Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
İldeniz ÇELEBİ

Student Number:
b2210356013

1 Problem Definition

This assignment focuses on analyzing the complexity of sorting and searching algorithms. Specifically, it involves implementing and evaluating insertion sort, merge sort, counting sort, linear search, and binary search algorithms. The goal is to assess their performance under different scenarios, such as sorting random, sorted, and reversed sorted data, as well as searching for elements within these datasets. Understanding algorithm complexity aids in selecting efficient algorithms for various tasks in computer science.

2 Solution Implementation

In this section, we detail the implementations of sorting and searching algorithms, along with methods for reading data from a file. Additionally, the main program orchestrates the execution of these algorithms, records their performance metrics, and visualizes the results.

Sorting Algorithms

1. Insertion Sort: This algorithm sequentially inserts each element into its correct position within the sorted portion of the array. It has a time complexity of $O(n\hat{2})$ in the worst case.
2. Merge Sort: Employing a divide-and-conquer strategy, merge sort divides the array into halves, recursively sorts them, and merges them back together. It guarantees a time complexity of O(n log n).
3. Counting Sort: Suitable for sorting integers within a specific range, counting sort counts the occurrences of each element and arranges them accordingly. It runs in linear time, O(n + k), where k is the range of the input.

Searching Algorithms

1. Linear Search: This simple searching algorithm traverses the array sequentially until it finds the target element. It has a time complexity of O(n), making it suitable for small datasets or unsorted arrays.
2. Binary Search: Ideal for sorted arrays, binary search repeatedly divides the search interval in half until it finds the target element or determines its absence. It boasts a time complexity of O(log n), making it highly efficient for large datasets.

Conclusion: This comprehensive solution provides implementations of essential sorting and searching algorithms, accompanied by methods for data retrieval and performance analysis. By evaluating the algorithms' behavior under different conditions, it facilitates a deeper understanding of their efficiency and scalability.

2.1 Insertion Sort

```java
    public static int[] insertionSort(int[] A) {
        int n = A.length;
        for (int j = 1; j < n; j++) {
            int key = A[j];
            int i = j - 1;
            while (i >= 0 && A[i] > key) {
                A[i + 1] = A[i];
                i = i - 1;
            }
            A[i + 1] = key;
        }
        return A;
    }
```

2.2 Merge Sort

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

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

        return merge(left, right);
    }

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

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

2.3 Counting Sort

```java
        public static int[] countingSort(int[] A, int k) {
            int[] count = new int[k + 1];
            int[] output = new int[A.length];
            int size = A.length;

            Arrays.fill(count, 0);

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

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

            for (int i = size - 1; i >= 0; i--) {
                int j = A[i];
                count[j]--;
                output[count[j]] = A[i];
            }
            return output;
        }
```

