Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

Student name:
Erdinç Arici

Student Number:
2210356035

1 Problem Definition

This project aims to compare the efficiency of these algorithms by analyzing the time complexity of each algorithm under different scenarios and evaluating their performance with datasets of various sizes. Through this comparative analysis, the project seeks to enhance students' algorithmic thinking skills and deepen their understanding of the complexity of sorting algorithms within the realm of computer science.

2 Solution Implementation

2.1 Merge Sort

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

    int mid = array.length / 2;
    int[] left = Arrays.copyOfRange(array, 0, mid);
    int[] right = Arrays.copyOfRange(array, mid, array.length);

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

private int[] merge(int[] left, int[] right) {
    int[] temp = new int[left.length + right.length];
    int l = 0, r = 0, t = 0;

    while (l < left.length && r < right.length) {
        if (left[l] <= right[r]) temp[t++] = left[l++];
        else temp[t++] = right[r++];

    }

    System.arraycopy(left, l, temp, t, left.length - l);
    System.arraycopy(right, r, temp, t, right.length - r);

    return temp;
}
```

This Java code implements the Merge Sort algorithm. The mergeSort method splits the array into two halves and recursively calls itself to sort each half. Then, the merge method merges the two sorted halves.

2.2 Insertion Sort

```java
int[] insertionSort(int[] array) {
    for (int i = 1; i < array.length; 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;
    }

    return array;
}
```

This Java code implements the Insertion Sort algorithm. It traverses the array from the second element onwards, placing each element in its correct position within the sorted subarray.

2.3 Counting Sort

```java
int[] countingSort(int[] array) {
    int max = Utilities.findMax(array);

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

    for (int i : array) {
        count[i]++;
    }

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

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

    return output;}
```

This Java code implements the Counting Sort algorithm. It first finds the maximum value in the array and then counts how many times each element is repeated. Then, it uses the counts to compute the correct position for each element. This way, it produces the sorted array as output.

2.4 Linear Search

```java
int linearSearch(int[] array, int x) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == x) return i;
    }

    return -1;
}
```

This Java code implements the Linear Search algorithm. It iterates through the array and checks if each element matches the target value. If a match is found, it returns the index of the element; otherwise, it returns -1.

2.5 Binary Search

```java
int binarySearch(int[] array, int x) {
    int low = 0;
    int high = array.length - 1;

    while (high - low > 1) {
        int mid = (high + low) / 2;

        if (array[mid] < x) low = mid + 1;
        else high = mid;
    }

    if (array[low] == x) return low;
    else if (array[high] == x) return high;

