Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 17, 2024

Student name:
Özge Bülbül

Student Number:
b2220765008

1 Problem Definition

Efficiency Analysis of Sorting and Searching Algorithms:

The objective of this assignment is to conduct a comprehensive examination of sorting and searching algorithms, focusing on their efficiency in terms of time complexity, space complexity, and stability. Sorting algorithms are crucial for organizing data efficiently, while searching algorithms aid in locating specific elements within datasets. The assignment aims to analyze various sorting and searching algorithms, categorize them based on their efficiency metrics, and compare their performance under different scenarios.

2 Solution Implementation

There were 3 sorting and 2 searching algorithms to be implemented: insertion sort, merge sort, counting sort, linear search and binary search. Their pseudo codes were given, so the functions were written by me accordingly.

2.1 Sorting Algorithm 1: Insertion Sort

My java code for insertion sort:

```
0  public static int[] insertionSort(int[] array){
1          for (int j = 1; j < array.length; j++) {
2              int key = array[j];
3              int i = j - 1;
4              while (i >= 0 && array[i] > key) {
5                  array[i + 1] = array[i];
6                  i = i - 1;
7              }
8              array[i + 1] = key;
9          }
10         return array;
11     }
```

The insertion sort algorithm maintains a sorted subarray in the input array and iteratively inserts new elements into the sorted subarray, resulting in a sorted array at the end of the process.

2.2 Sorting Algorithm 2: Merge Sort

My java code for merge sort:

```
0  public static int[] merge(int[] left, int[] right) {
1          int[] result = new int[left.length + right.length];
2          int leftIndex = 0, rightIndex = 0, resultIndex = 0;
3          while (leftIndex < left.length && rightIndex < right.length) {
4              if (left[leftIndex] <= right[rightIndex]) {
5                  result[resultIndex++] = left[leftIndex++];
```

```
            } else {
                result[resultIndex++] = right[rightIndex++];
            }
        }
        while (leftIndex < left.length) {
            result[resultIndex++] = left[leftIndex++];
        }
        while (rightIndex < right.length) {
            result[resultIndex++] = right[rightIndex++];
        }
        return result;
    }
    public static int[] mergeSort(int[] array){
        if (array.length <= 1) {
            return array;
        }
        int[] left = Arrays.copyOfRange(array, 0, (array.length / 2));
        int[] right = Arrays.copyOfRange(array, (array.length / 2)+1, array.
           length);
        left = mergeSort(left);
        right = mergeSort(right);
        return merge(left, right);
    }
```

The merge sort algorithm recursively divides the input array into smaller halves until it reaches base cases (arrays of size 0 or 1), sorts them individually, and then merges them back together to produce the final sorted array.

2.3 Sorting Algorithm 3: Counting Sort

My java code for counting sort:

```
public static int[] countingSort(int[] array){
        int maxElement = array[0];
        for (int j = 1; j < array.length; j++) {
            if (array[j] > maxElement) {
                maxElement = array[j];
            }
        }
        int[] count = new int[maxElement + 1];
        int[] output = new int[array.length];
        for (int i = 0; i < array.length; i++) {
            count[array[i]]++;
        }
        for (int i = 1; i < count.length; 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;
    }
```

Counting Sort works by counting the occurrences of each element in the input array, calculating the cumulative sum of counts, and then placing each element in its correct sorted position in the output array based on the cumulative count. This algorithm has a time complexity of O(n + k), where n is the number of elements in the input array and k is the range of input values. The space complexity of the algorithm is O(k), where k is the maximum element in the input array.

2.4 Searching Algorithm 1: Linear Search

My java code for linear search:

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

The linear search algorithm checks each element of the array one by one until it finds the target element or reaches the end of the array. It has a time complexity of O(n), where n is the number of elements in the array, as it may need to traverse the entire array in the worst-case scenario.

2.5 Searching Algorithm 2: Binary Search

My java code for binary search:

```
public static 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;
    }
```

The binary search algorithm repeatedly divides the search range in half until the target element is found or the search range is reduced to zero. It has a time complexity of O(log n), where n is the number of elements in the array, making it significantly more efficient than linear search for large sorted arrays.