2.4 Linear Search

```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 Binary Search

```java
    public static int binarySearch(int[] A, int x) {
        int low = 0;
        int high = A.length - 1;
        while (high - low > 1) {
            int mid = (high + low) / 2;
            if (A[mid] < x) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }
        if (A[low] == x) {
            return low;
        } else if (A[high] == x) {
            return high;
        }
        return -1;
    }
```

3 Results, Analysis, Discussion

Running time test results for sorting algorithms are given in Table 1. This table presents the execution times (in milliseconds) of sorting algorithms—Insertion Sort, Merge Sort, and Counting Sort—across different input sizes. The results demonstrate the performance of each algorithm under varying data complexities, including random, sorted, and reversed sorted datasets.

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	1.9	0.0	0.0	1.6	6.4	27.8	106.1	383.4	1540.1	7208.1
Merge sort	0.0	0.0	1.5	0.0	1.6	1.5	4.8	8.0	17.6	32.9
Counting sort	473.1	153.5	155.8	156.8	154.3	154.5	155.9	157.3	157.7	160.6
	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	1.7	1.6	5.9	8.5	18.6
Counting sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.6	200.5
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	3.2	11.1	43.3	186.3	847.9	3260.5	12886.1
Merge sort	0.0	0.0	0.0	0.0	1.6	0.0	3.1	6.4	14.2	29.6
Counting sort	213.7	221.5	174.8	175.3	163.9	175.4	166.5	158.7	182.0	161.3

