HACETTEPE UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BBM204 SOFTWARE PRACTICUM II – 2024 SPRING

Programming Assignment 1

March 22, 2024

Student Name: UYGAR ERSOY

Student Number: 2220356114

1- Problem Definition

Storing and analyzing data has been crucial in computer science since the origin of the area. Due to the information age and increasing data, the role of sorting these data has gained more attention than ever. Generally, sorting algorithms can be divided into two subparts. Firstly, comparison-based sorting algorithms can be inspected. A few examples of this type of sorting algorithm can be given as follows: merge sort, insertion sort, selection sort, bubble sort and so on. In this type of algorithm, elements to be sorted, are compared with each other to determine their respective place in the array. In this problem set, merge sort and insertion sort are going to be analyzed under this category. Secondly, non-comparison-based can be inspected. Most know algorithm in this category is called counting sort algorithm. To determine the place of an item in the sorted array, it does not require that item to be compared with rest of the elements. In this problem set, counting sort is going to analyzed. Merge sort, insertion sort and counting sort is going to be analyzed with random, sorted and reversely sorted data sets, and their theoretical running and space complexities will be compared with the obtained real data values.

Additionally, if you have data and if you can sort that data, you want to be able to search for a value in that data to further the analysis and extract value from it. Some of the most-known algorithms for searching are linear and binary search. In this experiment, these algorithms are going to be tested with random and sorted data and their obtained real life running time values will be compared with theoretical values.

2- Solution Implementation

In this experiment, a large set of sample data is going to be used for sorting and searching operations. A csv file with over 250000 values is given to be analyzed. Integer values are extracted from the 7th column of the csv file to be used. After obtaining the values, the given data is partitioned into ten smaller parts ranging in size from 500 to 250000. This process is done to observe the running time of mentioned algorithms on changing sizes of arrays. To obtain the partitions, the following code on Figure 1 is used.

```
static int[] partitionValues = {500, 1_000, 2_000, 4_000, 8_000, 16_000, 32_000, 64_000, 128_000, 250_000};

1 usage
public static ArrayList<int[]> part (int[] arr) {
    ArrayList<int[]> res = new ArrayList<>();

    for (int i = 0; i < partitionValues.length; i++) {
        int[] temp = new int[partitionValues[i]];
        for (int j = 0; j < partitionValues[i]; j++) {
            temp[j] = arr[j];
        }
        res.add(temp);
    }
    return res;
}
```

Figure 1: Partition operation of the main array in smaller subarrays

From the beginning of the main array, each subarray is created according to the ranging sizes given as partitionValues as above. These arrays are stored inside of an arraylist for the choice of easiness.

After obtaining partitioned arrays, sort and search operations can be implemented on these arrays. Sort operations are done with merge sort, insertion sort and counting sort. Search operations are done with binary and linear search.

Firstly, sorting algorithms are going to be tested in the order as follows: insertion sort, merge sort, counting sort.

2.1- Insertion Sort

Insertion sort is a comparison-based sorting algorithm. Theoretically, it has a worst case scenario of $O(n^2)$, a best case scenario of $O(n)$ and $O(n^2)$ average running time complexities. In insertion sort, algorithm goes over each item and does the following, picks the item, and looks for an item from the initial item's position to the beginning of the array until it finds an item that is smaller than the initial item. When the algorithm finds that position, it puts that item there and continues the same process. Until it finds that item, algorithm pushes the bigger item to the right 1 unit to create a gap to put the initial item. Implementation of insertion sort is given in Figure 2 below.

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;

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

Figure 2: Implementation of insertion sort algorithm

As it can be seen, it compares each item with each other and makes swapping in place, which means that this algorithm uses constant space. Insertion sort has a O(1) space complexity for each case. In the worst case scenario, in which a reversely sorted array is given to the algorithm to be sorted, insertion sort has to compare each item with previous items in the array, shift every value until finding the corresponding position for that item. Eventually, after finding the correct position for an item, it places it to the correct position, which ends up with quadratic running time complexity. In the base case, in which a sorted array is given to the algorithm to be sorted, insertion

sort goes through the array only once. The inner while loop in Figure 2 never evaluates to be true and algorithm traverse the array only once, which causes the algorithm to run on O(n) runtime complexity in this case. In the average case, similar to the worst case, algorithm compares most of the items till the beginning of the array, which causes insertion sort to run on quadratic time at average case.

