Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 18, 2024

Student name: Ömer Faruk Güler

Student Number: b2210356084

1 Problem Definition

This assignment focuses on the analysis of algorithm efficiency, particularly sorting and searching algorithms, in terms of their computational (time) and space complexity. The objective is to implement specified sorting and searching algorithms in Java and analyze their performance on datasets of varying sizes to understand how empirical data aligns with theoretical asymptotic complexities.

2 Solution Implementation

Three sorting algorithms—insertion sort, merge sort, counting sort—and two searching algorithms—linear search and binary search—needed to be implemented. Here are the implementations:

2.1 Sorting Algorithm 1: Insertion Sort

Insertion Sort Implementation in Java:

```java
public static void insertionSort(int[] array) {
        int n = array.length;

        for (int i = 1; i < n; i++) {
            int key = array[i];
            int j = i - 1;

            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = key;
        }
    }
```

2.2 Sorting Algorithm 2: Merge Sort

Merge Sort Implementation in Java:

```java
public static int[] mergeSort(int[] array) {
        if (array == null || array.length <= 1) {
            return array;
        }
        int mid = array.length / 2;

        int[] leftHalf = new int[mid];
        int[] rightHalf = new int[array.length - mid];

        System.arraycopy(array, 0, leftHalf, 0, mid);
        System.arraycopy(array, mid, rightHalf, 0, array.length - mid);

        leftHalf = mergeSort(leftHalf);
        rightHalf = mergeSort(rightHalf);

        return merge(leftHalf, rightHalf);
    }

    private static int[] merge(int[] leftHalf, int[] rightHalf) {
        int leftSize = leftHalf.length;
        int rightSize = rightHalf.length;
        int[] mergedArray = new int[leftSize + rightSize];

        int i = 0, j = 0, k = 0;

        while (i < leftSize && j < rightSize) {
            if (leftHalf[i] <= rightHalf[j]) {
                mergedArray[k++] = leftHalf[i++];
            } else {
                mergedArray[k++] = rightHalf[j++];
            }
        }

        while (i < leftSize) {
            mergedArray[k++] = leftHalf[i++];
        }
        while (j < rightSize) {
            mergedArray[k++] = rightHalf[j++];
        }

        return mergedArray;
    }
```

2.3 Sorting Algorithm 3: Counting Sort

Counting Sort Implementation in Java:

```java
public static void countSort(int[] array) {
        if (array.length == 0) {
            return;
        }

        int k = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i] > k) {
                k = array[i];
            }
        }

        int[] count = new int[k + 1];
        int size = array.length;
        int[] output = new int[size];

        for (int i = 0; i <= k; ++i) {
            count[i] = 0;
        }

        for (int i = 0; i < size; i++) {
            count[array[i]]++;
        }

        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }

        for (int i = size - 1; i >= 0; i--) {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }

    }
```

2.4 Searching Algorithm 1: Linear Search

Linear Search Implementation in Java:

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

2.5 Searching Algorithm 2: Binary Search

Binary Search Implementation in Java:

```
public static int binSearch(int[] A, int x){
        int lo = 0;
        int hi = A.length - 1;
        while ((hi - lo) > 1){
            int mid = (hi + lo) / 2;

            if (A[mid] < x)
                lo = mid+1;
            else
                hi = mid;
        }
        if (A[lo] == x)
            return lo;
        else if (A[hi] == x)
            return hi;
        else
            return -1;
    }
```

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	3	13	45	178	739	3120
Merge sort	0	0	0	0	0	1	2	5	10	21
Counting sort	105	72	72	71	70	71	71	73	72	74
	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	0	0	2	3	8
Counting sort	72	71	71	71	72	72	72	72	73	79
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	6	22	88	364	1478	5544
Merge sort	0	0	0	0	0	0	0	2	4	8
Counting sort	75	72	72	72	72	72	72	72	72	73

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)	1068	1297	1086	400	673	1135	2151	5074	8289	13421
Linear search (sorted data)	131	172	254	426	746	1431	2777	5469	10920	21107
Binary search (sorted data)	431	135	155	154	153	152	152	158	175	186

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

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	$O(1)$

The obtained results generally match the theoretical asymptotic complexities of the sorting and searching algorithms. Insertion Sort shows a sharp increase in time with larger inputs, fitting its O(n^2) complexity. Merge Sort's times are consistently low, aligning with its O(n log n) complexity across all cases. Counting Sort has stable times, which correspond to its O(n+k) complexity. Linear Search's times increase linearly with input size, as expected from its O(n) complexity, and Binary Search shows minimal increases, consistent with its O(log n) complexity.