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	1	4	18	76	291	1183	4756
Merge sort	0	0	0	0	1	1	5	13	10	22
Counting sort	293	120	121	165	144	108	138	115	113	113
	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	1	1	4	7
Counting sort	103	100	119	117	101	97	122	105	105	127
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	2	8	36	148	605	2463	9657
Merge sort	0	0	0	0	0	0	0	1	3	6
Counting sort	123	129	115	173	145	137	165	128	124	116

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)	3659	1693	197	340	642	1221	2750	5119	10089	19912
Linear search (sorted data)	72	94	132	201	367	633	1212	2426	4736	9335
Binary search (sorted data)	250	152	185	208	140	107	110	120	128	138

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	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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	$O(1)$

Insertion sort is an in-place sorting algorithm, meaning it doesn't require any extra space other than the input array. Its space complexity is O(1), as it sorts the elements within the given array itself.

Merge sort is a classic example of a divide-and-conquer algorithm. While its time complexity is O(n log n), its space complexity is O(n) due to the need for temporary storage when merging the subarrays.

Counting sort is a non-comparison based sorting algorithm suitable for sorting integers within a specific range. Its auxiliary space complexity is O(n + k), where n is the number of elements in the input array and k is the range of input. It requires additional two arrays with sizes n and k+1.

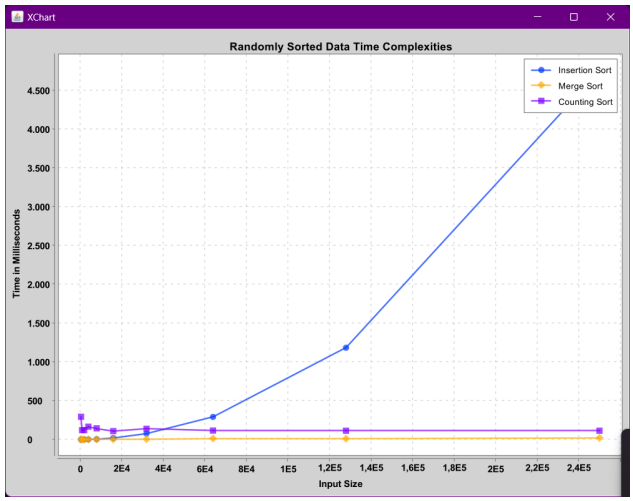

Figure 1: Time Chart for Random Data

Random Data:

For insertion sort, the average time increases as the number of inputs increase. This method is highly inefficient for large inputs since it has a high time complexity. Mergesort shows consistently low average time (0) for all dataset sizes. Counting sort also demonstrates a consistent trend in average time as the dataset size increases. It doesn't get affected by input size as much but performs badly even with a smaller dataset. This is expected as counting sort has a time complexity of $O(n + k)$.

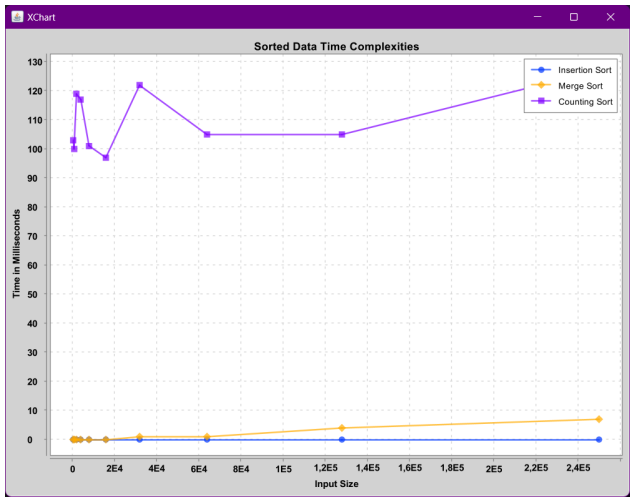

Figure 2: Time Chart for Sorted Data

Sorted Data:

Insertion sort is efficient with average times remaining consistently low (0) for all dataset sizes. Merge sort also maintains efficiency, although there's a slight increase in average time for larger dataset sizes. Counting sort however shows a relatively higher and inconsistent average time.