2.2- Merge Sort

Merge sort is a comparison-based divide and conquer sorting algorithm. It runs on O(nlogn) time complexity at best, average, and worst case scenarios and has an O(n) space complexity for each case. Good thing about merge sort is, its O(nlogn) runtime complexity is guaranteed for every scenario, which makes it desirable to use when sorting. Its working principle is as follows: when an array is given to the algorithm, if it has a length over 1, divide the array into two parts, called left and right. Find the middle item of the given array and add items till the middle item to the left array. From the middle item to the last item of the given array, add them to the right array. Recursively call merge sort function on this algorithm repeatedly. In Figure 3, implementation of this logic can be observed.

```
public static void mergeSort(int[] arr) {
    if (arr.length < 2) {
        return;
    }

    int middle = arr.length / 2;

    int[] left = new int[middle];
    int[] right = new int[arr.length - middle];

    for (int i = 0; i < middle; i++) {
        left[i] = arr[i];
    }

    for (int j = middle; j < arr.length; j++) {
        right[j-middle] = arr[j];
    }

    mergeSort(left);
    mergeSort(right);
    merge(left, right, arr);
}
```

Figure 3: Implementation of merge sort algorithm

This repeated call for the merge sort function is the divide part of the divide and conquer concept. To conquer, we have to merge these divided left and right arrays into the main one.

mergeSort function in Figure 3, will be called recursively until the given parameter has a length smaller than 2. After each left and right subarrays have a length of 1, merge algorithm is going to be called and, this is the conquer part. In this part, divided smaller subarrays are going to be merged, hence the name merge sort, and due to the recursive calls, main array will be sorted at the hand with these divided subarrays. In Figure 4, algorithm for merge function can be seen.

```
public static void merge(int[] left, int[] right, int[] arr) {
    int i = 0; int j = 0; int k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        }
        else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    while (i < left.length) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while (j < right.length) {
        arr[k] = right[j];
        j++;
        k++;
    }
}
```

Figure 4: Implementation of merge function logic

Merge function has three parameters called, right, and arr. Arr is the final main array given to the merge sort function recursively. Let's say main call to the merge sort is with arr [4,1,3,2]. merge function will be called with left = [4], right = [1] and arr = [4,1] for the first time. What merge function does is, according to the comparison based logic, it puts them to their correct sorting order on the arr array. Since java does the memory handling by itself, function has a void return type and it sorts the part of the given original array. Hence, for the given example, arr will be [1, 4] due to the first while loop. Other two while loops guarantees that there won't be any item left on either left or right array. Recursive calls will sort the whole array according to the given logic. Dividing causes the logn part and merging causes the n part in the complexity analysis, which makes merge sort run on O(nlogn) time. Creation of left and right arrays in the merge sort function causes merge sort to have O(n) space time complexity.

2.3- Counting Sort

Counting sort is a non-comparison-based sorting algorithm. Best we can do with comparison-based sorting algorithms at the worst case scenarios is O(nlogn) runtime. However,

counting sort can run on linear time while sorting with its criteria and special cases. Instead of comparing items, counting sort applies a different logic. Firsty, it creates an array with the length of (max + 1) items all equal to zero. In this array, counts of each item in the original array is stored.

Starting from index 1, let's say i^{th} index, value at $(i-1)^{th}$ index is added to the value at the i^{th} index. The reason for this is to obtain the cumulative sum of the items in the original array. Starting from the end of the original array, cumulative sum is checked at the count array for the value at the current index of the original array. To handle duplicates and find the correct position of the item, 1 is subtracted from the cumulative count in that index and it is placed the result array. After applying this process for each item, result array will be the sorted array of the main array. Implementation of this algorithm is given in Figure 5 below.

```
public static int[] countingSort(int[] arr, int max) {
    int[] count = new int[max+1];
    int[] output = new int[arr.length];

    for (int i = 0; i < max + 1; i++) {
        count[i] = 0;
    }

    for (int i = 0; i < arr.length; i++) {
        count[arr[i]] += 1;
    }

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

    for (int i = arr.length - 1; i > -1; i--) {
        count[arr[i]]--;
        output[count[arr[i]]] = arr[i];
    }
    return output;
}
```

Figure 5: Implementation of counting sort algorithm

If we call the length of arr as n, and length of count as m running time complexity for counting sort will be O(n+m) for best, worst, and average cases due to the usage of for loops in the algorithm above. In every case, these loops have to run, which concludes this algorithm to have O(m+m) runtime. Also, count and output arrays use extra space, which makes counting sort algorithm to have O(n+m) space complexity as well. If we have a small range of items to be sorted. In that case, it can run on linear time and have better performance than comparison-based sorting algorithms.

2.4- Linear Search

Linear Search is the most basic search algorithm one can apply. It traverses the whole array until it finds it in the array. If the searched item exists in the array, it returns the index. If it reaches the end of the array without finding the item, it returns -1. In runs in O(n) runtime complexity in worst case, which is not finding the item in the array. It has Ω(1) runtime complexity at best case, which is finding the searched element at the first index of the given array. In average case, it still runs in O(n) time because, until finding the item in the array, it still has to look for most of the items in the array. It does not require additional space and O(1) space time complexity. Simple implementation of the algorithm is given at Figure 6 below.

