Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 26, 2023

Student name:
Eren Gül

Student Number:
b2200356037

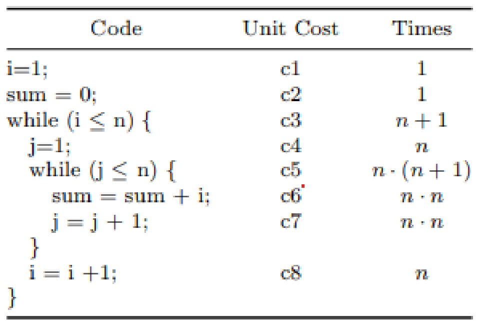
1 Problem Definition

Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be sorted. Furthermore, modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to computation. The efficiency of a sorting algorithm can be observed by applying it to sort datasets of varying sizes and other characteristics of the dataset instances that are to be sorted. In this assignment, the given sorting and searching algorithms will be classified based on two criteria:

• Computational (Time) Complexity: Determining the best, worst and average case behavior in terms of the size of the dataset. Table 3 illustrates a comparison of computational complexity of some well-known sorting and searching algorithms.

• Auxiliary Memory (Space) Complexity: Some sorting algorithms are performed "in-place" using swapping. An in-place sort needs only O(1) auxiliary memory apart from the memory used for the items being sorted. On the other hand, some algorithms may need O(log n) or O(n) auxiliary memory for sorting operations. Searching algorithms usually do not require additional memory space. Table 4 illustrates an auxiliary space complexity comparison of the same well-known sorting and searching algorithms.

A time complexity analysis focuses on gross differences in the efficiency of algorithms that are likely to dominate the overall cost of a solution. See the example given below:



Code	Unit Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i ≤ n) {`	c3	$n+1$
`  j=1;`	c4	n
`  while (j ≤ n) {`	c5	$n \cdot (n+1)$
`    sum = sum + i;`	c6	$n \cdot n$
`    j = j + 1;`	c7	$n \cdot n$
`  }`		
`  i = i +1;`	c8	n
`}`		

The total cost of the given algorithm is c1 + c2 + (n + 1) · c3 + n · c4 + n · (n + 1) · c5 + n · n · c6 + n · n · c7 + n · c8. The running time required for this algorithm is proportional to n^2, which is determined as its growth rate, and it is usually denoted as O(n^2).

2 Solution Implementation

2.1 Selection Sort

```java
public class Algorithms {
    public static void selectionSort(Integer[] list, int n) {
        for (int i = 0; i < n - 1; i++) {
            int min = i;
            for (int j = i + 1; j < n; j++) {
                if (list[j] < list[min])
                    min = j;
            }
            if (min != i)
                swap(list, min, i);
        }
    }

    private static void swap(Integer[] list, int i, int j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
}
```

2.2 Quicksort

```java
public class Algorithms {
    public static void quicksort(Integer[] list, int low, int high) {
        int stackSize = high - low + 1;
        int[] stack = new int[stackSize];
        int top = -1;
        stack[++top] = low;
        stack[++top] = high;
        while (top >= 0) {
            high = stack[top--];
            low = stack[top--];
            int pivot = partition(list, low, high);
            if (pivot - 1 > low) {
                stack[++top] = low;
                stack[++top] = pivot - 1;
            }
            if (pivot + 1 < high) {
                stack[++top] = pivot + 1;
                stack[++top] = high;
            }
        }
    }

    private static int partition(Integer[] list, int low, int high) {
        int pivot = list[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (list[j] <= pivot) {
                i++;
                swap(list, i, j);
            }
        }
        swap(list, i + 1, high);
        return i + 1;
    }

    private static void swap(Integer[] list, int i, int j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
}
```

