        k++;
    }
    while (i < A.length) {
        C[k] = A[i];
        i++;
        k++;
    }
    while (j < B.length) {
        C[k] = B[j];
        j++;
        k++;
    }
    return C;
}
```

2.3 Counting Sort Algorithm

```
60 public void sort(int[] arr) {
61         countingSort(arr, Arrays.stream(arr).max().orElse(0));
62 }
63
64 public static int[] countingSort(int[] A, int k) {
65     int[] count = new int[(k + 1)];
66     int[] output = new int[A.length];
67     int size = A.length;
68
69     for (int i = 0; i < size; i++) {
70         count[A[i]]++;
71     }
72
73     for (int i = 1; i <= k; i++) {
74         count[i] += count[i - 1];
75     }
76
77     for (int i = size - 1; i >= 0; i--) {
78         output[(count[ A[i]] - 1)] = A[i];
79         count[ A[i]]--;
80     }
81
82     return output;
83 }
```

2.4 Linear Search Algorithm

```
86 public int Search(int[] arr, int value) {
87         int size = arr.length;
88         for (int i = 0; i < size; i++) {
89             if (arr[i] == value) {
90                 return i;
91             }
92         }
93         return -1;
94 }
```

2.5 Binary Search Algorithm

```
96 public int Search(int[] arr, int value) {
97         int lowIndex = 0;
98         int highIndex = arr.length - 1;
99 
100        while (highIndex - lowIndex > 1){
101            int mid = (highIndex + lowIndex) / 2;
102            if(arr[mid] < value) {
103                lowIndex = mid + 1;
104            }
105            else {
106                highIndex = mid;
107            }
108        }
109 
110        if(arr[lowIndex] == value)
111            return lowIndex;
112 
113        if(arr[highIndex] == value)
114            return highIndex;
115 
116        return -1;
117 }