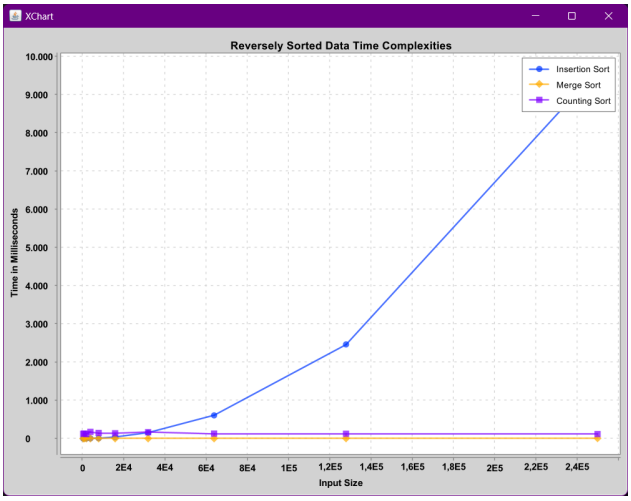

Figure 3: Time Chart for Reversely Sorted Data

Reversely Sorted Data:

Insertion sort's average time increases for larger dataset sizes. Mergesort maintains its efficiency, with low average times similar to those observed for random and sorted data. Counting sort again remains its stability, times don't change much as the dataset gets larger. Obviously, times are higher than the previous two tables, since this is the worst case for sorting.

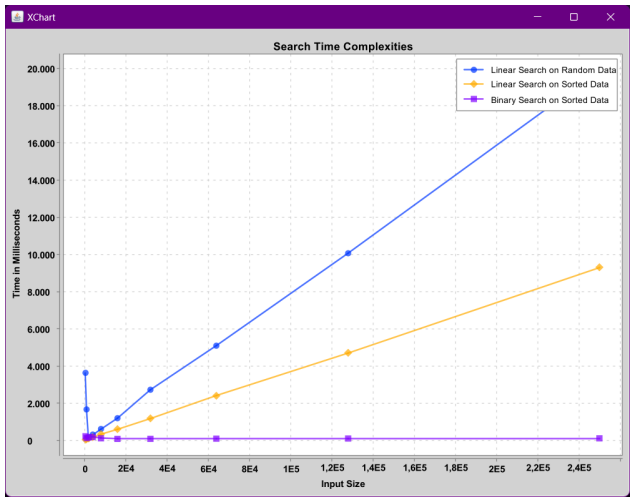

Figure 4: Time Chart for Search Times

Linear Search on Random Data:

As the dataset size increases, the time taken for linear search on random data also increases. This increase is expected as linear search has a time complexity of O(n), where n is the number of elements in the dataset. The time taken for linear search on random data grows rapidly with the increase in dataset size, indicating that linear search is not suitable for large random datasets.

Linear Search on Sorted Data:

For linear search on sorted data, the time taken remains relatively low compared to random data. This is because linear search on sorted data can terminate early if the target element is found, leading to faster search times compared to random data. The time taken for linear search on sorted data increases with the increase in dataset size, but it remains lower than the time taken for linear search on random data.

Binary Search on Sorted Data:

Binary search on sorted data demonstrates significantly lower and more consistent search times compared to linear search. Binary search has a time complexity of O(log n), where n is the number of elements in the dataset. This logarithmic time complexity results in faster search times, especially for large datasets. The time taken for binary search remains relatively constant even as the dataset size increases, indicating its efficiency for sorted data.

4 Notes

For the search analysis part, since the searched item was randomly selected, I believe the results in that section may differ as the experiment gets repeated.

In conclusion, sorting and searching algorithms vary in efficiency. Insertion sort struggles with large random datasets, while merge sort remains consistent but requires more space. Counting sort is efficient for datasets with a limited range of values but less so for those with a wider range. Linear search is slow for large random datasets but faster for sorted data. Binary search consistently outperforms linear search for sorted datasets. The choice of algorithm depends on factors like dataset size and nature.