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

Figure 6: Implementation of linear search algorithm

2.5- Binary Search

Binary search algorithm is another search algorithm which excels when the given data is already sorted, actually, for binary search to work, it has to be sorted. Firstly define two pointers called left and right. Left has a value of zero and right has a value of size of the array minus 1. Until left is smaller or equal than the right pointer, pick the middle element of the array. Check if that value is the searched value and return the index if it is. Otherwise, if the middle element is smaller than the searched value, update the left pointer and we can get rid of the left half of the array because, searched value will be on the right side of the array no matter what due to that array being sorted. The same work for the right side of the array when the middle element is bigger than the searched item. Continuously applying this operations, two outcomes might appear: either the item is found and index is returned, or item does not exist in the array and left becomes bigger than right, which breaks the loop. In the latter, -1 is returned. Implementation of this algorithm can be seen at Figure 7. Binary search improves the runtime drastically compared to linear search. In the worst case, it runs on O(logn) time, which is item does not exist in the array. In the best case, it runs on O(1) time, which is the searched item is the middle item. Also it runs on O(logn) time for average case, cause it has to divide the array to halves until finding the item. It does not require additional space and runs on O(1) space complexity.

```java
public static int binarySearch (int[] arr, int val) {
    int left = 0; int right = arr.length - 1;

    while (left <= right) {
        int middle = left + (right - left) / 2;

        if (arr[middle] == val) {
            return middle;
        }
        if (arr[middle] > val) {
            right = middle - 1;
        }
        else {
            left = middle + 1;
        }
    }
    return -1;
}
```

Figure 7: Implementation of binary search algorithm

2.6- Analysis

Using the algorithms mentioned and the partitioned data set, analysis of the runtime of these algorithms are going to be inspected. Using the code in Figure 1, an arraylist with arrays with different partition sizes are obtained to be tested. To be more precise, each sorting algorithm is going to be run 10 times on each partition and their running time will be average on each partition. In Figure 8, code for this operation is given.

```java
ArrayList<int[]> partedResRandom1 = Utils.part(res);

long[] insertionSortRandomTimes = InsertionSortTest.test(partedResRandom1);
Utils.print(insertionSortRandomTimes, sortSearch: "InsertionSort", secType: "milliseconds", randSortRev: "random data");
long[] mergeSortRandomTimes = MergeSortTest.test(partedResRandom1);
Utils.print(mergeSortRandomTimes, sortSearch: "MergeSort", secType: "milliseconds", randSortRev: "random data");
long[] countingSortRandomTimes = CountingSortTest.test(partedResRandom1);
Utils.print(countingSortRandomTimes, sortSearch: "CountingSort", secType: "milliseconds", randSortRev: "random data");
```

Figure 8: Runtime value analysis of sorting algorithms

For each sorting algorithm, their respective test functions are called and their values on sorting each partition is obtained in a long array. These values are printed to the console using Utils class and later going to be used for plotting purposes.

Each sorting algorithm uses their respective test function from different classes, but the logic behind is the similar. To be an example, test function for merge sort algorithm is given in Figure 9 below.

```java
public static long[] test (ArrayList<int[]> arr) {
    long[] res = new long[10];
    for (int i = 0; i < arr.size(); i++) {
        int[] original = arr.get(i).clone();
        long average = 0;
        for (int j = 0; j < 10; j++) {
            int[] temp = original.clone();
            long start = System.nanoTime();
            Sort.mergeSort(temp);
            long end = System.nanoTime();
            long elapsedTime = end - start;
            average += elapsedTime;
        }
        long milli = average / 10_000_000;
        res[i] = milli;
    }
    return res;
}
```

Figure 9: Test function for merge sort algorithm

In test functions, given arraylist is iterated for each partition and clone of each array is created to prevent those arrays to be sorted, which makes them reusable for other operations. We are only sorting shallow copy of those arrays. Each array is tested ten times in the inner loop. To sort them, respective algorithms from Figure 2 to Figure 5 are called from the Sort class, in this case, merge sort is called. Using System.nanotime() more precise measurements are done and dividing it by 10 million gives it the average of 10 experiments in milliseconds. This values are stored in a an array an returned for further operations.

