HACETTEPE ÜNİVERSİTESİ

COMPUTER ENGINEERING BBM 204

ASSIGNMENT I

Student: Abdullah Atahan Türk

Student ID: 21827943

Topic: Algorithm Complexity Analysis

Problem Statement

Algorithms are of paramount importance in computer science and computer engineering. Algorithm design in many software products is done by considering the efficiency, speed and memory usage of the algorithm. In this assignment, we aimed to measure the speed and memory usage of sort and search algorithms in different situations, in short, their efficiency. For this, we theoretically compared the space complexity and time complexities of the algorithms. In our experiments, we measured the speed of the algorithms.

Algorithm Implementations In Java

Selection Sort

```java
public class SelectionSort {

    public static void selection(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n-1; i++) {
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
}
```

Quick Sort

```java
public class QuickSort {

    public static void quickSort(int[] arr) {
        if (arr == null || arr.length <= 1) {
            return;
        }
        int[] stack = new int[arr.length];
        int top = -1;
        stack[++top] = 0;
        stack[++top] = arr.length - 1;
        while (top >= 0) {
            int end = stack[top--];
            int start = stack[top--];
            int pivot = partition(arr, start, end);
            if (pivot - 1 > start) {
```

```
                stack[++top] = start;
                stack[++top] = pivot - 1;
            }
            if (pivot + 1 < end) {
                stack[++top] = pivot + 1;
                stack[++top] = end;
            }
        }
    }

    public static int partition(int[] arr, int start, int end) {
        int pivot = arr[end];
        int i = start - 1;
        for (int j = start; j < end; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, end);
        return i + 1;
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

Bucket Sort

```
public class BucketSort {
    public static void bucketSort(int[] arr) {
        int n = arr.length;
        int numOfBuckets = (int) Math.ceil(Math.sqrt(n));
        int max = Arrays.stream(arr).max().getAsInt();

        List<Integer>[] buckets = new List[numOfBuckets];
        for (int i = 0; i < numOfBuckets; i++) {
            buckets[i] = new ArrayList<Integer>();
        }
        for (int k : arr) {
            int bucketIndex = (int) ((k * 1.0 / max) * (numOfBuckets - 1));
            buckets[bucketIndex].add(k);
        }
        for (int i = 0; i < numOfBuckets; i++) {
            Collections.sort(buckets[i]);
        }
        int index = 0;
        for (int i = 0; i < numOfBuckets; i++) {
            for (int j = 0; j < buckets[i].size(); j++) {
                arr[index++] = buckets[i].get(j);
            }
        }
    }
}
```

Linear Search

```java
public class LinearSearch {

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

Binary Search

```java
public class BinarySearch {

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

Time Result Tables

According to the experiments I have done, I have listed the results we obtained in different input sizes and different array sequences in tables. The results of sorting algorithms are in ms, and the results of searching algorithms are in ns.

Sorting on Random Data

Input Size

Algorithm	500	1000	2000	4000	8000	16K	32K	64K	128K	250K
Selection	0	0	2	7	30	108	398	1567	6279	23972
Quick	0	0	0	0	0	1	3	7	19	33
Bucket	0	0	1	1	4	5	6	11	21	48

Sorting on Sorted Data

Input Size

Algorithm	500	1000	2000	4000	8000	16K	32K	64K	128K	250K
Selection	0	0	1	3	15	39	137	529	2101	8448
Quick	0	0	2	9	31	98	364	1491	5969	20286
Bucket	0	0	0	0	0	0	1	1	2	5

Sorting on Reverse Sorted Data

Input Size

Algorithm	500	1000	2000	4000	8000	16K	32K	64K	128K	250K
Selection	0	1	4	18	77	315	1236	4879	19692	75170
Quick	1	0	1	5	24	105	426	1720	7122	26533
Bucket	0	0	0	0	0	0	1	2	3	10

Searching

Input Size

Algorithm	500	1000	2000	4000	8000	16K	32K	64K	128K	250K
Linear (random)	1754	589	722	865	1649	3249	5680	11498	21914	36133
Linear (sorted)	3059	1221	789	1116	1900	3687	6871	14263	26792	58837
Binary (sorted)	1629	851	473	410	1020	1220	1629	1969	2215	2574

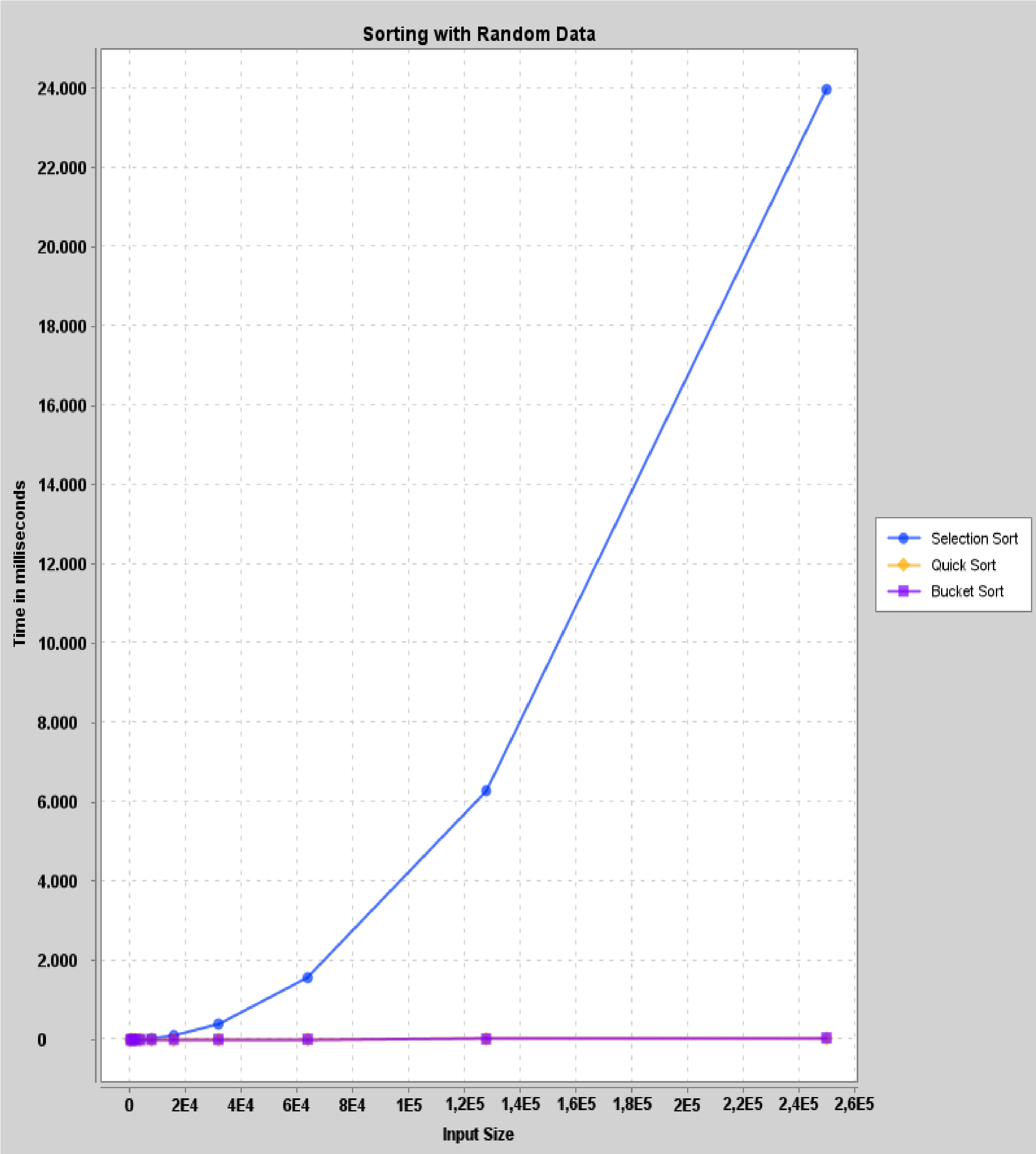
Computational Complexity Table

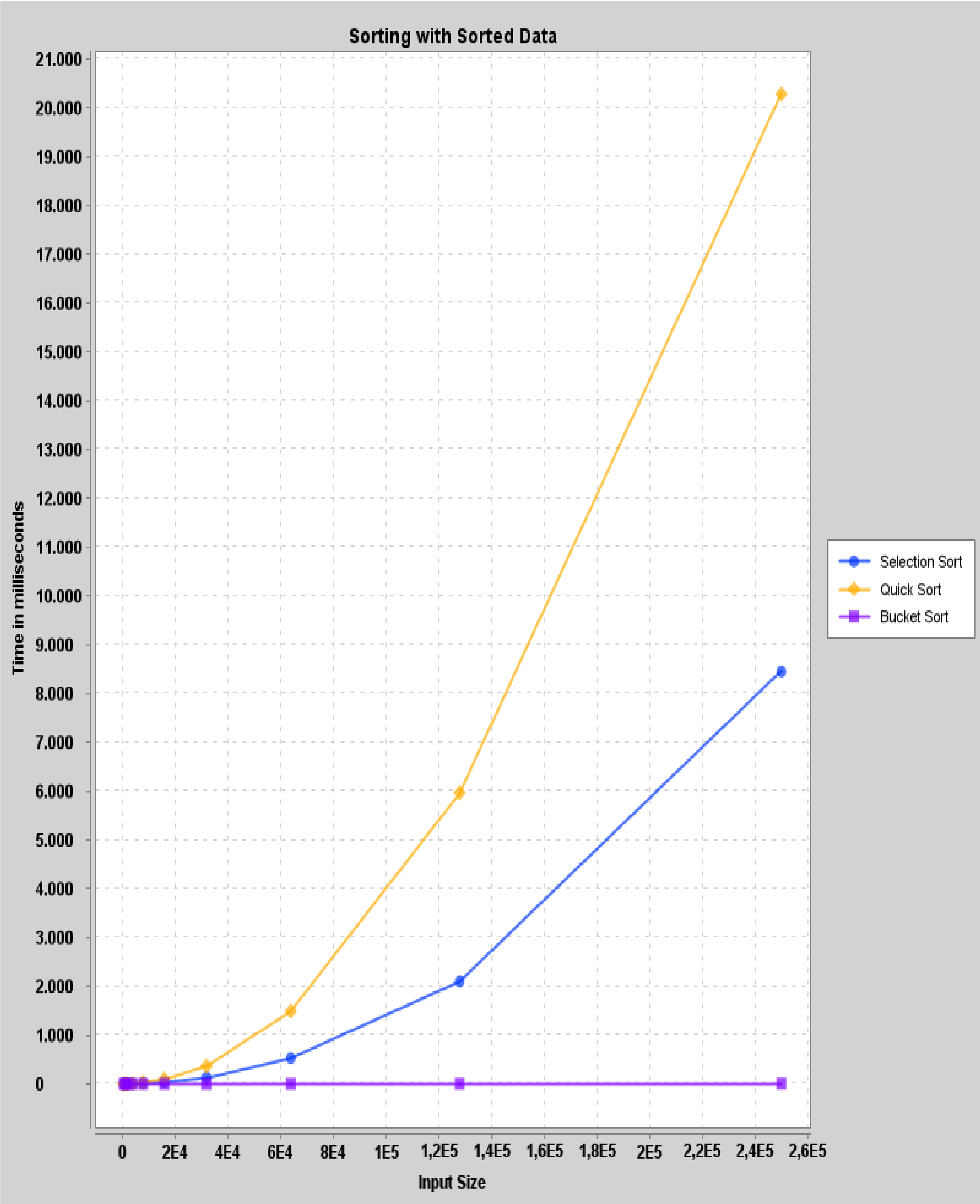
Algorithm	Best Case	Average Case	Worst Case
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Bucket Sort	$\Omega(n)$	$\Theta(n+k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$
Binary Search	$\Omega(1)$	$\Theta(n)$	$O(n)$

Auxiliary Space Complexity Table

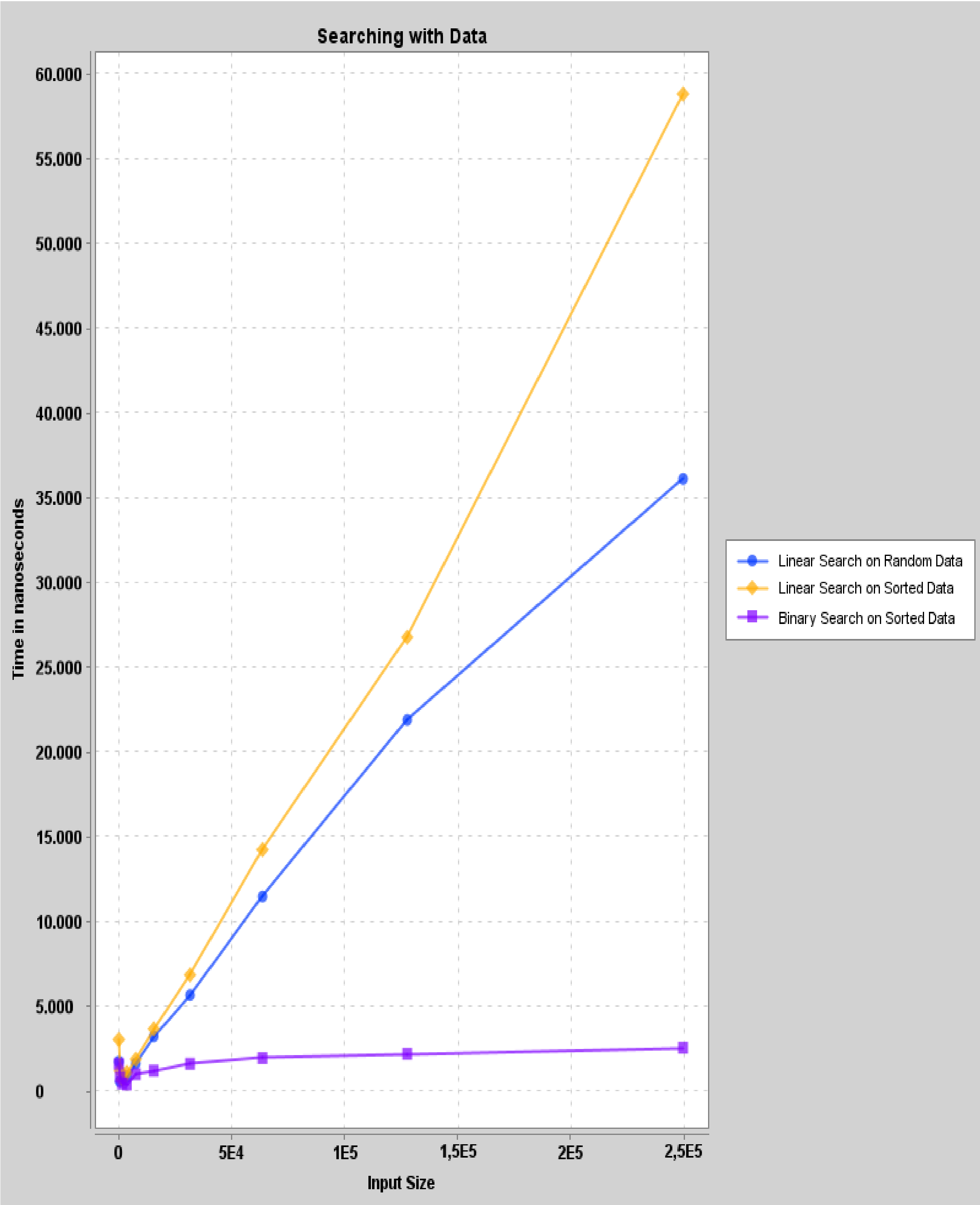
Algorithm	Aux Space Comp.
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

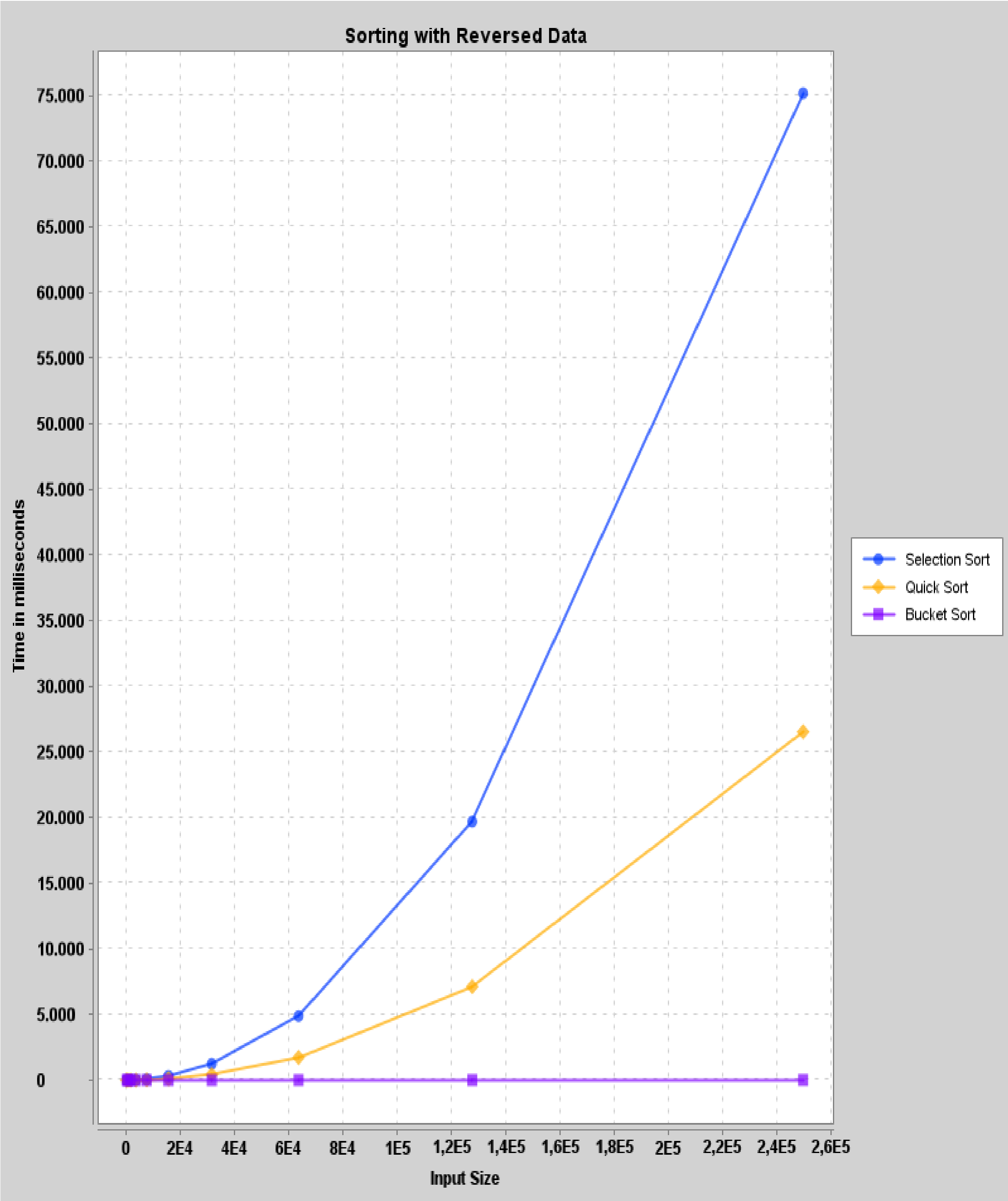
Plots Of The Experiments





Sorting with Sorted Data
Time in milliseconds
21.000
20.000
19.000
18.000
17.000
16.000
15.000
14.000
13.000
12.000
11.000
10.000
9.000
8.000
7.000
6.000
5.000
4.000
3.000
2.000
1.000
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
2,6E5
Input Size
Selection Sort
Quick Sort
Bucket Sort


Searching with Data
60.000
55.000
50.000
45.000
40.000
35.000
30.000
25.000
20.000
15.000
10.000
5.000
0
Time in nanoseconds
0
5E4
1E5
1,5E5
2E5
2,5E5
Input Size
Linear Search on Random Data
Linear Search on Sorted Data
Binary Search on Sorted Data



Sorting with Reversed Data
Time in milliseconds
75.000
70.000
65.000
60.000
55.000
50.000
45.000
40.000
35.000
30.000
25.000
20.000
15.000
10.000
5.000
0
0
2E4
4E4
6E4
8E4
1E5
1,2E5
1,4E5
1,6E5
1,8E5
2E5
2,2E5
2,4E5
2,6E5
Input Size
Selection Sort
Quick Sort
Bucket Sort

Resulst and Analysis Discussion

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

For **selection sort**, the best case is sorting on sorted data, average cas is sorting on random data and worst case is reverse sorted data. Theoretically its complexity is always O(n^2). But in the best case altough the algorithm will perform n comparisons, it will make no swaps. So its actual running time will be shorter than the other two cases. In the average case and the worst case the algorithm will perform n*(n-1)/2 comparisons and n swaps. But in actual running time, there can be different results.

For **quick sort**, theoretically the best case is sorting on sorted data, the average case is sorting on random data and the worst case is sorting on reverse sorted data. Because in the best case, partitioning will create balanced partitions, in the average case the partitioning will create roughly balanced partitions and in the worst case the partitioning will create imbalanced partitions. But in my experiments, I get the best results in randomly sorted data. I researched for it and I found some possible reasons:

- Input size may effect the results. Altough I tried different input sizes, it may not be enough.
- Implementation of the algorithm may affect the results. Maybe I coulnd't implement the algorithm efficiently enough.
- Altough I tried the algorithm 10 times, it may not be enough.
- Data may effect the results. Maybe the data I used may have properties that make it easier or harder for quick sort to sort, which can affect the running time.

For **bucket sort**, theoretically the best case is sorting on sorted data, the average case is on sorting on random data and the worst case is sorting on reversed sorted data. But in my results, I got the worst case in random data and the average case in reversed sorted data. For Bucket sort to perform well, the data must be as well uniformly distiributed. So maybe the data I used is not well uniformly distiributed and when I reverse sort it, its distiribution became better.

In search algorithms, I always get a bad result in the first input size. So I wonder why that happens and I start the experiment with 250 input size just for see the difference. And when I star with 250 input size, I get much better results in 500 input size. So I guess when initiating the search algorithm for the first time, it loses some time.

I got the fastest resuls in **binary search** which worked on the sorted data. Then second best result that I got was the **linear search** on sorted data and the worst result is the **linear search** on random data. I searched a random value every time, so obviously it effected my results but I ran the algorithm 1000 times so I think choosing the searching element randomly is fair.

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

Unfortunately, they didn’t always match. As I mentioned above, sometimes I got different results. The results obtained in these experiments depend on many variables. The way the algorithm is implemented, the dataset on which the experiment is performed, the hardware of the device on which the experiment is performed, etc. There are many different factors that affect the results.