2.3 Bucket Sort

```java
public class Algorithms {
    public static int[] bucketSort(Integer[] list, int n) {
        int numberOfBuckets = (int) Math.sqrt(n);
        ArrayList<Integer>[] buckets = new ArrayList[numberOfBuckets];
        for (int i = 0; i < numberOfBuckets; i++)
            buckets[i] = new ArrayList<>();
        int max = max(list);
        for (int number : list)
            buckets[hash(number, max, numberOfBuckets)].add(number);
        for (ArrayList<Integer> bucket : buckets)
            Collections.sort(bucket);
        int[] sortedArray = new int[n];
        int index = 0;
        for (ArrayList<Integer> bucket : buckets) {
            for (int number : bucket)
                sortedArray[index++] = number;
        }
        return sortedArray;
    }

    private static int hash(int i, int max, int numberOfBuckets) {
        return (int) Math.floor(i / max * (numberOfBuckets - 1));
    }

    private static int max(Integer[] list) {
        int max = list[0];
        for (int i = 1; i < list.length; i++) {
            if (list[i] > max)
                max = list[i];
        }
        return max;
    }
}
```

2.4 Linear Search

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

2.5 Binary Search

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

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									
Selection sort	0	0	1	3	12	47	161	641	2592	9987
Quick sort	0	0	0	0	0	0	2	9	28	49
Bucket sort	0	0	0	0	1	3	5	9	17	40
	Sorted Input Data Timing Results in ms									
Selection sort	0	0	0	1	5	21	70	274	1083	4152
Quick sort	0	0	0	1	7	37	138	544	2173	8249
Bucket sort	0	0	0	0	0	0	0	1	1	4
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0	0	3	13	52	192	764	3058	12198	46499
Quick sort	0	0	0	2	5	10	34	148	566	1200
Bucket sort	0	0	0	0	0	0	0	1	3	7

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)	150	201	310	484	732	1821	3058	5497	7587	12261
Linear search (sorted data)	153	128	251	503	953	1570	4091	7957	14600	24046
Binary search (sorted data)	165	174	162	132	141	152	166	168	169	143

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Each sorting algorithm is tested 10 times and each searching algorithm is tested 1000 times for each input size. The results can be seen in running time test tables. By looking at Table 1 and Table 2 along with the plots, it can be seen that the complexities in Table 3 and Table 4 holds.

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

Tests on Random Data: Fig. 1.