After testing random data, using sortArrayListOfArrays method from Utils class, arrays inside of the arraylist are sorted to test sorting algorithms running time on sorted data. Implementation of sortArrayListOfArrays is given in Figure 10. It basically uses the predefined merge sort algorithm from Sort class to sort arrays.

```java
public static void sortArrayListOfArrays (ArrayList<int[]> arr) {
    for (int i = 0; i < arr.size(); i++) {
        Sort.mergeSort(arr.get(i));
    }
}
```

Figure 10: Sorting of the arrays of Arraylist

After sorting the arrays, same logic is applied above and functions are called to obtain running time values of the sorting algorithms like in Figure 8. To test the reversely tested data, arrays inside of the arraylist need to be reversed. When the experiment on the sorted data is

finished, reverse method from Utils class is called to reverse those arrays, which can be seen at Figure 11.

```
public static void reverse (ArrayList<int[]> arr) {
    for (int i = 0; i < arr.size(); i++) {
        for (int j = 0; j < (arr.get(i).length) / 2; j++) {
            int temp = arr.get(i)[j];
            arr.get(i)[j] = arr.get(i)[arr.get(i).length-j-1];
            arr.get(i)[arr.get(i).length-j-1] = temp;
        }
    }
}
```

Figure 11: Reversing algorithm for arrays

Reverse method iterates over arrays inside of the arraylist and then swaps the items from the beginning and the end to reverse the order of the items in the array. This works in O(n) time.

After obtaining the reversely sorted arrays, each sorting method is called from their respective class with test functions on these reversely sorted arrays. Their runtimes are obtained, and the same process is proceeded in Figure 8.

With the runtime values obtained, using the Plot class and the graph method from it, these values are plotted as a graph using xchart module for further analysis.

With sorting analysis data obtained, searching can be implemented next. Most of the logic is same for searching as it was in the sorting. Each search algorithm has its own test method with the same logic. For instance, search values are obtained for linear search with random data is given in Figure 12.

```
long[] linearSearchRandomTimes = LinearSearchTest.test(partedResRandom1);
```

Figure 12: Linear Search testing call

While testing searching algorithms, they are run on the same partitions for 1000 times and their running time values are averaged as nanoseconds instead of milliseconds. To find a search value in each partition, random item is selected for that specific partition each time for 1000 times. Implementation of the test algorithm for Binary Search is given in Figure 13 as an example. Picking an item for each partition and searching for it 1000 times is avoided due to the following reason. Firstly, random item could be the very first item in the array or items at the near end at the beginning and it will result in O(1) or near O(1) time complexity. When running times are compared in the graphs, these possible random selections can cause fluctuations on the graph, which disrupts to get an correct analysis. Instead, for each iteration throughout the 1000 iterations, new random value is created and searched for it. Still, these creates fluctuations as well, but it decreases the noise in the graph and gives more uniform results.

```java
public static long[] test (ArrayList<int[]> arr) {
    long[] res = new long[arr.size()];
    for (int i = 0; i < arr.size(); i++) {
        int[] temp = arr.get(i);
        long avgSearch = 0;
        for (int j = 0; j < 1_000; j++) {
            Random random = new Random();
            int randomIndex = random.nextInt(temp.length);
            int searchVal = temp[randomIndex];
            long start = System.nanoTime();
            int index = Search.binarySearch(temp, searchVal);
            long end = System.nanoTime();
            long elapsedTime = end - start;
            avgSearch += elapsedTime;
        }
        avgSearch /= 1_000;
        res[i] = avgSearch;
    }
    return res;
}
```

Figure 13: Test function for Binary Search operation

After obtaining the results from linear search on random data on Figure 12, we have to sort the arrays to perform operations with binary search and test linear search on sorted data. Last time, arrays on the arraylist, called partedResRandom1 were reversely sorted. To obtain the sorted arrays, we can just reverse them once again using reverse method again from Utils class. Then, linear search and binary search are tested with sorted data using the same logic in Figure12. When the running time values are obtained, they are printed to the console using print method from Utils class, and using graph method from Plot class, they are displayed in a graph for analysis.

3- Results, Analysis, Discussion

After the implementation of the sorting and searching algorithms mentioned above and the preparation of the dataset to be sorted, experiment is set to be executed. To run the tests, all .java files are compiled and then executed with the name of the dataset given from the command line. When the program has ended, sorting, and searching running times for the algorithms are observed as in Table 1 below.

Before analyzing the sort and search algorithms, their expected theoretical time and space complexities are given in Table 1 and Table 2 below. In the worst case, theoretically, counting sort has to have the best running time, and in the base case, insertion sort has to have the best running time. Merge sort generally performs well under any condition apart from other two algorithm and has guaranteed O(nlogn) runtime complexity.

Linear search is expected to have O(n) worst case and $\Omega(1)$ best case runtime complexity. Binary search is expected to have O(logn) worst case and $\Omega(1)$ best case runtime complexity.

Only 1 sorting algorithm uses constant auxiliary space complexity and both of the search algorithms have constant space complexity. Merge and counting sort has O(n) and O(n+m) space complexity respectively, due to the usage of additional arrays used in the algorithm.

