Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:	*Student number:*
Sudenaz Yazıcı	2210356008

1 Problem Definition

In this project, we are given some sorting and searching algorithms. Since a given program's and software's efficiency depend on how we implement a specific algorithm, analysis of algorithms based on time/space is important for programmers. To do this analysis, firstly we implement all algorithms. Then we test all of them on different data(random data, sorted data, reversely sorted data) and make charts using Xchart. Lastly, we compare our results to theoretical common knowledge.

2 Solution Implementation

Here, you can see my sorting and searching algorithms that I did according to given pseudocodes.

2.1 Insertion Sort

Java code for insertion sort algorithm:

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

2.2 Merge Sort

For merge sort, at first I did not use indexes to follow the location of the index of the arrays(arrayIndexA, arrayIndexB). I tried to copy new array every time to remove the first element. But this method took a lot of time and space than intended.

Java code for merge sort algorithm:

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

```java
public int[] Merge(int [] arrayA, int [] arrayB) {
    int[] arrayC = new int[arrayA.length + arrayB.length];
    int arrayIndex = 0;
    int arrayIndexA = 0;
    int arrayIndexB = 0;

    while (arrayIndexA != arrayA.length && arrayIndexB != arrayB.length) {
        if(arrayA[arrayIndexA] > arrayB[arrayIndexB]) {
            arrayC[arrayIndex] = arrayB[arrayIndexB];
            arrayIndex++;
            arrayIndexB++;
        } else {
            arrayC[arrayIndex] = arrayA[arrayIndexA];
            arrayIndex++;
            arrayIndexA++;
        }
    }

    while (arrayIndexA < arrayA.length) {
        arrayC[arrayIndex] = arrayA[arrayIndexA];
        arrayIndex++;
        arrayIndexA++;
    }

    while(arrayIndexB < arrayB.length) {
        arrayC[arrayIndex] = arrayB[arrayIndexB];
        arrayIndex++;
        arrayIndexB++;
    }
    return arrayC;
}
```

2.3 Counting Sort

Java code for counting sort algorithm:

```java
public int[] CountingSort(int [] array) {
    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];
    Arrays.fill(count, 0);
    int[] output = new int[array.length];
    int size = array.length;

    for(int i=0; i<size; i++) {
        count[array[i]]++;
    }
    for(int i=1; i<k+1; i++) {
        count[i] = count[i] + count[i-1];
    }
    for(int i=size-1; i>=0; i--) {
        count[array[i]]--;
        output[count[array[i]]] = array[i];
    }
    return output;
}
```

2.4 Linear Search

Since searching relatively takes less time than sorting, instead of taking average of 10 operations, we take 1000 operations(in nanoseconds).

Java code for linear search algorithm:

```java
public 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;
}
```

2.5 Binary Search

Same as linear search, since searching relatively takes less time than sorting, instead of taking average of 10 operations, we take 1000 operations(in nanoseconds).

Java code for binary search algorithm:

```java
public 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;
}
```

3 Results, Analysis, Discussion

Here, you can see the charts my program generated:

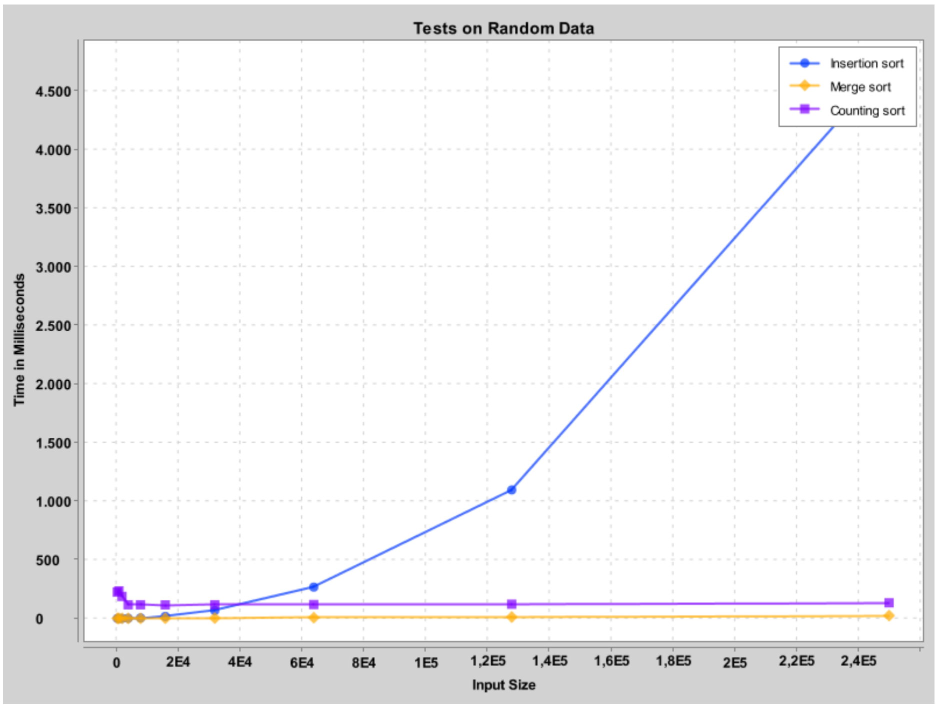

Figure 1: Plot of the Test on Random Data

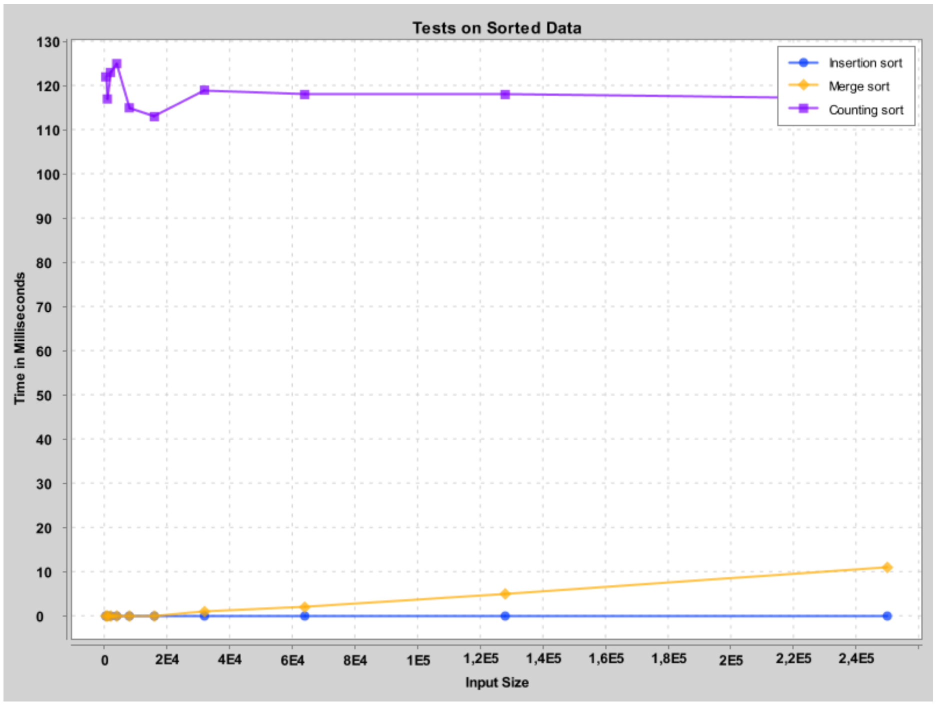

Figure 2: Plot of the Test on Sorted Data

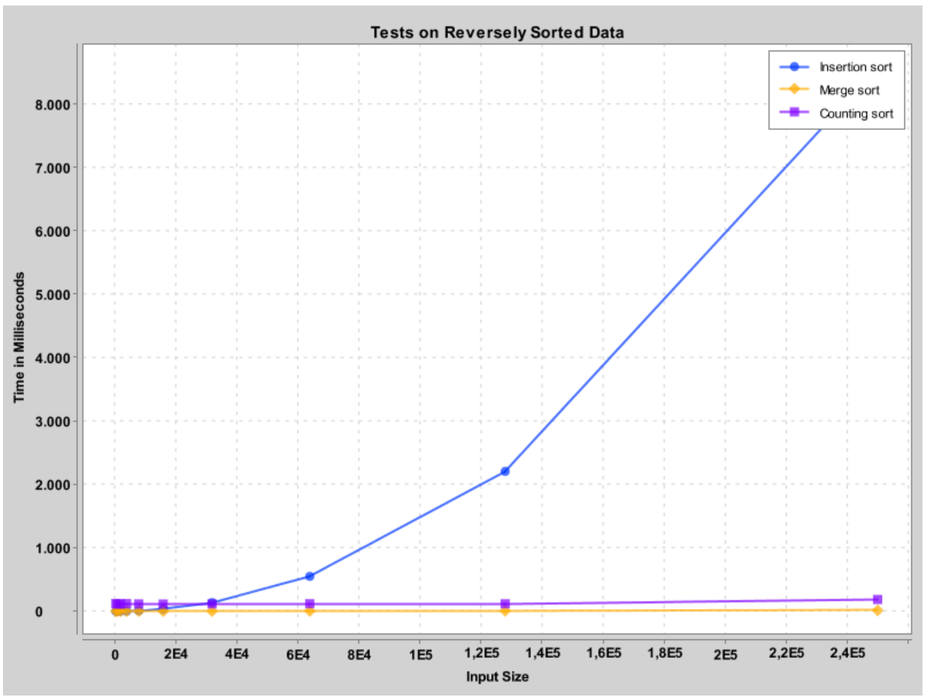

Figure 3: Plot of the Test on Reversely Sorted Data

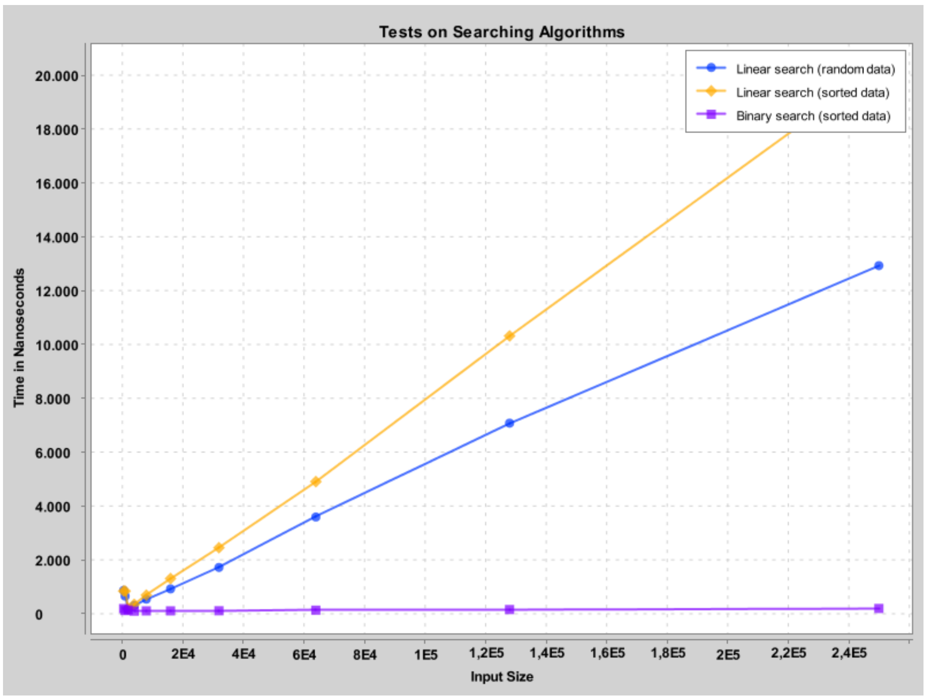

Figure 4: Plot of the Test on Searching Algorithms

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	0	0	1	4	17	67	267	1096	4729
Merge sort	0	0	0	0	0	1	3	6	12	24
Counting sort	224	235	187	116	116	115	116	118	121	133
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	2	5	11
Counting sort	122	117	123	125	115	113	119	118	118	117
Reversely Sorted Input Data Timing Results in ms										
Insertion sort	0	0	0	2	8	33	136	550	2207	8580
Merge sort	0	0	0	0	0	0	1	3	6	13
Counting sort	121	113	115	120	115	113	115	118	117	185

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)	882	670	199	316	556	937	1736	3607	7094	12945
Linear search(sorted data)	860	856	211	365	706	1322	2468	4923	10333	20313
Binary Search(sorted data)	207	133	158	111	121	122	126	148	171	211

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best case**	**Average case**	**Worst case**
Insertion sort	Ω(n)	Θ(n^2)	O(n^2)
Merge sort	Ω(n log n)	Θ(n log n)	O(n log n)
Counting sort	Ω(n+k)	Θ(n+k)	O(n+k)
Linear search	Ω(1)	Θ(n)	O(n)
Binary Search	Ω(1)	Θ(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)

Table 3 is filled according to theoretical information.

For Table 4:

- In insertion sort, we use the original array and only exchange some elements. Since there is no need to use extra space, the auxiliary space complexity is O(1).

- int[] left = Arrays.*copyOfRange*(array, 0, n/2);
  int[] right = Arrays.*copyOfRange*(array, (n/2), n);

  In these two lines from the merge sort algorithm, we copy the original array into two extra arrays. Since we are using n extra space, the auxiliary space complexity of merge sort is O(n).

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

  We can see that from these two lines in counting sort algorithm, we are creating n(array.length) and k(max element of array) sized arrays, so the auxiliary space complexity is the addition of these two(n+k).

- Looking at the linear search and binary search algorithms, we can see that we are using original arrays and not generating extra arrays. So their auxiliary space complexity is O(1).

What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted/searched?

- For insertion sort   Best case: Sorted input data(no need to exchange, just traverses)

  Average case: Random input data

  Worst case: Reversely sorted data(traverses entire array every time)

- For merge sort   Best case: Sorted input data(less swapping)

  Average case: Reversely sorted input data

  Worst case: Random data

- For counting sort, the results are really close for all random, sorted and reversely sorted data.

Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?

Here are the differences between my algorithms and theoretical complexities:

- For insertion sort, the best case is that the array is already sorted. Theoretically this must take O(n) time. In my algorithm, insertion sort is not exchanging any elements, so the passed time is very small. Taking ten operations' average leads the time to be zero.

- For linear search, both for random and sorted data, the increase rate seems correct. However theoretically, the time passed in searching for random data should be more than for sorted data. This could be because of data characteristics since we are taking the data from the beginning of the file each time.

- Theoretically, the time passed for the merge sort algorithm must be greater than the time passed for the counting sort algorithm. In my tests, counting sort takes more time. The reason could be the data we are testing(we read data only from the beginning). If the maximum value of the array(k) is greater than the length of the array, it is normal that counting sort takes more time.

- For counting sort, the charts starts a bit higher than the rest of the tests. This could be because of the computer used for testing.

REFERENCES

1) https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/#:~:text=The%20worst%2Dcase%20(and%20average,O(n)%20time%20complexity.
2) https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/
3) https://www.geeksforgeeks.org/counting-sort/
4) https://www.geeksforgeeks.org/linear-search/
5) https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/