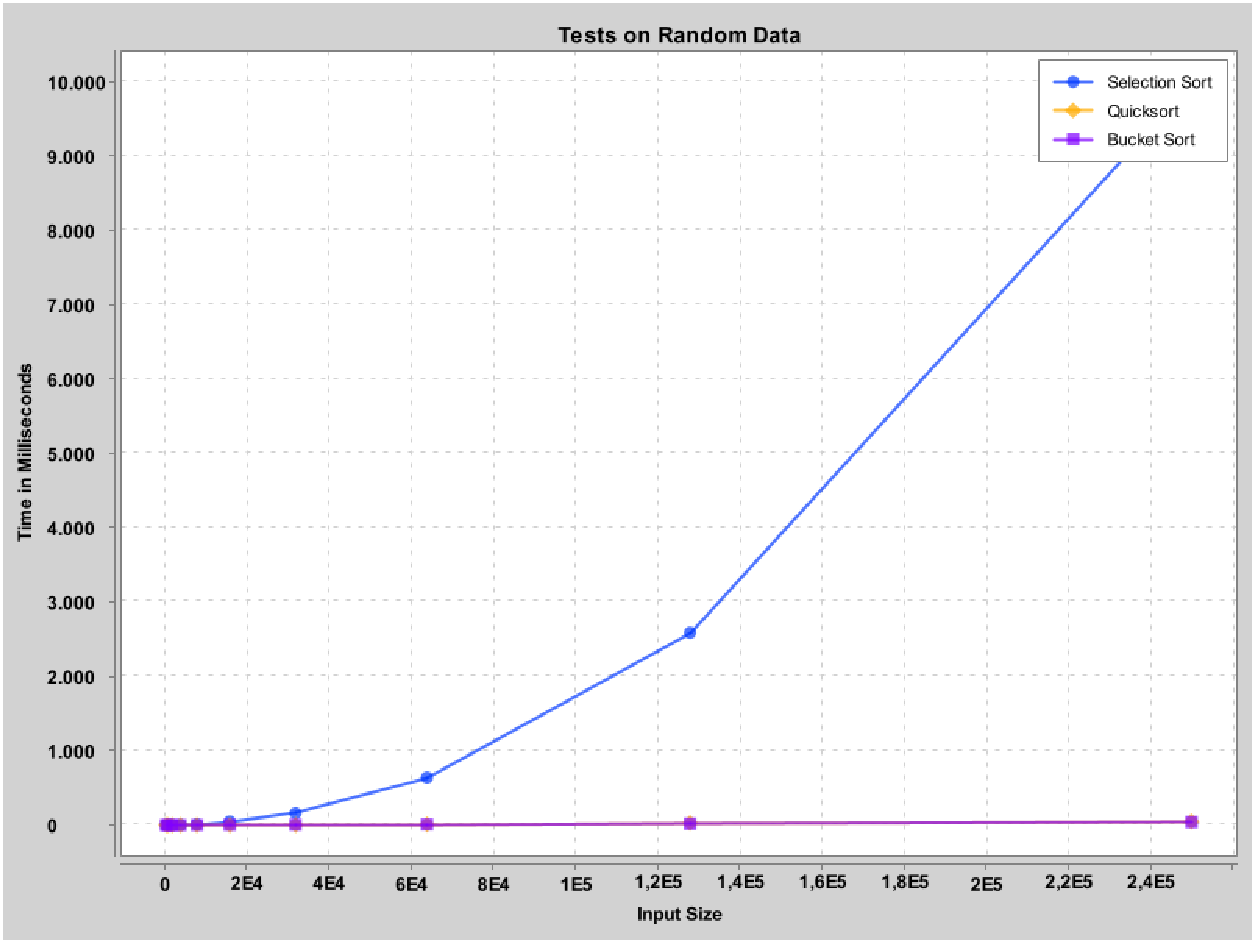


Figure 1: Plot of the tests on random data for all sorting algorithms.

Tests on Random Data: Fig. 2.