```

3 Results, Analysis, Discussion

Table 1: Results of the running time tests performed for varying input sizes (in ms).

Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion sort	0.5	0.3	0.7	2.0	7.0	33.0	108.3	387.4	1684.0	6986.8
Merge sort	0.2	0.1	0.3	0.3	0.8	1.5	3.1	7.6	20.4	34.9
Counting sort	296.1	168.0	149.2	108.3	104.2	103.8	105.3	107.0	110.8	120.6
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.1	0.3	0.6
Merge sort	0.1	0.0	0.1	0.2	0.4	1.0	1.9	3.8	7.7	23.6
Counting sort	113.5	100.7	96.0	96.7	101.3	104.8	104.8	109.3	107.9	111.0
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.3	0.8	3.2	12.7	50.6	198.9	772.5	3233.2	12653.8
Merge sort	0.0	0.1	0.1	0.2	0.4	0.9	2.0	5.3	14.9	15.8
Counting sort	100.0	104.3	107.3	112.8	107.3	111.7	112.5	112.9	110.5	116.1

Table 2: Results of the running time tests of search algorithms of varying sizes (in ns).

Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	160.8	259.4	396.3	676.7	1253.1	1901.6	3779.3	7611.0	14126.9	26240.5
Linear search (sorted data)	1675.0	2748.4	454.1	835.0	1543.5	2781.2	5771.2	9749.6	23151.8	47888.9
Binary search (sorted data)	298.6	178.0	212.8	124.4	127.7	140.9	157.9	183.2	267.2	366.2

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

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	$O(1)$

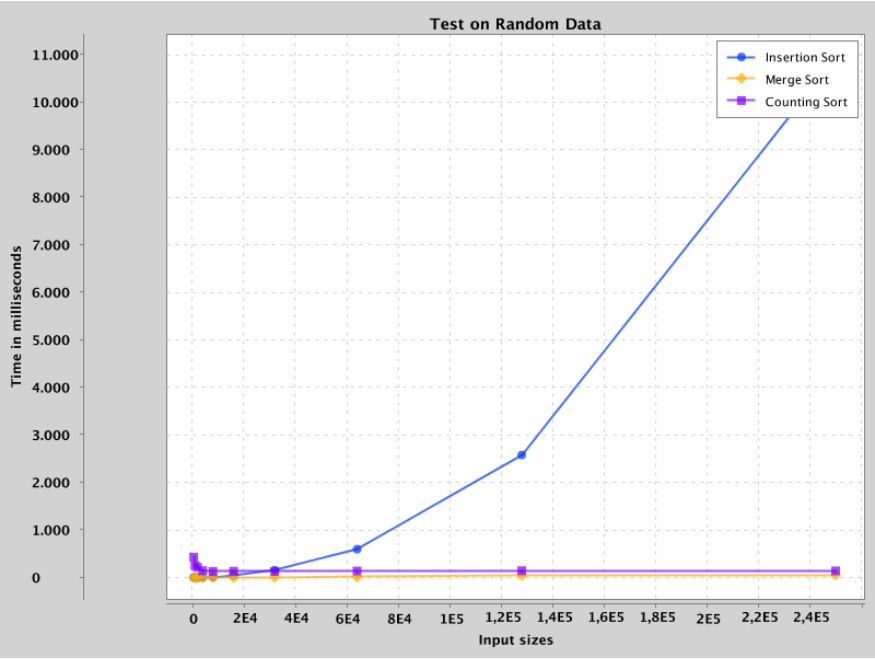

Figure 1: Algorithms Running Times - Random Data Set

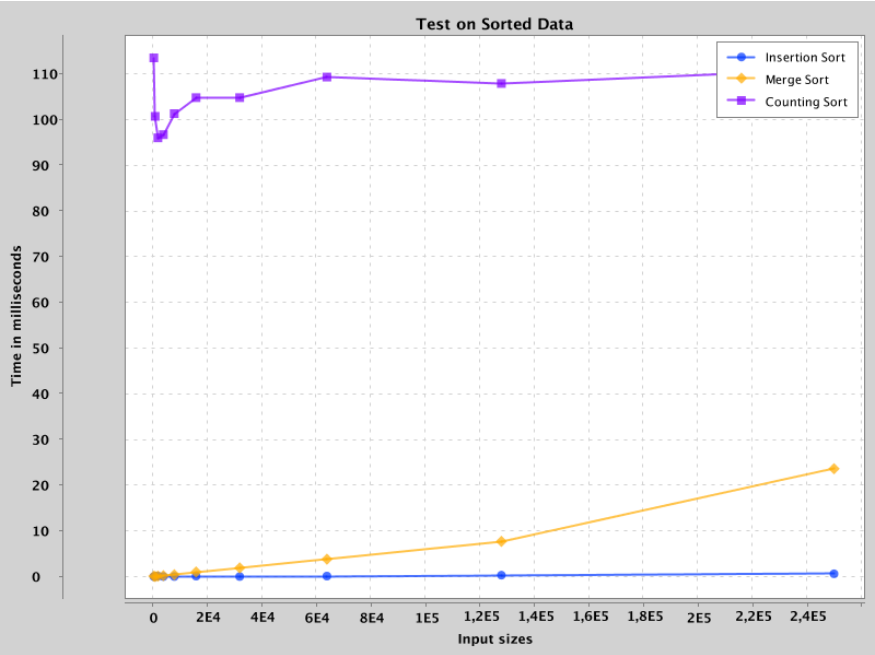

Figure 2: Algorithms Running Times - Sorted Data Set

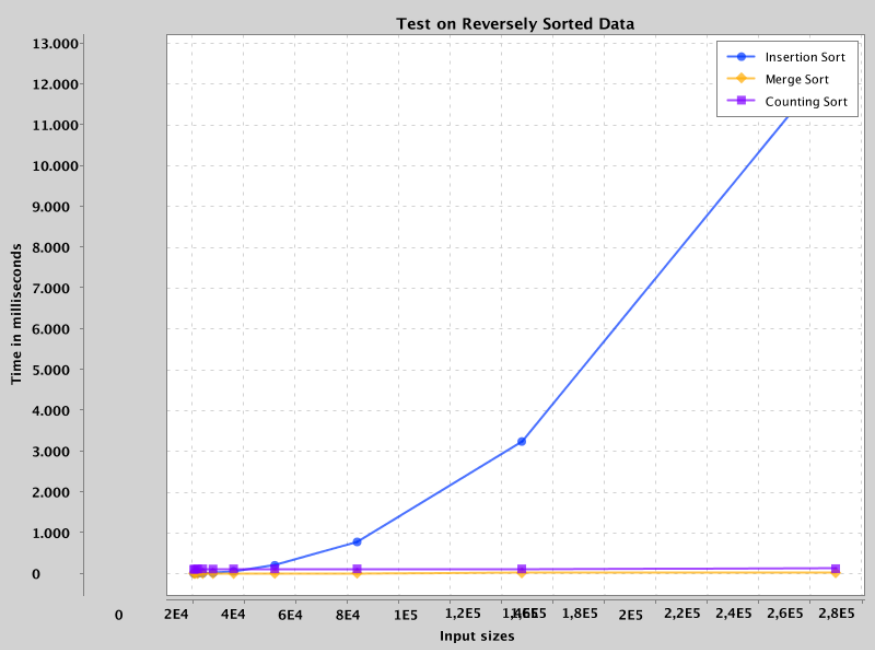

Figure 3: Algorithms Running Times - Reverse Sorted Data Set

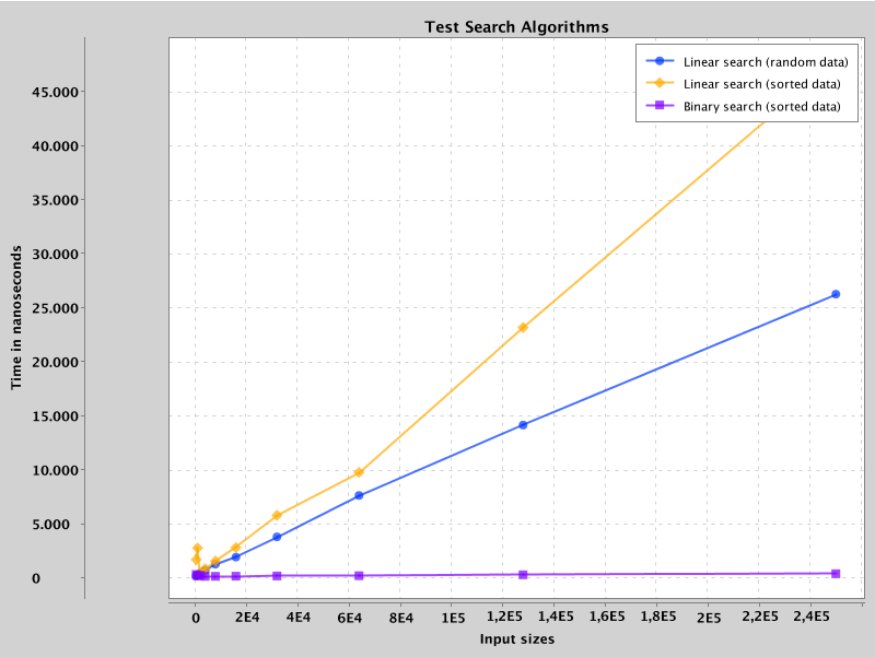

Figure 4: Search Algorithms Running Times - Random&Sorted Data Set