    return -1;
}
```

This Java code implements the Binary Search algorithm. It searches for a target value 'x' within a sorted array. The algorithm maintains two pointers, 'low' and 'high', which represent the range of indices to search within. It iteratively narrows down this range until the target value is found or the range becomes empty. If the target value is found, it returns the index of the target element; otherwise, it returns -1.

3 Results, Analysis, Discussion

Table 1: Results of time tests for sorting runs performed for variable 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	4	18	70	277	1167	5055
Merge sort	0	0	0	0	0	1	2	5	11	24
Counting sort	120	76	76	78	79	78	78	83	85	88
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	1
Merge sort	0	0	0	0	0	0	1	2	4	11
Counting sort	77	76	77	78	79	78	80	81	81	85
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	2	8	34	137	560	2333	9225
Merge sort	0	0	0	0	0	0	1	2	4	10
Counting sort	76	77	78	79	78	77	79	81	83	87

Table 2: Results of time tests for searhing runs performed for variable input sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	834	1120	188	300	513	852	1698	4003	6678	11977
Linear search (sorted data)	1128	837	210	391	658	1139	2264	4891	9738	19368
Binary search (sorted data)	388	173	197	112	117	103	116	163	152	177

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

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

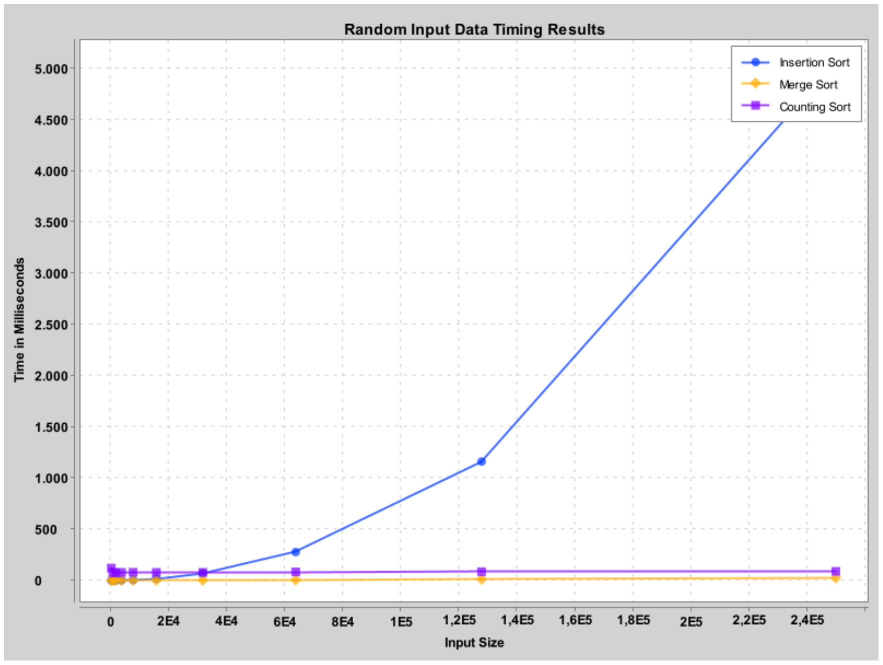

Figure 1: Sort with Random Data

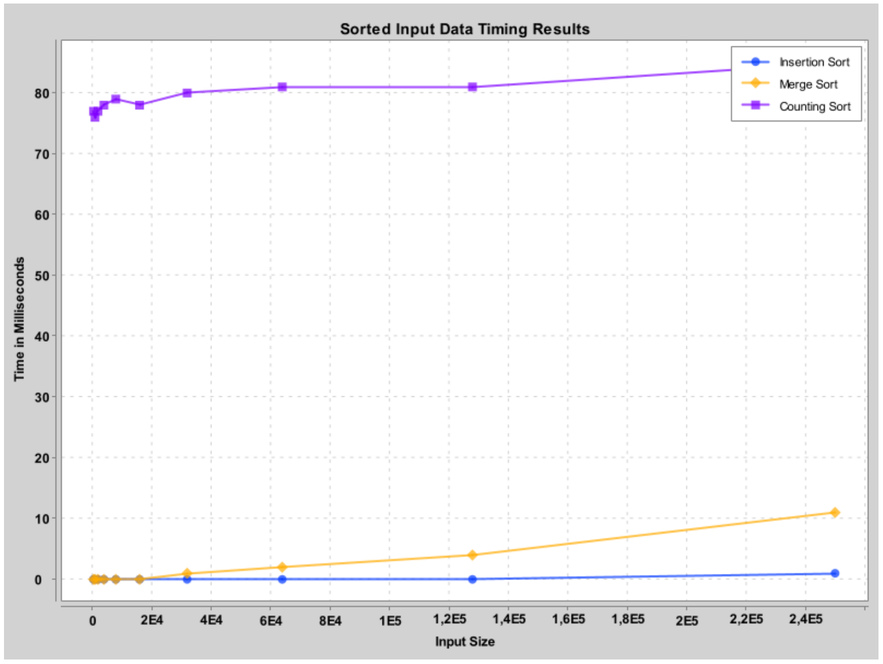

Figure 2: Sort with Sorted Data

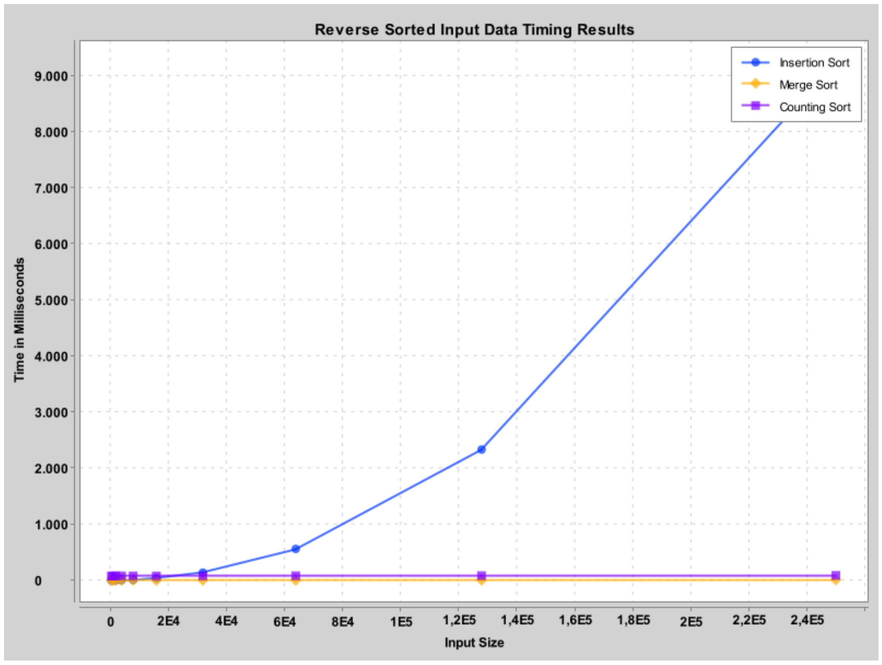

Figure 3: Sort with Reverse Sorted Data

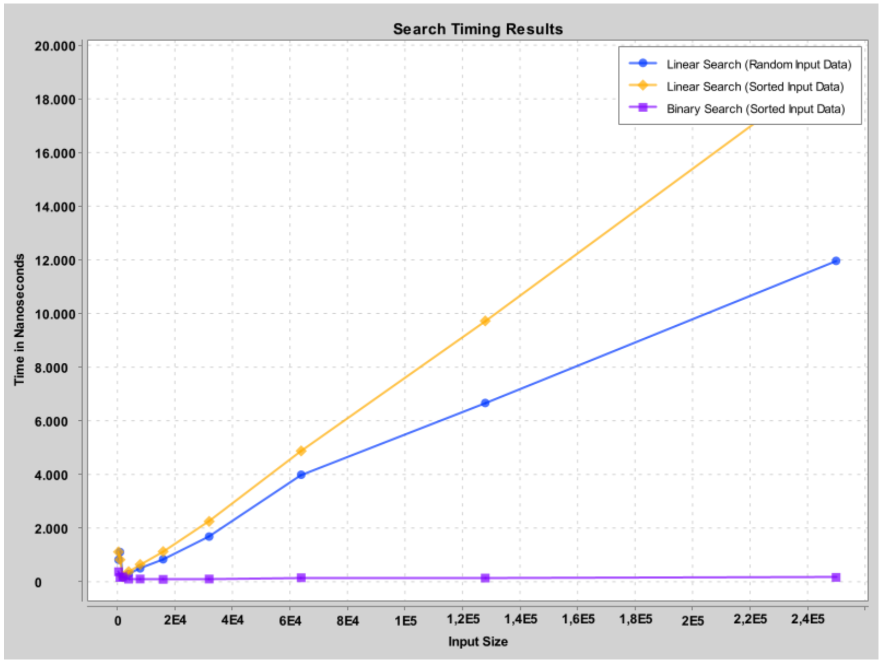

Figure 4: Search Graph

Sorting algorithms are fundamental algorithms used to organize data according to a specific criterion. Choosing the right sorting algorithm based on the size of the data is crucial for both time and performance considerations. For a few thousand data points, insertion sort is often the most logical choice as this algorithm can be effective on small data sets. As the number of data points increases, more efficient algorithms like merge sort may become a better option. For very large data sets, specialized algorithms like counting sort may offer the best performance.

For more accurate results in the search algorithms, I searched for a different value in each search process. As a result,binary search provides the best results for any size of data as it requires sorted data and has logarithmic time complexity.