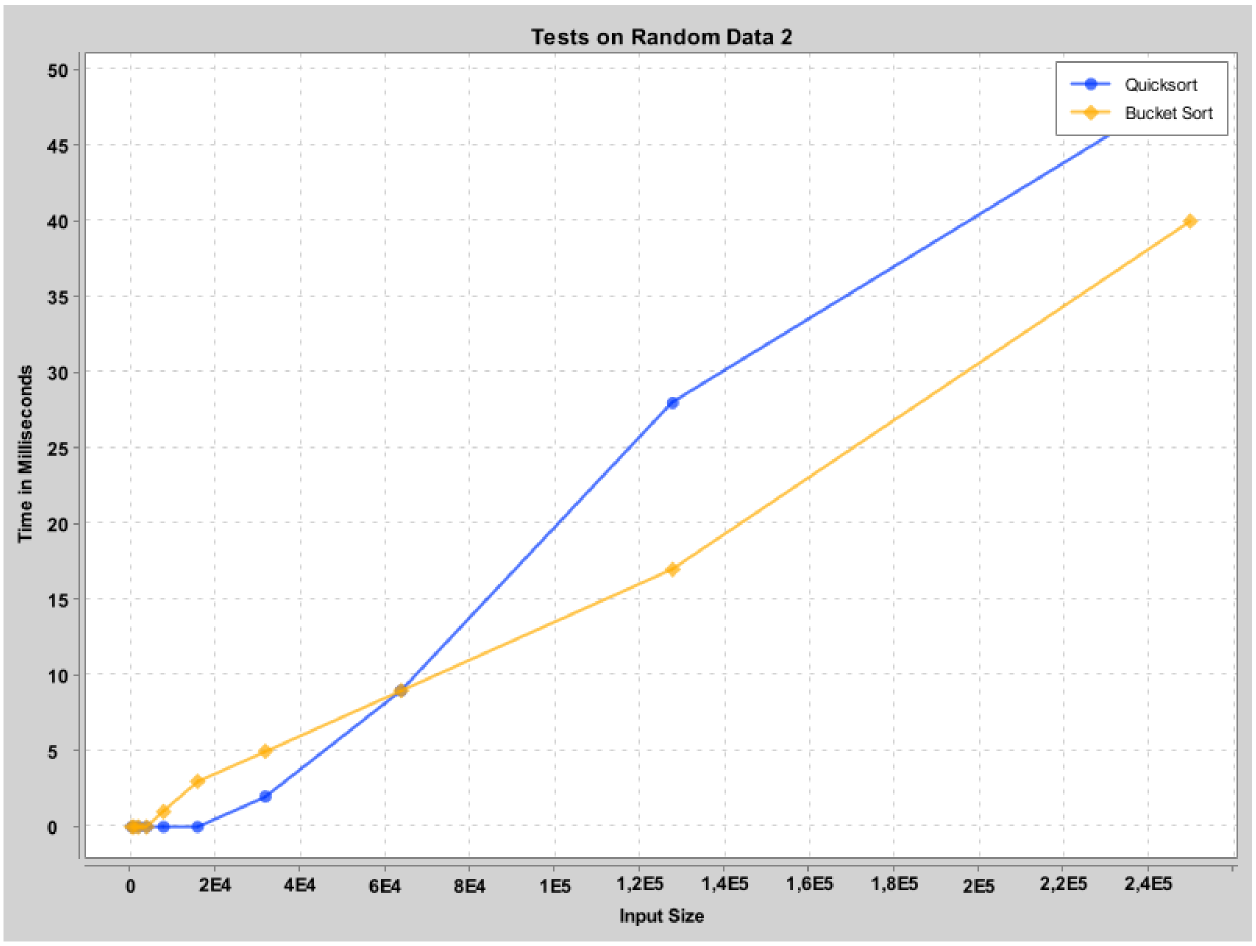


Figure 2: Plot of the tests on random data for quicksort and bucket sort algorithms.

Tests on Sorted Data: Fig. 3.