Table 1: Theoretical runtime complexities of the sort and search 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	$\Omega(n+m)$	$\Theta(n+m)$	$O(n+m)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Table 2: Auxiliary Space Complexities of sorting and searching algorithms

Algorithm	**Auxiliary Space Comlexity**
Insertion Sort	$O(1)$
Merge Sort	$O(n)$
Counting Sort	$O(n+m)$
Linear Search	$O(1)$
Binary Search	$O(1)$

In the case of insertion sort, best case run time is expected as O(n), average runtime is expected as $O(n^2)$ and, worst case run time is expected to be $O(n^2)$. When the data is analyzed from the Table 3, real life values adds up to the expectation. Best case scenario happens when the given array is already sorted. In the part, where the given array is sorted, runtime of insertion sort is 0 ms for every partition, as modern computers execute 10^8 compares per second [1]. Regarding to this information, O(n) best case running time for insertion sort ends up with 0 ms, which suits with the expected running complexity. In the average case, random data is given to the insertion sort and expected complexity is $O(n^2)$ because of the comparison of most of the elements in the array. Because modern cpus work pretty fast, run time for insertion sort did not rise quickly. However, from the array size of 8000 to 250000, quadratic runtime had its effects, and running time of insertion sort increased quadratically with the increasing size of the array. The worst possible case for insertion sort is that given array is reversely sorted. In that case, algorithm needs to make the most amount of comparison possible to sort the array. When the experiment data is observed, this hypothesis comes out to be true. As early as size 2000, insertion sort starts to have significant running time values and increases quadratically with increasing array sizes. Within three types of supplied data, insertion sort has the worst running time in reversely sorted arrays, which is expected from its theoretical runtime complexity.

Table 3: Running time values of sorting algorithms with sorted, random and, reversely sorted data (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	8	46	132	657	2080	8879
Merge Sort	0	0	0	0	1	2	4	13	23	43
Counting Sort	189	176	162	162	163	162	163	166	168	171
Sorted Input Data Timin Results in ms										
Insertion Sort	0	0	0	0	0	0	0	0	0	0
Merge Sort	0	0	0	0	0	1	2	5	10	23
Counting Sort	183	165	164	164	164	165	165	167	169	168
Reversely Sorted Input Data Timing Results in ms										
Insertion Sort	0	0	1	4	18	67	254	1004	4104	16655
Merge Sort	0	0	0	0	0	1	2	5	8	23
Counting Sort	174	164	163	162	163	163	163	165	166	166

In the case of merge sort, it always has a guaranteed runtime complexity of O(nlogn) given any type of input. This is the best there exist to be achieved among comparison-based sorting algorithms. Regardless of the given properties of the array, merge sort is going to sort it in O(nlogn) time at worst. When the real life values are observed from Table 1, it took merge sort at most 43 ms to sort an array with the size of 250000 with random distribution. When the sizes of the partitions doubled, running time of the merge sort did not change drastically like in the insertion sort algorithm. This is the result of O(nlogn) runtime complexity of the merge sort. Regardless of the properties of the given array, merge sort sorted that array within a short time. Real life data backs up the theoretical complexity of merge sort algorithm.

In the case of counting sort algorithm, theoretical complexity of its runtime is O(n+m), where n is the size of the array and m is the value of the maximum element in the array. Within insertion, merge and counting sort, theoretically counting sort has to have the best running time values. However, as mentioned before, counting sort is optimal when the given array has a small range of values. In the experiment, starting from the smallest partition with 500 values, that array has a maximum value of over 2 million. To sort an array of just 500 values, counting sort has to create an array with size of 2 million. Inside of the counting sort, algorithm traverses this count array twice, which is inefficient. Its desirable near linear runtime complexity becomes unrealistic when tested with wide range of values. Additionally, when the partition size increases, maximum value of that specific partition also increases, and counting sort becomes unsuitable to use for sorting. For that reason, real life values in Table 3, has a lot

higher values than merge sort. Theoretically, merge sort has to be slower, however, properties of the given array made theoretical complexity of counting sort far worse. Although, when the values are inspected, for random, sorted and, reversely sorted data, counting sort took around 160-170 ms to sort. If the given values in that partitions were to be in the small range, linear runtime looks attainable from the obtained data.

To understand the theoretical complexities of these algorithms and their real time running time values, data from Table is plotted and displayed at Figure 14 to Figure 16.