As the input size increases, the time taken also increases for all three sorting algorithms.The increase rate is highest for insertion sort and lowest for counting sort.The difference is not very pronounced in small datasets, but becomes more significant as the dataset grows.

For insertion sort, the best-case scenario appears to be when the data is already sorted because in this case only n-1 comparisons are needed, resulting in $O(n)$ time complexity in the best case. Insertion sort exhibits a quadratic curve with random and reverse-sorted data, fitting into $O(n^2)$ time complexity.

For merge sort, the best-case scenario seems to be when the data is already sorted. Merge sort shows more controlled increases compared to insertion sort and exhibits similar behavior with random and reverse-sorted data, fitting into $O(nlogn)$time complexity.

For counting sort, the best-case scenario appears to be when the data is already sorted because it almost instantly stops sorting in the sorted set. Similarly, counting sort exhibits similar behavior with random and reverse-sorted data, fitting into $O(n + k)$ time complexity.

In terms of search algorithms, the best-case scenario for both search algorithms is under a sorted dataset, which results in $O(1)$ time complexity. Binary search and linear search almost instantly stop searching in a random dataset, resulting in $O(n)$ and $O(logn)$ time complexity, respectively.

References

- Java Program to Reverse a List - GeeksforGeeks
- System.nanoTime() - Tutorialspoint
- System.currentTimeMillis() - GeeksforGeeks
- How to randomly pick an element from an array - Stack Overflow
- Reading a CSV File in Java - Baeldung
- System.arraycopy() in Java - GeeksforGeeks
- XChart Example Code - Knowm.org