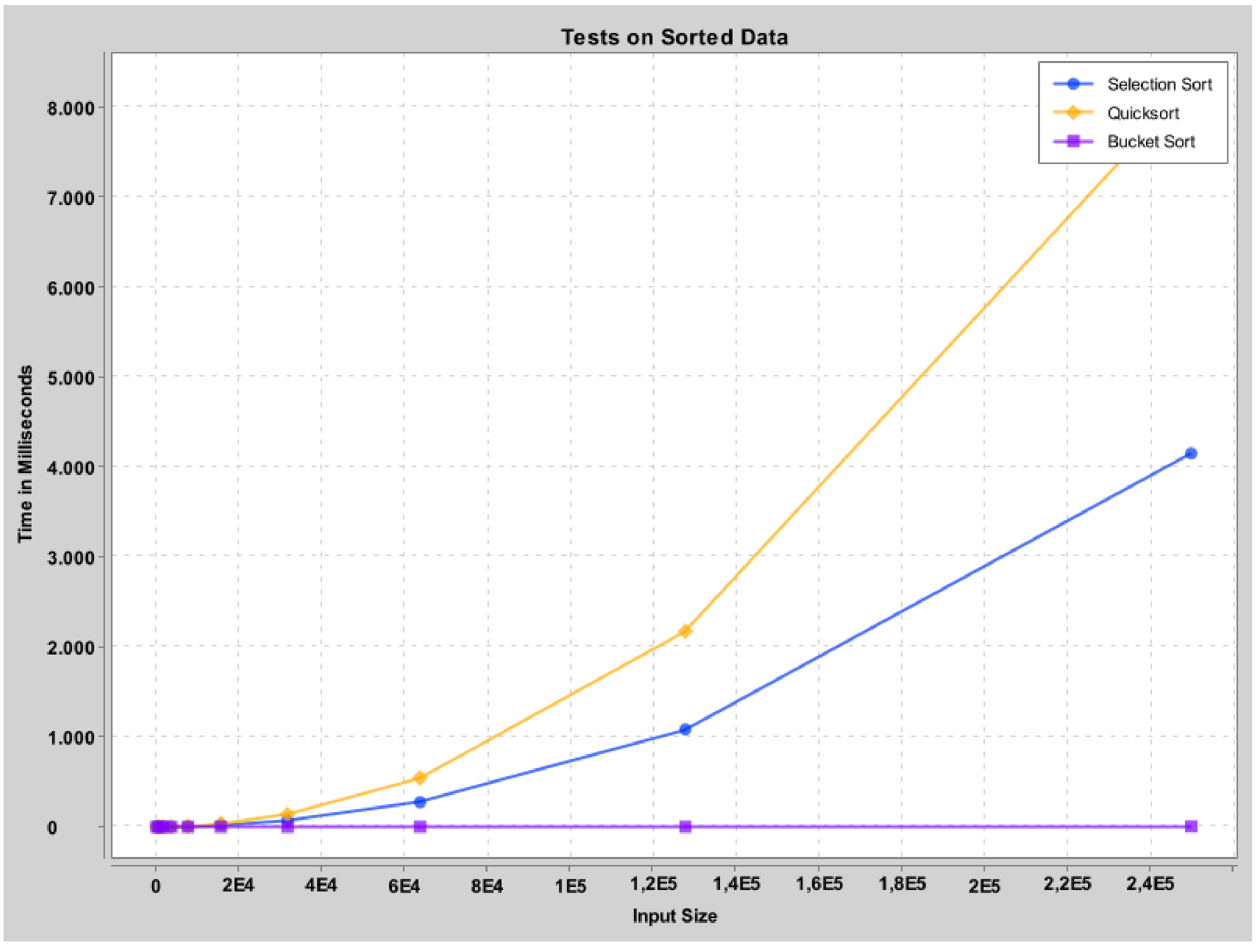


Figure 3: Plot of the tests on sorted data for all sorting algorithms.

Tests on Reversely Sorted Data: Fig. 5.