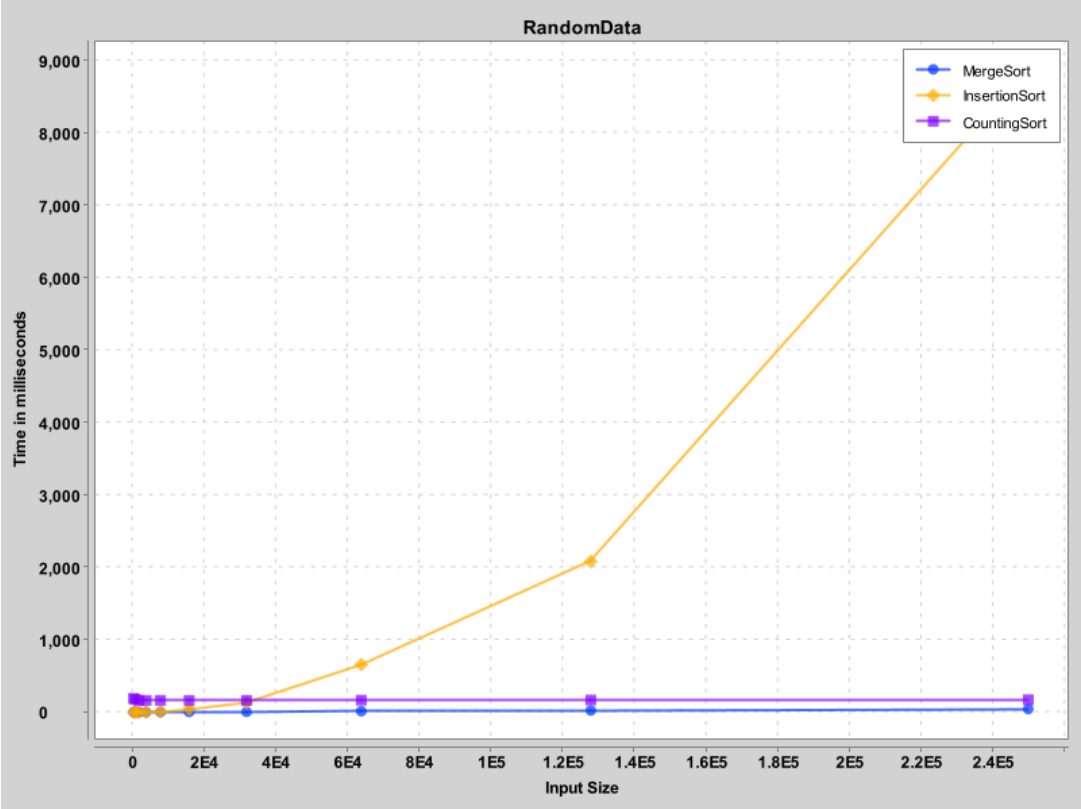

Figure 14: Running time values of sorting algorithms on random data

For random data, merge sort is expected to have O(nlogn) complexity, counting sort to have O(n+m) complexity and, insertion sort to have $O(n^2)$ complexity. As expected, insertion sort had the worst runtime values and, merge sort and counting sort performed according to their respective running time complexities with the mentioned properties above.

For sorted data, insertion sort has $\Omega(n)$ runtime complexity. Merge sort and counting sort performs same regardless of the given property of data. As expected, insertion sort behaved best among these three, merge sort came second and counting sort took more time to sort because of the wide range of values in the given arrays.

For reversely sorted data, insertion sort has the worst runtime complexity of quadratic runtime. The reason for this is insertion sort has to compare maximum amount of comparison until finding the correct position of the item in the array. Merge sort and counting sort has the same runtime complexity as random and sorted data. They performed quite well with respect to insertion sort as it can be seen in Figure 16 above. With the increasing array size, running time of insertion sort increased drastically, yet merge sort and counting sort did not change that much.