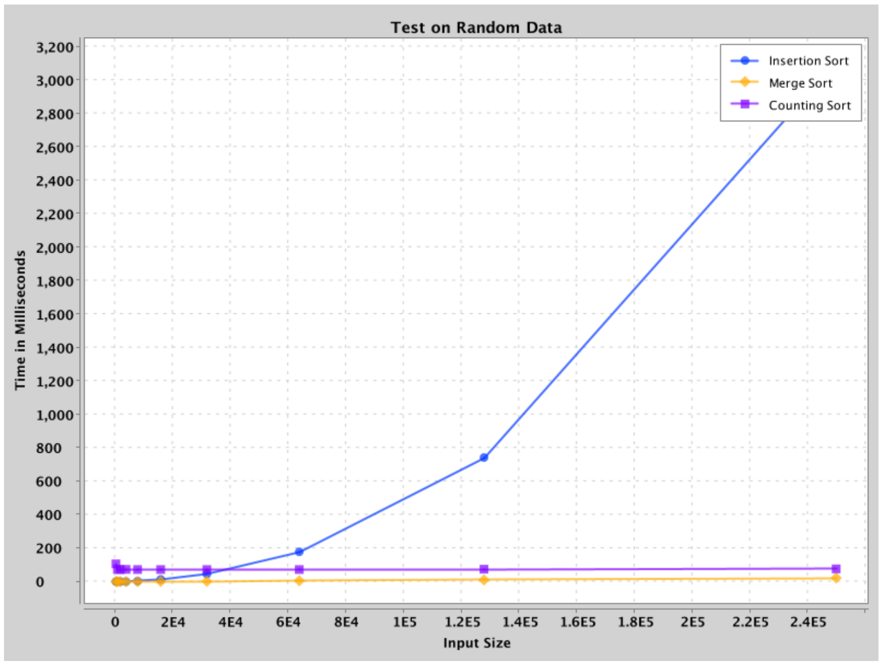

Figure 1: Time Chart for Random Data

Random Data: Insertion Sort's time increases dramatically with input size, confirming its impracticality for large datasets. Merge Sort consistently demonstrates low average processing times across various dataset sizes. Counting Sort also maintains low, stable times across varying input size

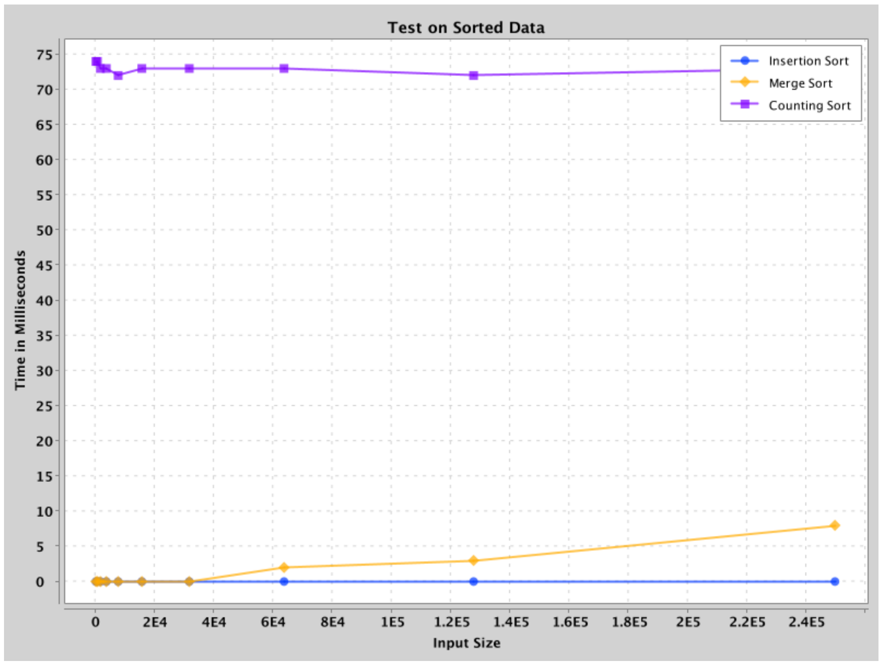

Figure 2: Time Chart for Sorted Data

Sorted Data: Insertion Sort shows excellent efficiency, with consistently minimal running times across various data sizes. Merge Sort's running time rises just a bit with larger data sizes. Counting Sort, on the other hand, displays greater and less consistent running times.

