HACETTEPE UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BM204 SOFTWARE PRACTICUM II - 2023 SPRING

Programming Assignment 1

March 10, 2023

Student name:
Sezin Yavuz

Student Number:
b2200356050

1 Problem Definition

In the first experiment, we implemented the three sorting algorithms and we ran the given sorting algorithms on different input types and size(in order random data, sorted data and reversely sorted data).At the end we measure the average running time of the sorting process.

In the second experiment, we implemented the two searching algorithms(linear search and binary search). We measured the average running time of each algorithm by running each experiment 1000 times and taking the average of the recorded running times for each input size.

2 Solution Implementation

Sorting Algorithms

2.1 Selection Sort

```
public static void swap(final ArrayList<Integer> arr, final int pos1, final int
pos2) {
    final int temp = arr.get(pos1);
    arr.set(pos1, arr.get(pos2));
    arr.set(pos2, temp);
}

public static ArrayList<Integer> selectionSort(ArrayList<Integer> data, int n) {
    for (int i = 0; i < n; i++) {
        int min = i;
        for (int j = 0; j < n; j++) {
            if (data.get(j) < data.get(min)) {
                min = j;
            }
        }
        if (min != i) {
            swap(data, min, i);
        }
    }
    return data;
}
```

2.2 Quick Sort

```
public static int partition(ArrayList<Integer> data, int low, int high) {
    int pivot = data.get(high);
    int i = low - 1;
    for (int j = low; j < high; j++) {
```

```
        if (data.get(j) <= pivot) {
            i = i + 1;
            swap(data,i,j);
        }
    }
    swap(data, i + 1, high);
    return i + 1;
}
```

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> data, 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(data, 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 data;
}
```

2.3 Bucket Sort

```
public static int hash(int i, int max, int numberOfBuckets) {
    return i / max * (numberOfBuckets - 1);
}

public static ArrayList<Integer> bucketSort(ArrayList<Integer> data, int n) {
    int numberOfBuckets = (int) Math.sqrt(data.size());
    ArrayList<ArrayList<Integer>> buckets = new ArrayList<>(numberOfBuckets);
    int max = Collections.max(data);

    for (int i = 0; i < numberOfBuckets; i++) {
        buckets.add(new ArrayList<>());
    }
    for (int i : data) {
        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) {
```

```
        for (int i : bucket) {
            sortedArray.add(i);
        }
    }
    return sortedArray;
}
```

Search Algorithms

2.4 Linear search

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

2.5 Binary Search

```
public static int binarySearch(ArrayList<Integer> data, int x) {
    int low = 0;
    int high = data.size() - 1;
    while (high - low > 1) {
        int mid = (high + low) / 2;
        if (data.get(mid) < x) {
            low = mid + 1;
        }
    }
    if (data.get(low) == 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	4.414646	1.9274957	7.6274915	32.7483335	127.5162542	513.9779084	2035.2727502	8111.3683125	32159.8239125	124916.2200249
Quick sort	0.3174416	0.2575667	0.4901376	1.0215041	2.293771	7.7128501	17.9825958	60.39505	213.85525	742.8820501
Bucket sort	189.1468458	73.8043791	72.3363293	76.5912876	72.9293043	72.132625	79.3207373	86.0872334	75.3405665	74.953725
	Sorted Input Data Timing Results in ms									
Selection sort	1.6064167	1.8681293	7.5527499	28.7259125	112.9397834	448.2890292	1845.1876417	7498.1809499	29716.9438666	112148.4682165
Quick sort	2.6919458	5.8328544	22.9461167	87.0375376	346.0434041	1382.9160459	5561.5877459	22547.0976834	90585.2058875	346417.7148373
Bucket sort	21.4323707	16.2674501	13.183229	9.6545126	20.8085835	15.0566959	16.2697126	8.7185999	9.1099834	8.9998958
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	5.7317958	1.9901333	7.7241126	33.6653375	151.0737	591.1530836	2024.0136126	7678.4532543	32001.2564666	
Quick sort	2.3023416	2.1830294	18.1107832	79.8132044	260.7422917	997.8217418	3760.6079792	14437.4062288	57542.2756292	
Bucket sort	29.9405043	18.8337876	14.9238666	12.5120876	12.7302209	12.5269876	13.2911333	12.9776958	13.2511126	

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)	2211.072	614.857	866.002	1522.166	2882.267	5660.879	12254.75	22488.578	44891.168	86780.863
Linear search (sorted data)	739.815	657.71	435.568	1201.416	1891.84	3986.669	11722.668	25529.14	59426.02	108634.03
Binary search (sorted data)	630.92	505.144	308.228"	370.003	168.803	142.911	128.112	122.324	188.196	170.349

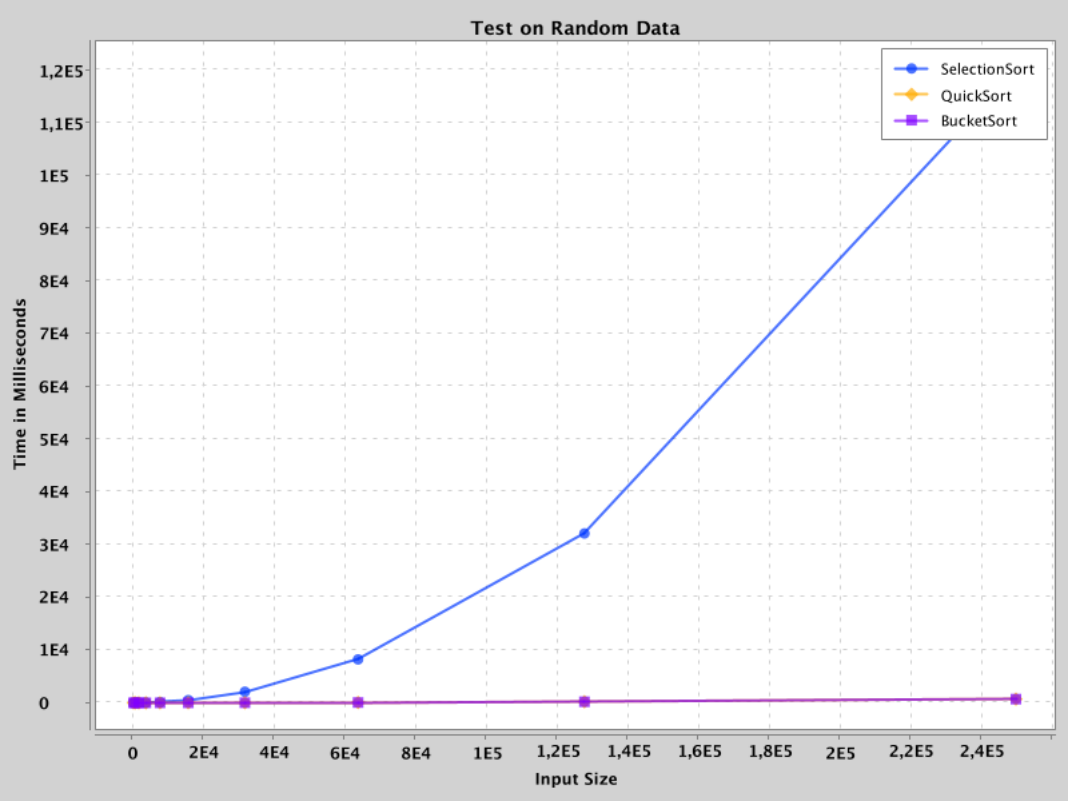
Complexity analysis tables :

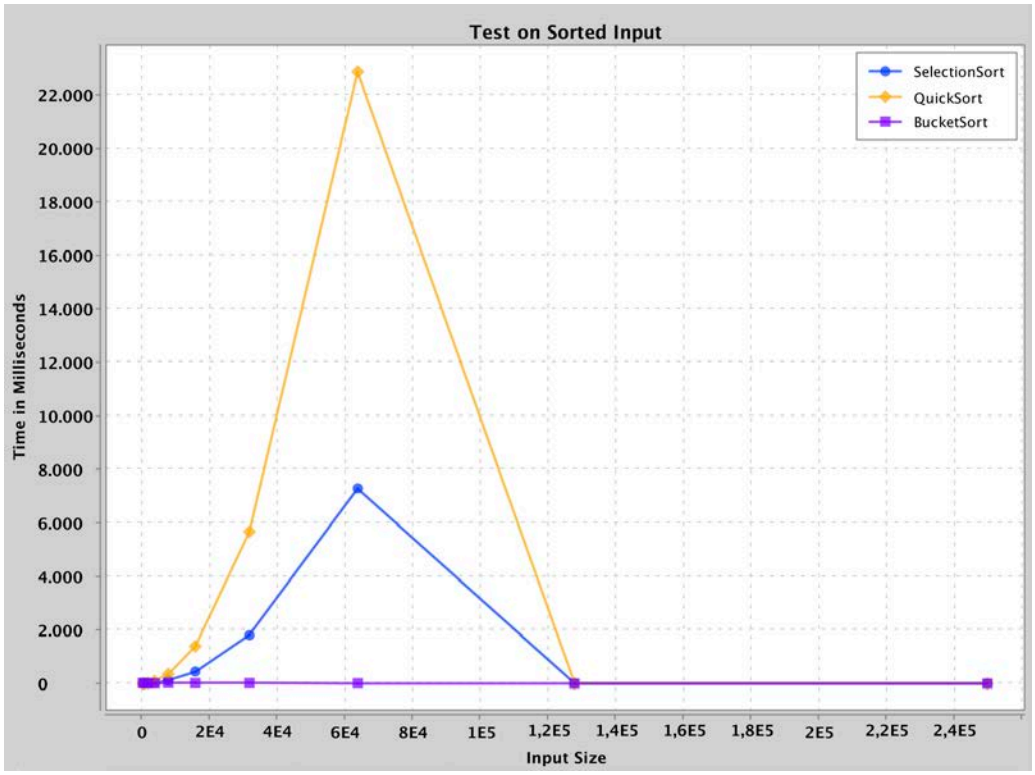
Computational complexity comparison of the given algorithms.

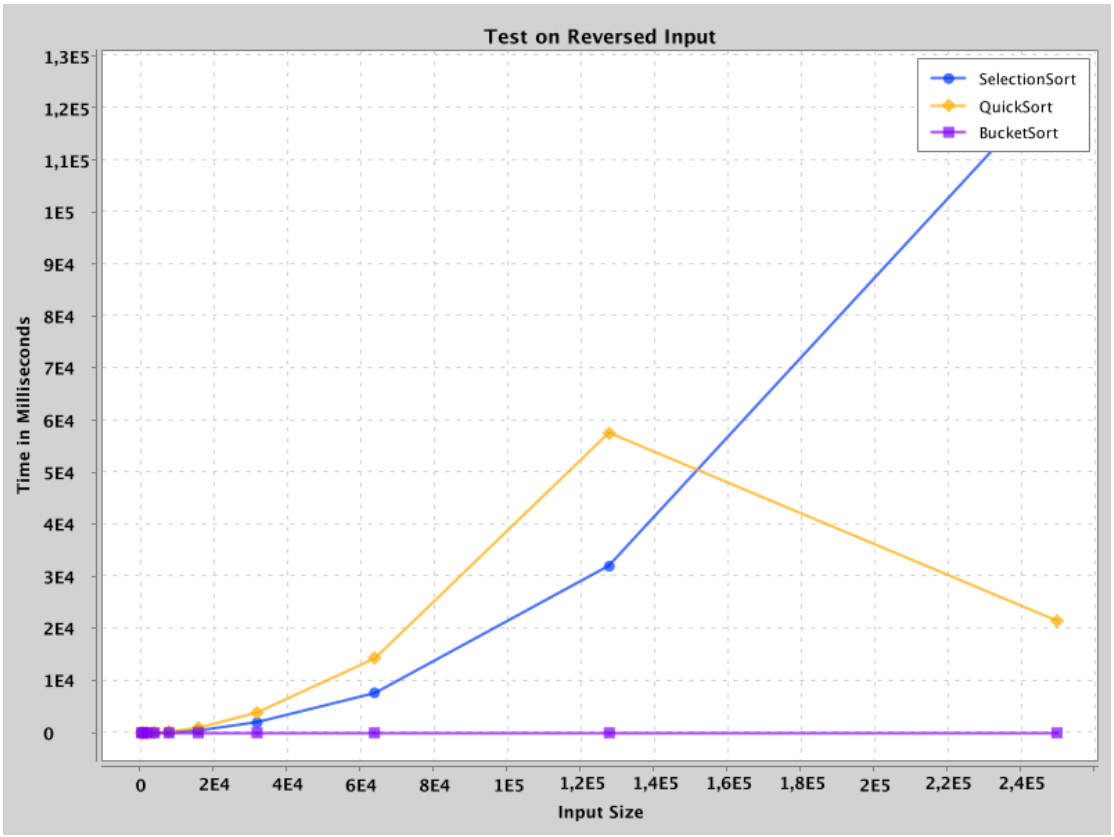
Algorithm	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Bucket Sort	Ω (n + k)	Θ (n + k)	$O(n^2)$
Linear Search	Ω (1)	Θ (n)	$O(n^2)$
Binary Search	Ω (1)	Θ (log n)	$O(\log n)$

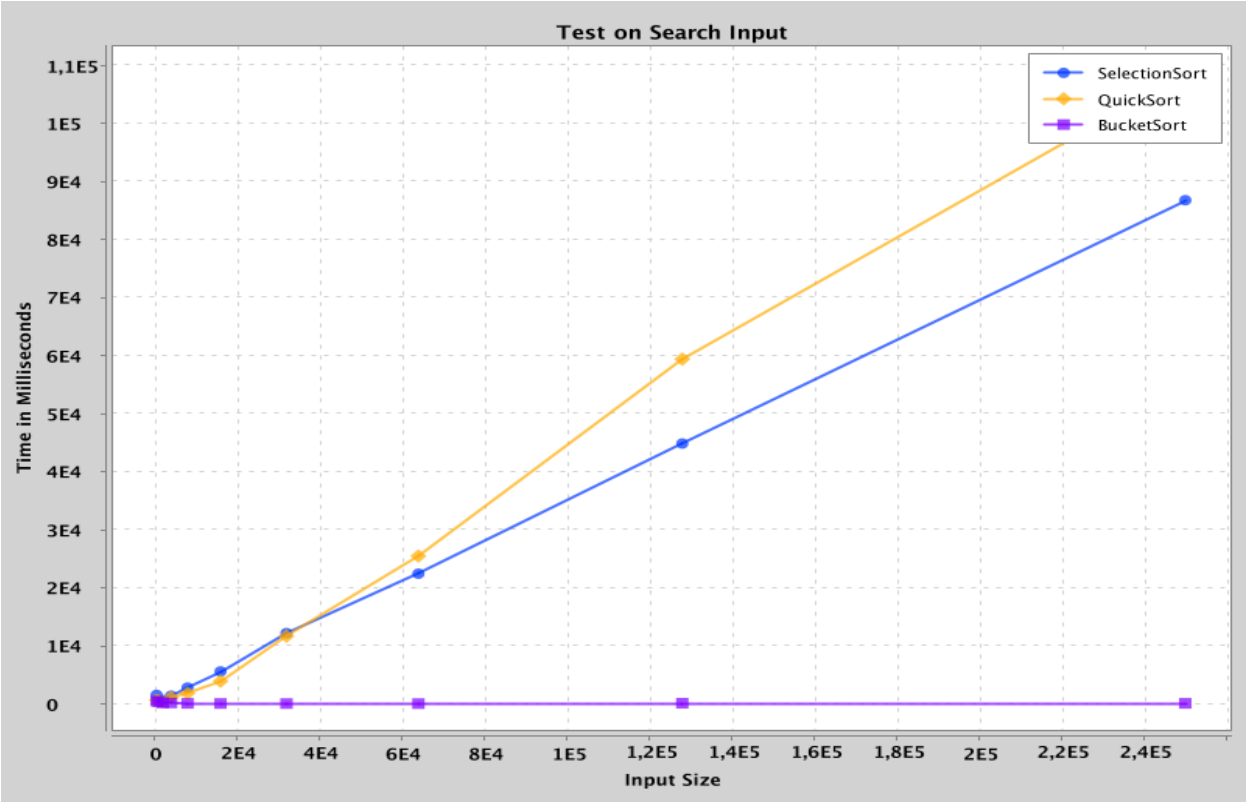
Auxiliary space complexity of the given algorithms.

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




Test on Sorted Input
SelectionSort
QuickSort
BucketSort
Time in Milliseconds
22.000
20.000
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
Input Size


Test on Reversed Input
SelectionSort
QuickSort
BucketSort
Time in Milliseconds
1,3E5
1,2E5
1,1E5
1E5
9E4
8E4
7E4
6E4
5E4
4E4
3E4
2E4
1E4
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
Input Size


Test on Search Input
SelectionSort
QuickSort
BucketSort
Time in Milliseconds
1,1E5
1E5
9E4
8E4
7E4
6E4
5E4
4E4
3E4
2E4
1E4
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
Input Size

In result, the slowest algorithm is selection sort then quick sort and fastest algorithm is bucket sort algorithm.

For the searching algorithms the fastest algorithm binary search, then linear seach for sorted data and the slowest linear search for random data.

In my assignment, I use array list instead of array. I noticed that, using a array list instead of aray caused output to take longer to produce results

References

- piazza
- https://stackoverflow.com/questions/2572868/how-to-time-java-program-execution-speed
- https://www.javatpoint.com/how-to-sort-arraylist-in-java
- https://www.geeksforgeeks.org/quick-sort/
- https://www.javacodeexamples.com/java-arraylist-get-random-elements-example/971