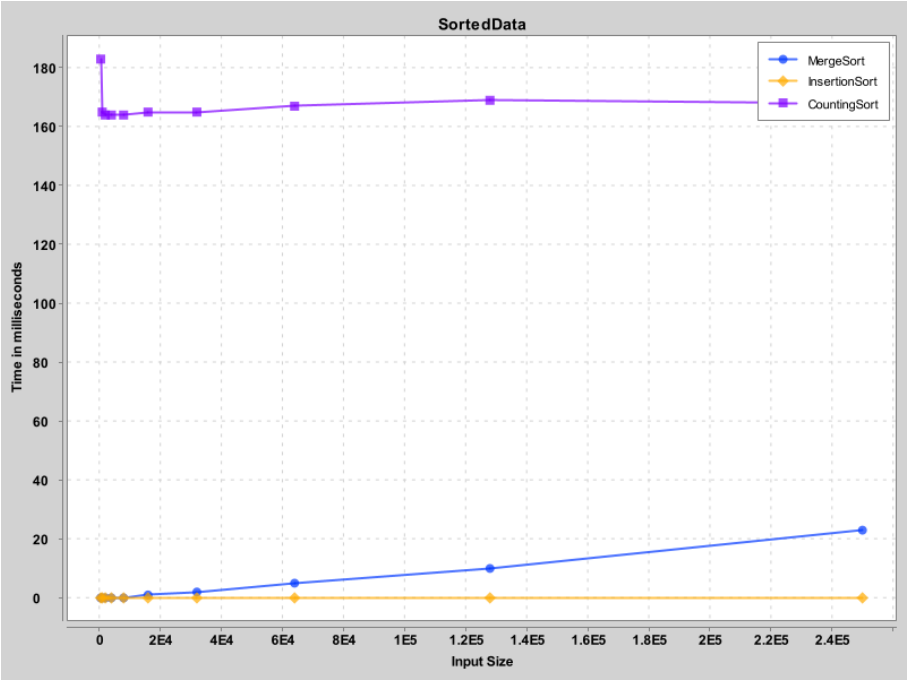

Figure 15: Running time values of sorting algorithms on sorted data

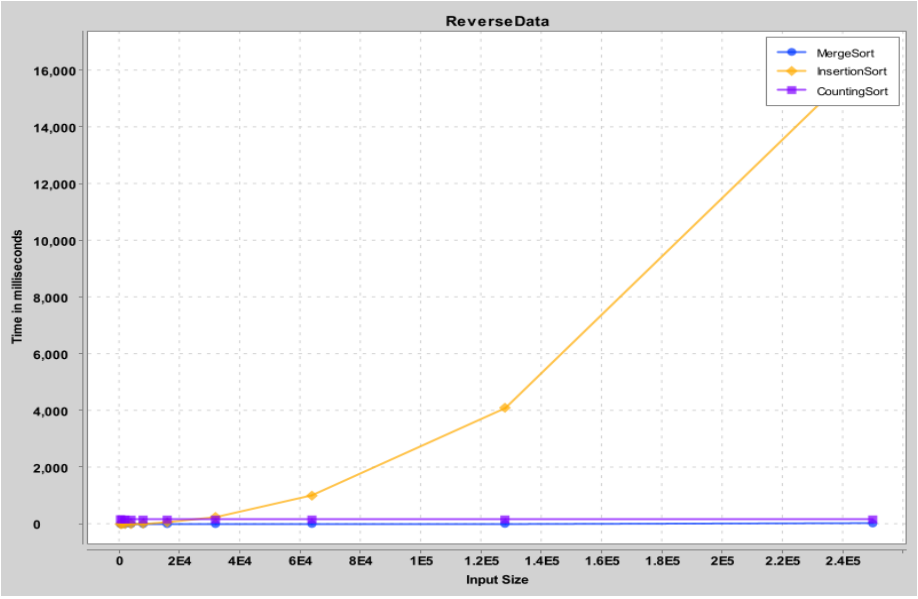

Figure 16: Running time values of sorting algorithms on reversely sorted data

Linear search and binary search are two of the most known search algorithms there exist. In this experiment, they are tested to see if their theoretical runtimes match with the real life running time values. For linear search, it has a worst case scenario of O(n) and best case scenario of $\Omega(1)$. For binary search, it has a worst case scenario of O(logn) and best case scenario of of O(1) as well. These two algorithms are tested with different sizes of arrays and their running time values are listed below at Table 4.

Table 4: Running time values of searching algorithms with sorted and, random data (in ns)

	Input Size *n*									
Algorithm	500	1000	2000	4000	8000	16000	32000	64000	128000	250000
Linear Search (random data)	33440	4662	5942	1586	2813	4535	9694	12447	16062	25836
Linear Search (sorted data)	203	480	931	1724	1798	3487	6717	14413	28221	77965
Binary Search (sorted data)	463	282	317	317	348	356	406	289	295	324

In the experiment, items from the arrays are picked randomly and searched within the array. In other words, both algorithms never tested with the worst case, which is item not existing in the array. Linear search is expected to have linear runtime complexity, however, in random data, it does not follow a linear running time values with the increasing input size. For example, in theory, linear search with array size of 4000 has to have bigger runtime than array size of 2000. However, in the experiment, this is not the case. The reason behind this results is the random item selection from the main array. We do not have a authority which item is going to be selected from the array. In this 1000 iteration in the 4000 array, all of them magically can be the starting 1000 element and in the 2000 array, all of them can be the ending 100 element. Due to this possible piling, fluctuation within the search algorithm can happen. However, when the array size starts to increase, probability of this piling decreases and running time of the linear search algorithm linearly increases as it can be seen from the Table 4, which matches with the theoretical complexity.

O(n) runtime complexity of the linear search can be easily observed and proved with real life data from the experiment with sorted data in Table 4. When the data is sorted for linear search, it has to search for it until it finds it. In random data, very large value might be at the starting index of the array and can disrupt the running time value of the algorithm. However, in sorted data, linear increase of the running time of linear search can be seen in Table 4. These values matches with the expected runtime of the linear search.