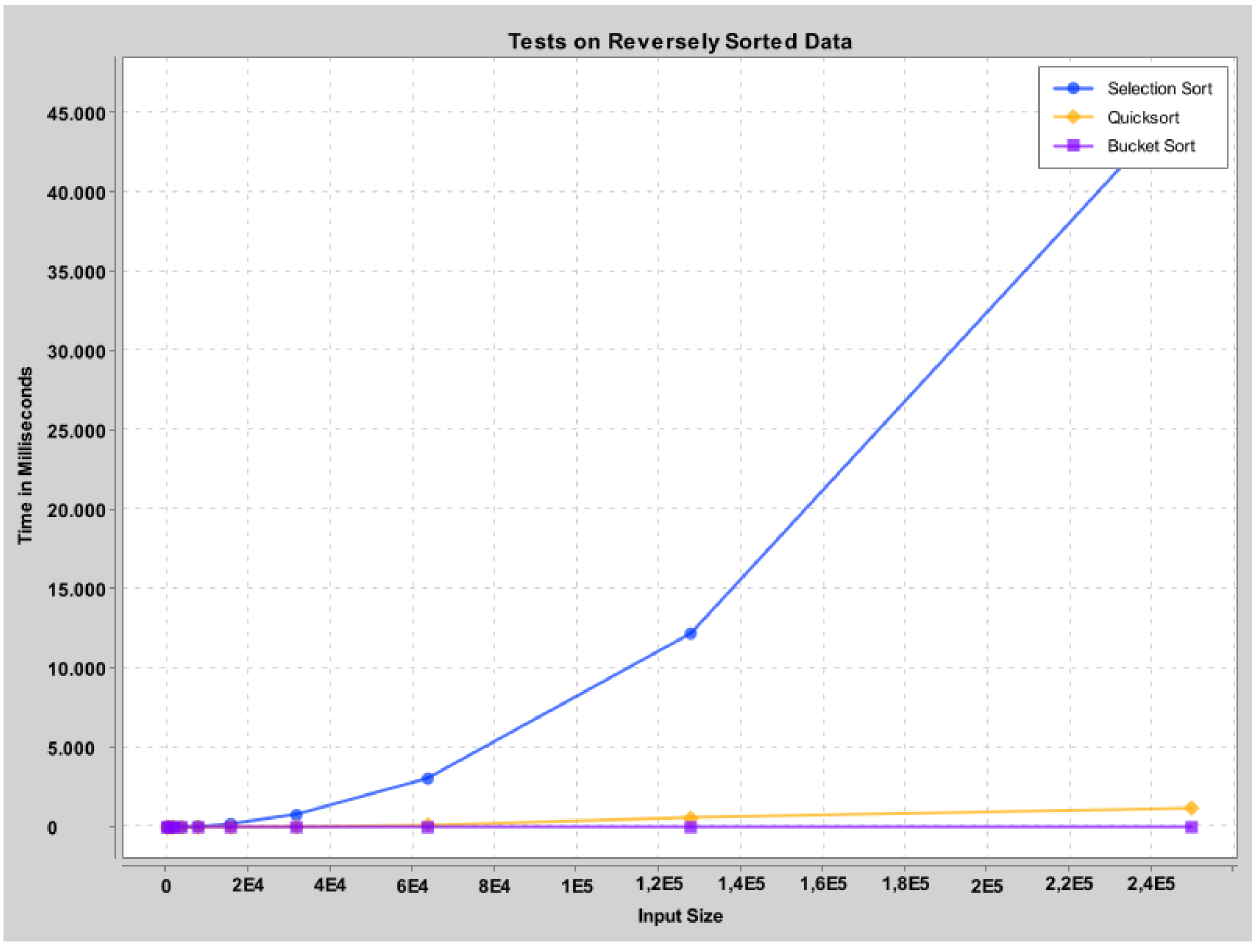


Figure 4: Plot of the tests on reversely sorted data for all sorting algorithms.

Tests on Reversely Sorted Data: Fig. 5.

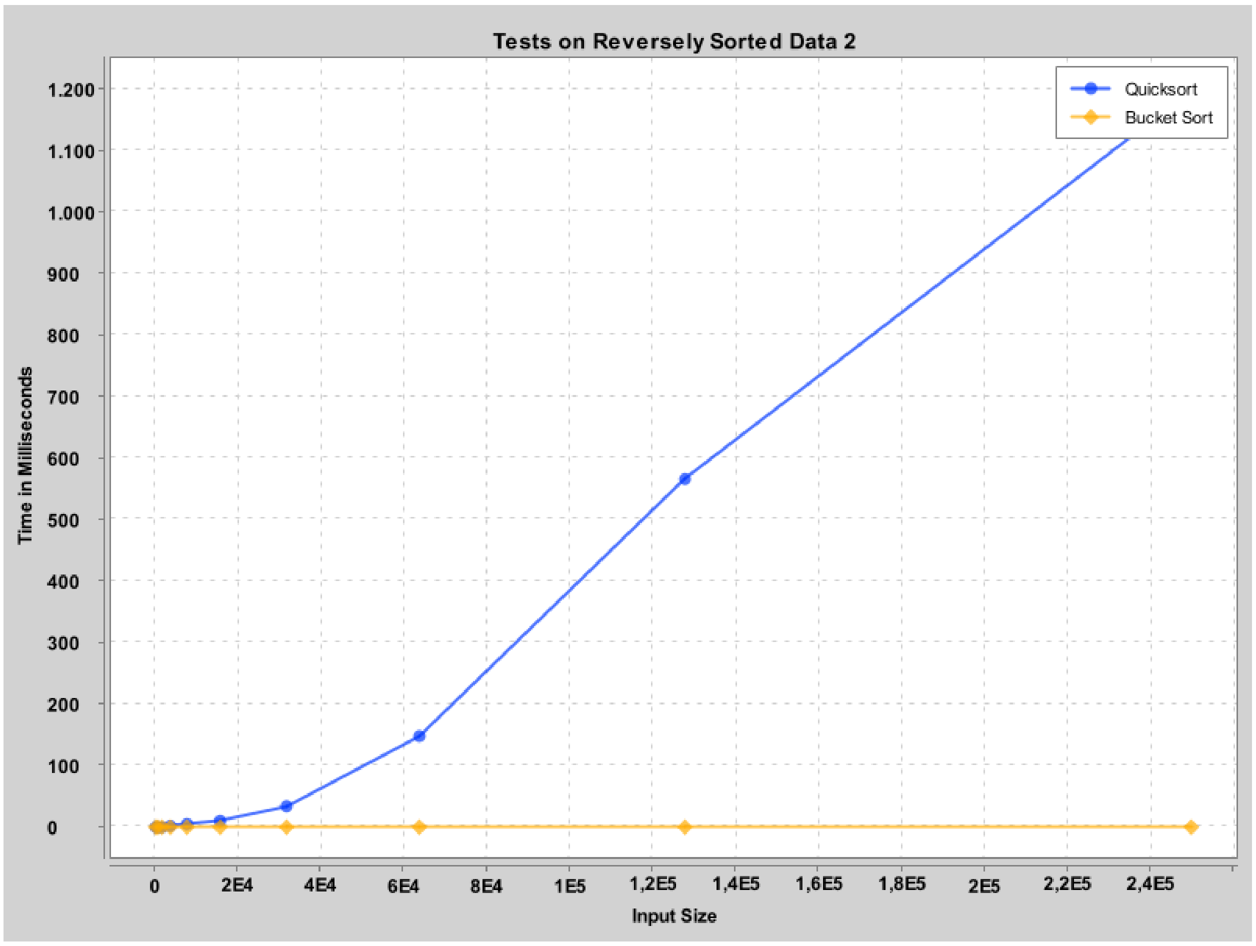


Figure 5: Plot of the tests on reversely sorted data for quicksort and bucket sort algorithms.

Tests on Searched Data: Fig. 6.

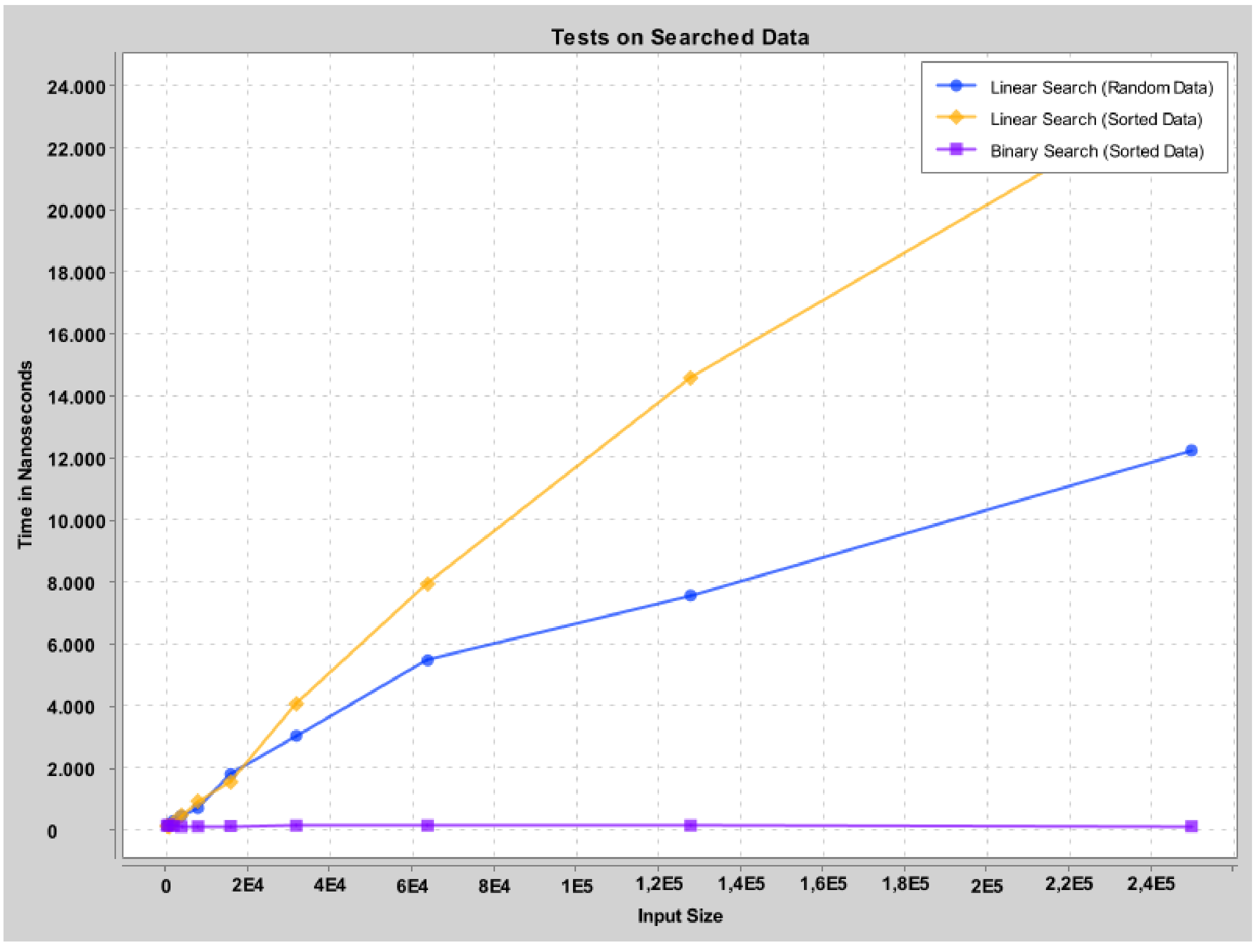


Figure 6: Plot of the tests on searched data for all algorithms.

According to the computational complexities in Table 3 and plots, selection sort is always $O(n^2)$ no matter what case it is. The worst case for quicksort is $O(n^2)$, if the pivot is always chosen as the greatest or lowest element in the array and when the data is sorted or reversely sorted. Quicksort is $\Omega(n \log n)$ and $\Theta(n \log n)$ for best and average case respectively if the above criteria isn't met. The worst case for bucket sort is $O(n^2)$ if all the elements are placed in a single bucket. Bucket sort is $\Omega(n+k)$ and $\Theta(n+k)$ for best and average case respectively if the above criteria isn't met. For linear search, the best case is $\Omega(1)$ when the value being searched is the first element of the list. Linear search is $\Theta(n)$ and $O(n)$ for average and worst cases respectively if the value being searched is at the end of the list or not the first element of the list. For binary search, the best case is $\Omega(1)$ when the target element is in the middle of the list. Binary search is $\Theta(\log n)$ and $O(\log n)$ for average and worst cases respectively if the target element is not in the middle of the list and somewhere else.

By looking at the obtained results, it can be seen that they match their theoretical asymptotic complexities. Although they may not look perfect, this can be because of various reasons such as CPU specifications, programming language or operating system.

References

- https://www.geeksforgeeks.org/
- https://www.wikipedia.org/
- https://www.bigocheatsheet.com/