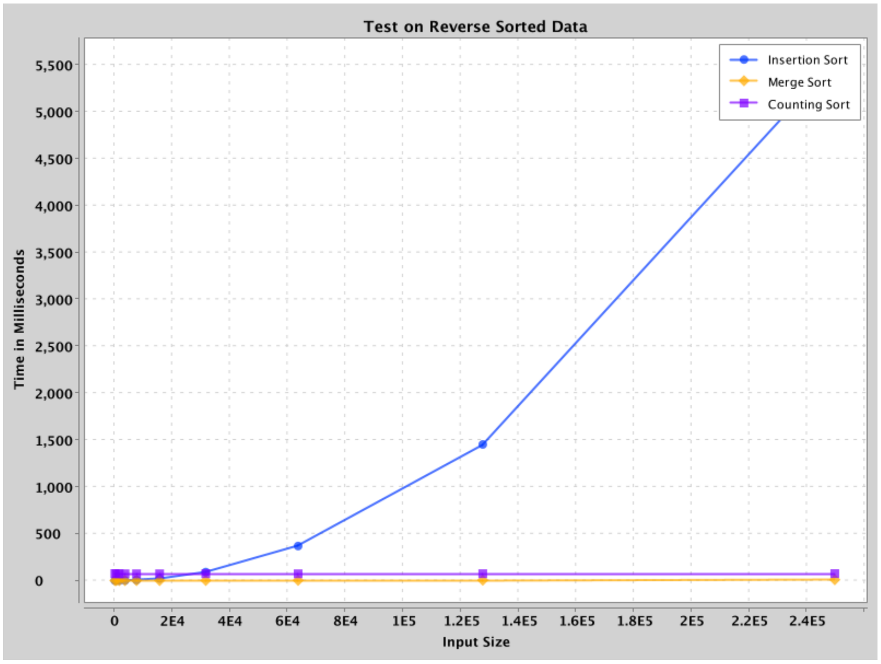

Figure 3: Time Chart for Reversely Sorted Data

Reverse-Sorted Data: Insertion Sort slows down significantly as data gets bigger, while Merge Sort and Counting Sort keep their speed, handling large datasets well. The times are higher than in the earlier tables, as this represents the worst-case scenario for sorting

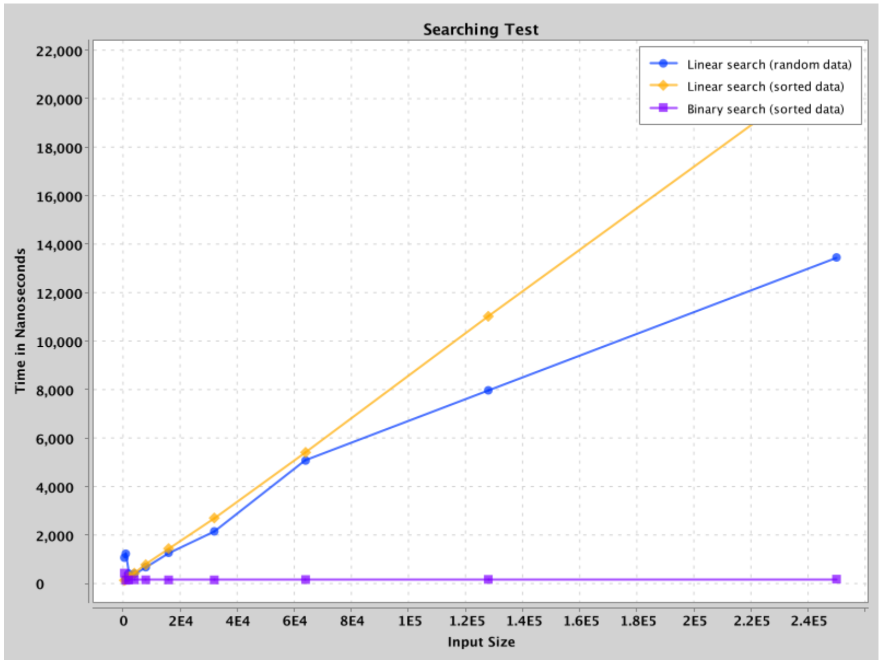

Figure 4: Time Chart for Search Times

The graph shows that Linear Search times increase with larger datasets, while Binary Search remains much quicker on sorted data, which aligns with its more efficient search process.

4 Notes

The study shows that sorting and searching algorithms perform as expected based on their theoretical complexities. Insertion Sort slows down with larger datasets, especially when data is reverse-sorted, due to its O(n^2) complexity. Merge Sort performs well across all data types with its O(n log n) complexity, while Counting Sort is stable but less consistent with sorted data. For searching, Linear Search's performance drops with larger datasets (O(n)), whereas Binary Search remains efficient on sorted data (O(log n)). This highlights the importance of selecting the right algorithm based on dataset characteristics for optimal efficiency.

References

- https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms
- https://www.codingninjas.com/studio/library/time-space-complexity-of-searching-algorithms
- https://www.youtube.com/watch?v=xJNOwfL7WNg