Binary search operates on a sorted data and has a O(logn) runtime complexity. In theory, it should be faster than the linear search and should have more tolerance to the changing size of the given array. When data from Table 4 is inspected, it runs faster than the linear search and does not change drastically when the size of the array is increased. This is the result of O(logn) time complexity. Due to the random selection of items from the array, fluctuation might appear in the data as it happened with array size of 500 with binary search. Also, binary search is applied to the 500 array, 1000 times. In this case, same items can be randomly selected which takes the average running time higher. Overall, running time values of linear and binary search algorithms are suitable with their theoretical runtime complexities. Their relation can be seen in a plot in Figure 17 below. As the size of the given array increases, fluctuations disappears and theoretical complexities matches with the real time running time values.

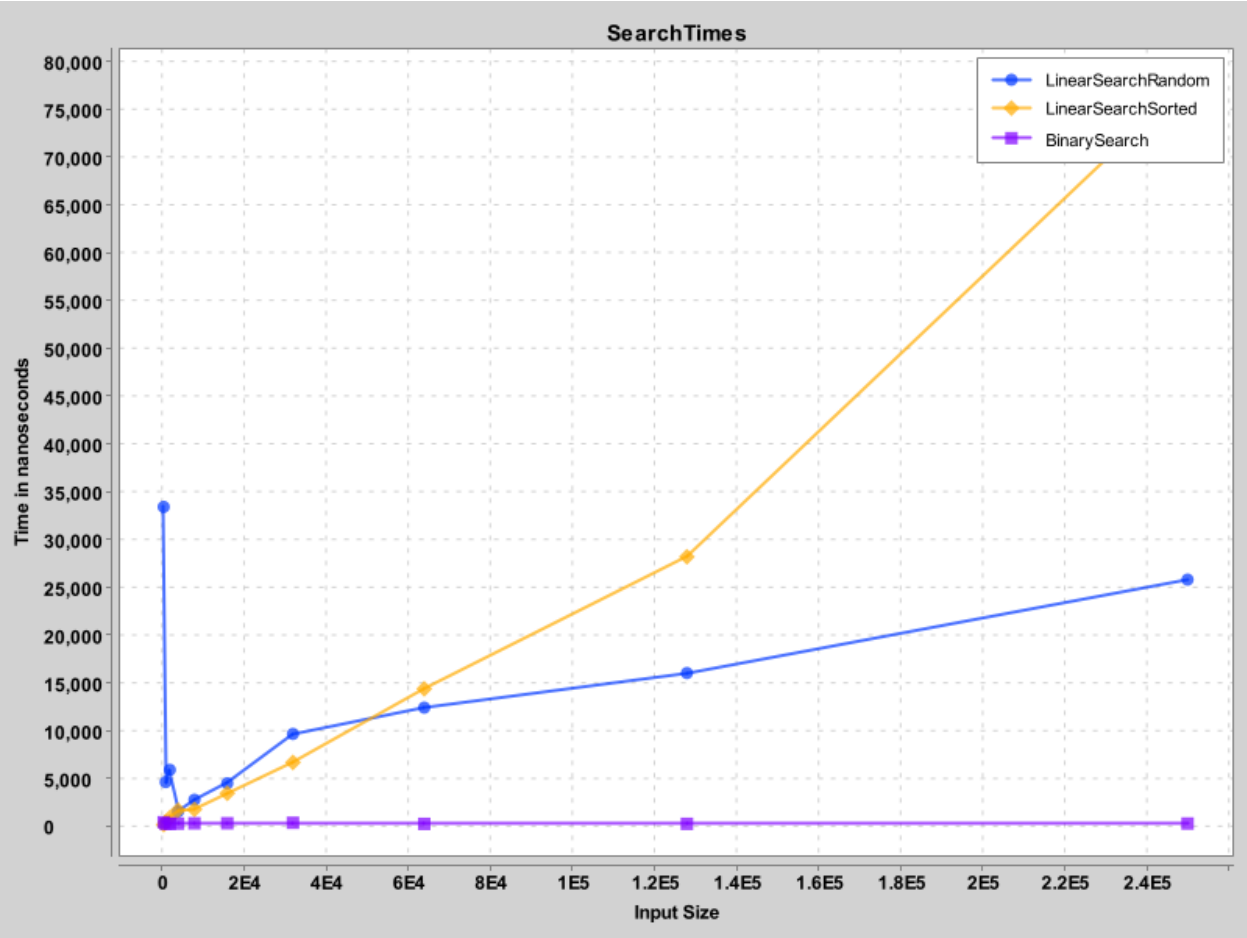

Figure 17: Running time values of searching algorithms on sorted and random data

References

[1] Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.