Running time test results for search algorithms are given in Table 2. This table illustrates the execution times (in nanoseconds) of search algorithms—Linear Search and Binary Search—across varying input sizes. The results highlight the efficiency of each algorithm in locating target elements within arrays of different lengths.

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)	3068.0	1345.2	233.9	430.6	772.5	1044.8	2737.6	4932.3	14584.1	31505.7
Linear search (sorted data)	58.8	98.7	113.4	285.7	529.8	1241.3	3406.4	6908.7	16638.9	48058.3
Binary search (sorted data)	318.8	104.5	104.2	89.0	82.7	81.8	113.4	191.1	289.9	691.2

Complexity analysis tables are given in Table 3 and Table 4: These tables provide a comprehensive analysis of the time complexity of sorting and searching algorithms. They summarize the theoretical complexities of each algorithm and compare them to the observed execution times from the experimental data. By juxtaposing theoretical expectations with empirical results, these tables offer valuable insights into the practical efficiency of the algorithms.

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

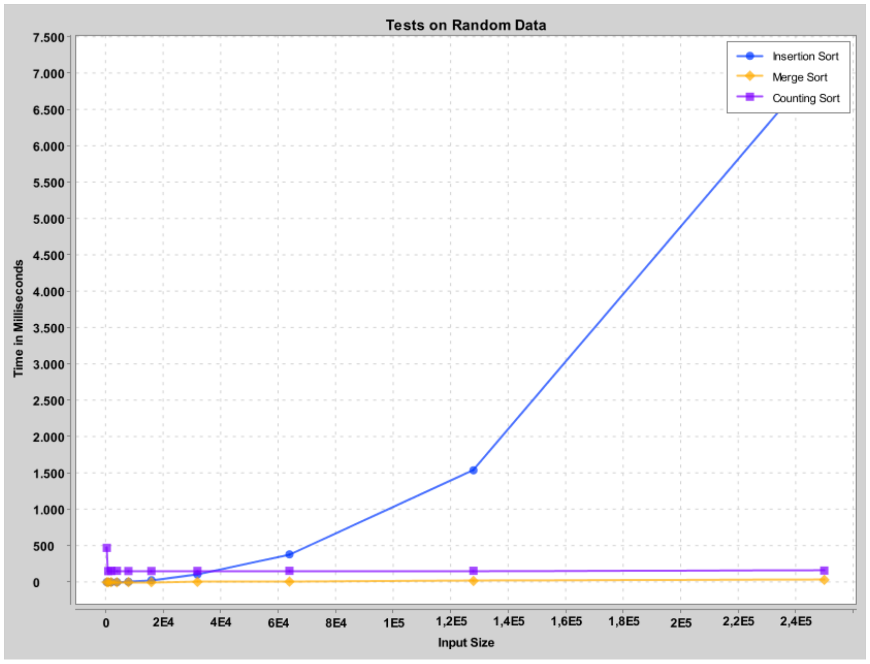

Figure 1: Plot of the tests on random data.

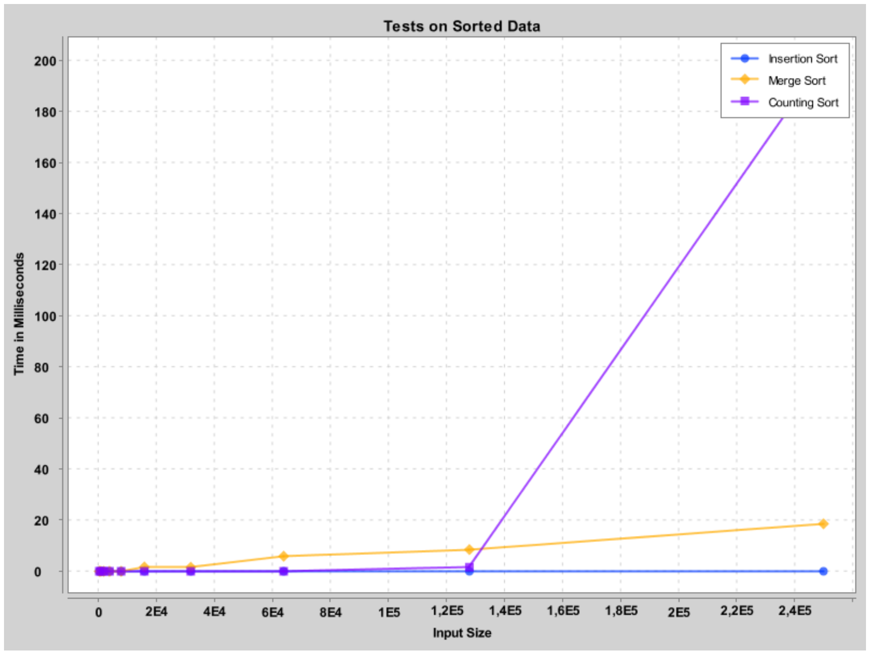

Figure 2: Plot of the tests on sorted data.

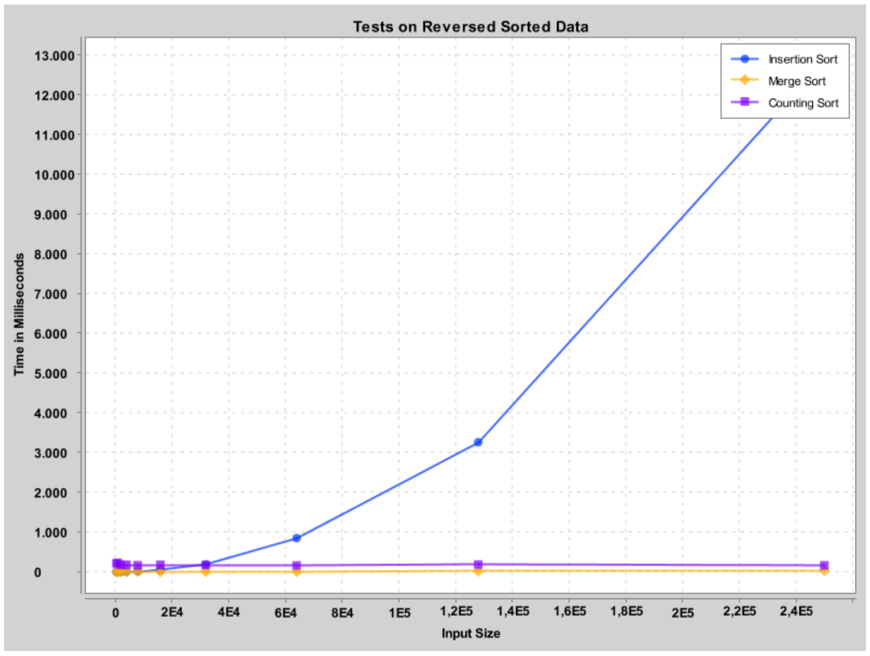

Figure 3: Plot of the tests on reversed sorted data.

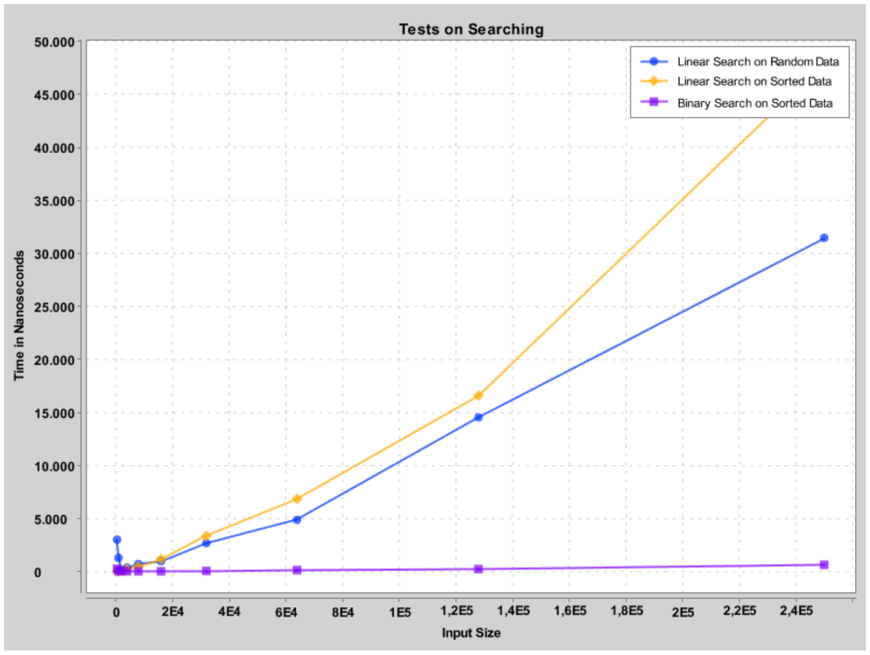

Figure 4: Plot of the tests on serching.

References

- https://www.geeksforgeeks.org/reverse-an-array-in-java/
- https://www.geeksforgeeks.org/java-program-for-program-to-find-largest-element-in-an-array/
- https://www.boardinfinity.com/blog/time-complexity-of-sorting-algorithms/
- https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms