Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 24, 2023

Student name:
Gizem Aleyna TUZCU

Student Number:
b2200356816

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. 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. Two different search algorithms performed on the data to see the more efficient one.

2. Solution Implementations

2.1.Bucket Sort

```
public class BucketSort {
    public int hash(int i, int max, int numberOfBuckets){
        return (int) ((double) i / max * (numberOfBuckets-1));
    }
    public ArrayList<Integer> bucketSort(ArrayList<Integer> arr){
        int numberOfBuckets = (int)Math.sqrt(arr.size());
        ArrayList<ArrayList<Integer>> buckets = new ArrayList<>(
            numberOfBuckets);
        for (int i = 0; i<numberOfBuckets; i++){
            buckets.add(new ArrayList<>());
        }
        int max = Collections.max(arr);
        for (int i: arr){
            buckets.get(hash(i,max,numberOfBuckets)).add(i);
        }
        Comparator<Integer> comparator = Comparator.naturalOrder();
        for (ArrayList<Integer> bucket : buckets){
            bucket.sort(comparator);
        }
        ArrayList<Integer> sortedArray = new ArrayList<>();
        for (ArrayList<Integer> bucket : buckets){
            sortedArray.addAll(bucket);
        }
        return sortedArray;
    }
}
```

2.2.Quick Sort

```java
public class QuickSort {
    public ArrayList<Integer> quickSort(ArrayList<Integer> arr, 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(arr, low, high);
            if ((pivot - 1) > low) {
                stack[++top] = low;
                stack[++top] = pivot - 1;
            }
            if ((pivot + 1) < high) {
                stack[++top] = pivot + 1;
                stack[++top] = high;
            }
        }
        return arr;
    }

    public int partition(ArrayList<Integer> arr, int low, int high) {
        int pivot = arr.get(high);
        int i = (low - 1);

        for (int j = low; j < high; j++) {
            if (arr.get(j) <= pivot) {
                i++;
                Integer temp = arr.get(i);
                arr.set(i, arr.get(j));
                arr.set(j, temp);
            }
        }

        Integer tempH = arr.get(i + 1);
        arr.set(i + 1, arr.get(high));
        arr.set(high, tempH);
        return i + 1;
    }
}
```

2.3.Selection Sort

```
public class SelectionSort {
    public ArrayList<Integer> selectionSort(ArrayList<Integer> arr, int n){
        for (int i = 1; i < n-1;i++){
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (arr.get(j) < arr.get(i)){
                    min = j;
                }
            }
            if (min!=i){
                int temp = arr.get(min);
                arr.set(min, arr.get(i));
                arr.set(i, temp);
            }
        }
        return arr;
    }
}
```

2.4.Linear Search

```
public class LinearSearch {
    public int linearSearch(ArrayList<Integer> arr, int x){
        int size = arr.size();
        for (int i = 0;i<size;i++){
            if (arr.get(i) == x){
                return i;
            }
        }
        return -1;
    }
}
```

2.5.Binary Search

```
public class BinarySearch {
    public int binarySearch(ArrayList<Integer> arr, int x){
        int low = 0;
        int high = arr.size()-1;
        while ((high-low)>1){
            int mid = (high+low)/2;
            if (arr.get(mid)<x){
                low=mid+1;
            }else {
                high = mid;
            }
        }

        if (arr.get(low)==x){
            return low;
        } else if (arr.get(high)==x) {
            return high;
        }
        return -1;
    }
}
```

3. Results, Analysis, Discussion

3.1. Sorting Algorithms on Random Data

	512	1024	2048	4096	8192	16384	32768	65536	131072	251282
Quick Sort	0.0	0.0	0.0	0.0	0.2	2.2	14.2	64.4	504.7	856.4
Bucket Sort	0.0	0.0	0.0	0.1	1.6	2.1	4.0	8.5	20.0	59.3
Selection Sort	0.0	0.0	1.1	5.5	23.8	95.6	405.1	1608.9	6540.4	25971.1

Figure 1. Random Data Test Results (in ms)

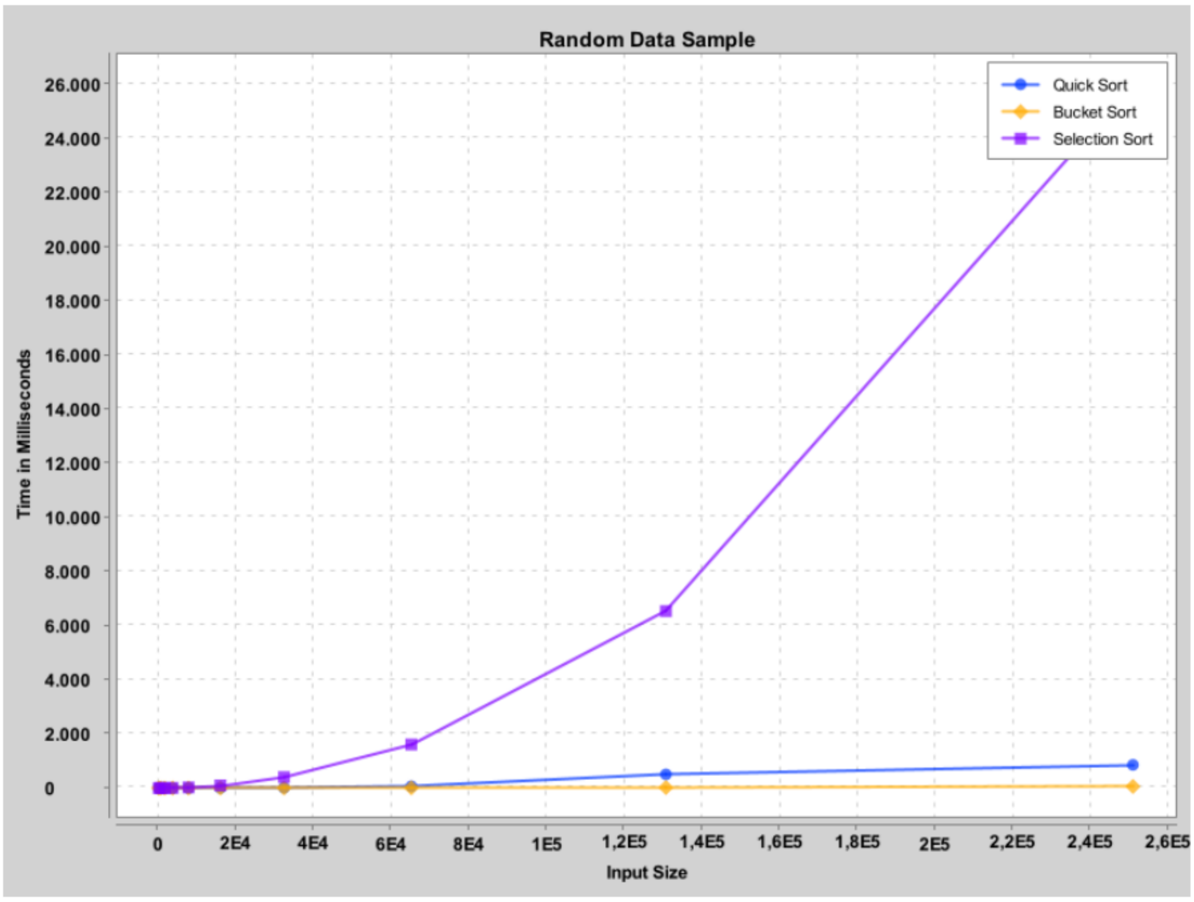
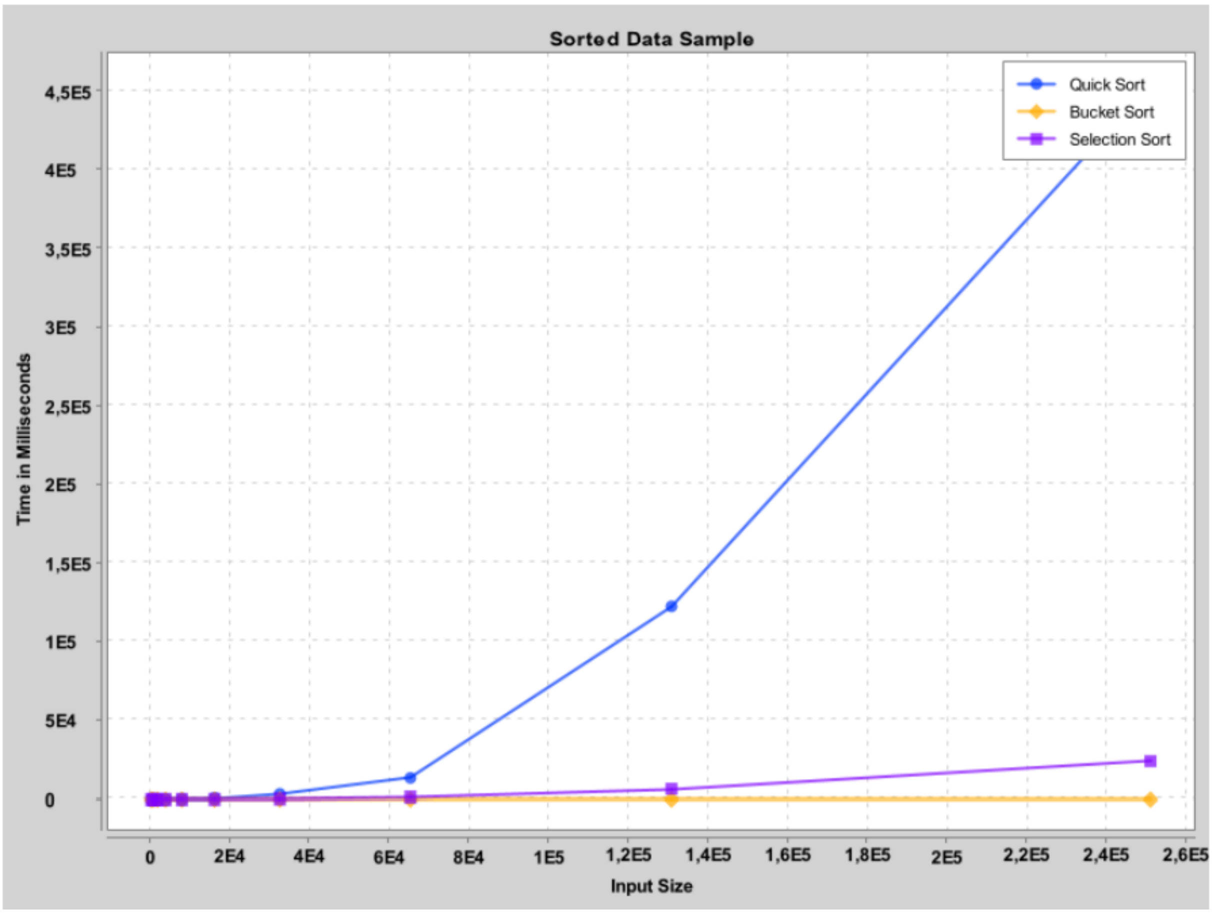


Figure 2. Random Data Sample

3.2.Sorting Algorithms on Sorted Data

	512	1024	2048	4096	8192	16384	32768	65536	131072	251282
Quick Sort	0.0	2.5	12.7	46.7	191.1	744.7	3218.4	13368.9	122497.7	454175.5
Bucket Sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	7.0	13.1
Selection Sort	0.0	0.0	1.3	5.6	23.6	93.4	390.8	1565.8	6453.9	24063.4

Figure 3.Sorted Data Test Results (in ms)



Şekil 4.Sorted Data Sample

3.3.Sorting Algorithms On Reversely Sorted Data

	512	1024	2048	4096	8192	16384	32768	65536	131072	251282
Quick Sort	0.0	1.0	5.1	15.6	50.0	96.1	354.9	1708.4	11921.7	24236.9
Bucket Sort	0.0	0.0	0.0	0.0	0.0	0.1	0.3	2.4	11.5	31.8
Selection Sort	0.0	0.0	1.2	5.6	23.4	97.0	408.5	1648.8	6709.1	24764.3

Figure 5. Reversely Sorted Data Test Results (in ms)

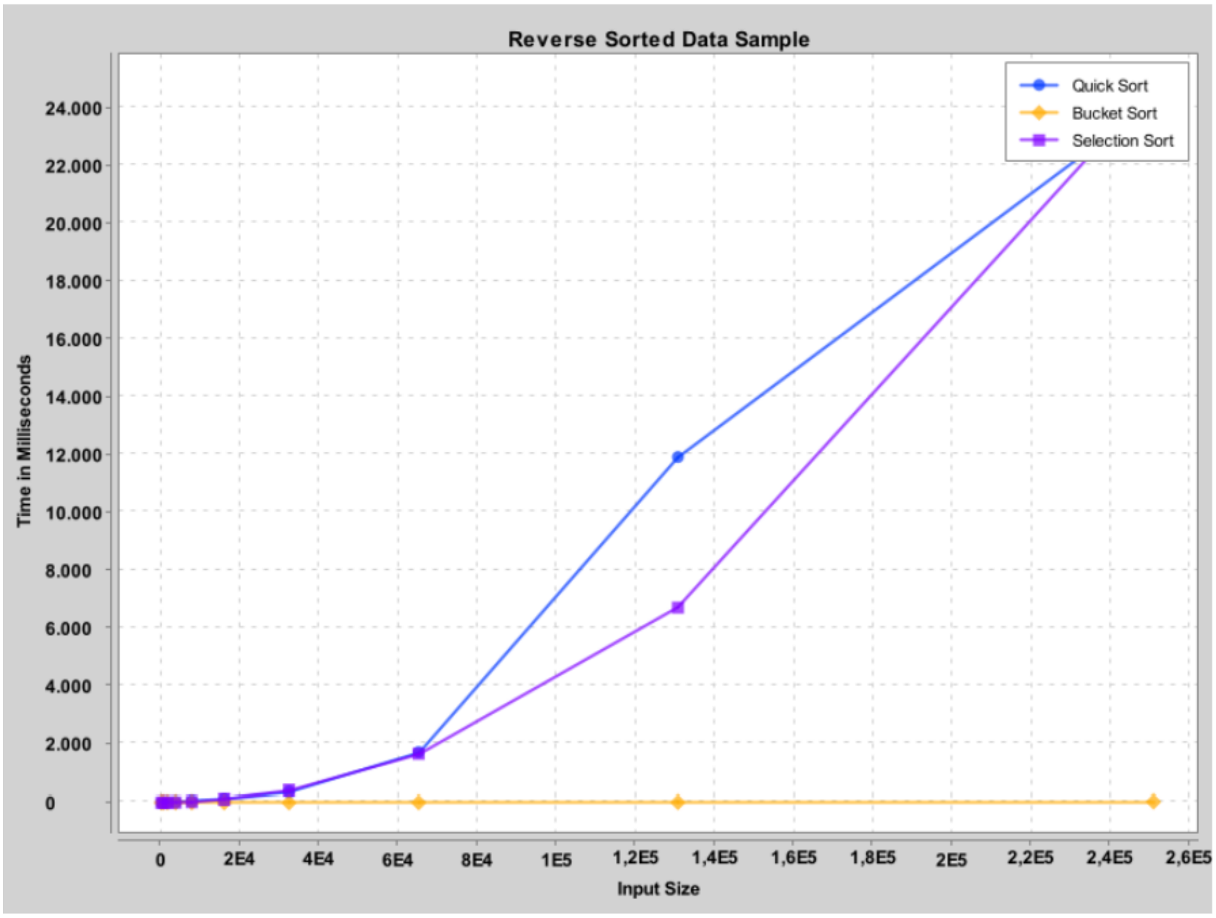


Figure 6.Reversely Sorted Data Sample

3.4.Searching Algorithms

	512	1024	2048	4096	8192	16384	32768	65536	131072	251282
Random Linear Search	346.895	676.502	1279.13	2597.408	4699.812	9852.608	20152.419	48846.687	102650.8	264403.827
Sorted Linear Search	331.001	583.891	1223.401	2628.502	4830.01	11711.101	26503.224	68720.398	146887.09	386804.067
Sorted Binary Search	212.802	95.104	174.198	130.709	68.709	126.003	210.292	209.599	258.609	533.608

Şekil 7.Test results of Searching Algorithms (in nanosec.)

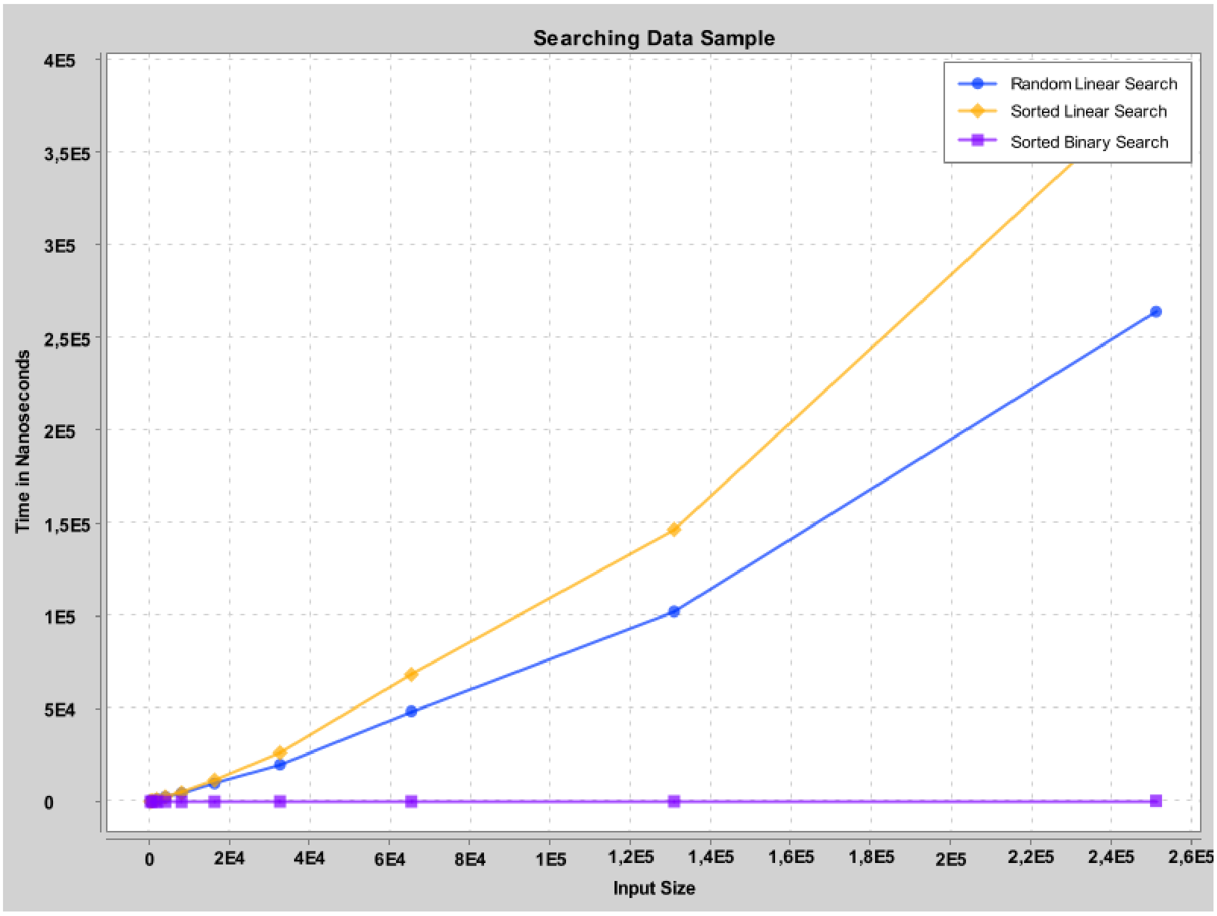


Figure 8. Searching Sample Data

4. The Theoretical Computational and Auxiliary Space Complexities

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

Figure 9. Computational complexity comparison of the given algorithms.

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

Figure 10. Auxiliary space complexity of the given algorithms.

Time Complexity of Selection Sort: The time it takes to sort an array grows quadratically as the number of elements in the array rises because selection sort has an O(n^2) time complexity. This is due to the fact that, in the worst case, selection sort performs (n^2)/2 comparisons. Selection sort is less effective than other sorting algorithms when dealing with large input sizes.

Space Complexity of Selection Sort: Selection sort has an O(1) space complexity, which means that regardless of the size of the input, it always requires a fixed amount of extra space to sort the array.

Time Complexity of Quick Sort: Quick sort is a sorting algorithm that sorts an array using a divide-and-conquer strategy. For high input sizes, it outperforms selection sort and other quadratic sorting algorithms thanks to its average time complexity of O(nlogn). Nonetheless, in the worst case, it could have an O(n^2) time complexity, which is avoidable by picking a strong pivot element.

Space Complexity of Quick Sort: Quick sort has an average space complexity of O(logn) and a worst-case space complexity of O(n), where n is the number of entries in the array.

Time Complexity of Bucket Sort: A linear time sorting method called bucket sort divides an array into buckets and sorts each bucket separately. When n is the number of elements and k is the number of buckets, the time complexity is O(n+k). When the elements are evenly distributed among the buckets, bucket sort is efficient.

Space Complexity of Bucket Sort: The space complexity of bucket sort is O(n+k), where n is the number of elements in the array and k is the number of buckets.

Time and Space Complexity of Linear Search: An array is sequentially searched for a target value using the simple search method known as linear search. While the technique might need to scan through every element in the array to find the target, its time complexity is O(n), where n is the number of elements in the array. It simply needs a fixed amount of extra space to store the target value and index variables, making its space complexity O(1). Although linear search is simple to use, its effectiveness is limited for big input sizes.

Time and Space Complexity of Binary Search: A binary search technique divides an ordered array in half periodically until the desired value is located. Due to the fact that each comparison reduces the search space in half, the time complexity is O(logn), where n is the number of elements in the array. As index variables only need a fixed amount of extra space, it has an O(1) space complexity. For high input sizes, binary search is effective and can locate the target in a sorted array quickly. However, it requires that the array be sorted first.

5. Results

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

  For sorting algorithms:

  - Selection sort: Regardless of the input, Selection Sort always has an O(n2) time complexity since it scans the array n times and makes n-1 comparisons on each run. As it sorts the array in place and doesn't need any more space, its space complexity is O(1).
  - Bucket Sort: The bucket sort has an O(n+k) time complexity with evenly distributed items, the best and average scenario is O(n+k), while the worst situation is O(n2) with all elements in one bucket. Bucket storage has an O(n+k) space complexity.
  - Quick Sort: In the best and average situation, Quick Sort's time complexity is O(nlogn). The worst-case scenario is O(n2) with bad pivot selection. In the average situation, space complexity is O(logn), and in the worst case, it can be O(n).

  For searching algorithms:

  Whereas the time complexity of Binary Search is related to the logarithm of the number of elements in the array, the time complexity of Linear Search is proportional to the number of elements in the array.

When the target is located in the first position, Linear Search has a best case of O(1), whereas Binary Search has a best case of O(1) when the target is located in the middle place. Comparing the average situation, Binary Search is faster than Linear Search, with a time complexity of O(logn), as compared to O(n/2).

The worst-case time complexity for both algorithms is O(n) for Linear Search and O(logn) for Binary Search. The space complexity of Binary Search is the same as that of Linear Search, which is O(1).

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

  Other tests exactly match with theoretic assumptions as opposed to the quick sort with reversely sorted data. They match more closely since tests were conducted several times and the average of the results was used.
  I think more tests would produce more accurate results for the quick sort with reversely sorted data.

References

- geeksforgeeks.org
- stackoverflow.com
- Lecture Notes
- Text Book