Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 25, 2023

Student name:
Yunus Emre Bayraktar

Student Number:
b2210765023

1 Problem Definition

In today's digital age, there is an an enormous amount of data and being able to effectively search through this data is very important for computing. At this point some searching algorithms like search and merge sort need sorted datasets. However, sorting huge datasets can be a challenging task, and different datasets may require different sorting algorithms to achieve a good performance. Therefore in this experiment the efficiency of different algorithms will be observed by using them to sort datasets of varying sizes and types.

2 Solution Implementation

To see the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities, three different sorting algorithms and two different searching algorithms were used. First two sorting algorithms are 'Selection Sort' and 'Quick Sort' which are comparison based algorithms. The third is 'Bucket Sort' which is a non-comparison based sorting algorithm. As searching algorithms 'Linear Search' and 'Binary Search' were used.

2.1 Selection Sort Algorithm

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and swapping it with the first element of the unsorted part. Selection sort has a time complexity of $O(n^2)$, making it inefficient for large datasets.

```
public static long sort(int[] array, int size) {
        //Start timer
        long start = System.nanoTime();

            //Perform sorting
        for (int i = 0; i < size-1; i++) {
            int min = i;

            for (int j = i+1; j < size; j++) {
                if(array[min] > array[j])
                    min = j;
        }
            int temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }

            //End timer
        long end = System.nanoTime();

            //Return runtime in nanoseconds
        return (end - start);
    }}
```

2.2 Quick Sort Algorithm

Quick sort is an algorithm that uses a divide-and-conquer approach to sort an array of elements. It works by partitioning the array into two sub-arrays based on a pivot element and recursively sorting the sub-arrays. It has a time complexity of O($nlogn$) in the average case and O(n^2) in the worst case.

```
public static long sort(int[] array, int low, int high) {
        //Start timer
        long start = System.nanoTime();

        //Perform sorting
        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(array, low, high);

            if (pivot-1 > low) {
                stack[++top] = low;
                stack[++top] = pivot - 1;
            }

            if (pivot + 1 < high) {
                stack[++top] = pivot + 1;
                stack[++top] = high;
            }
        }
        //End timer
        long end = System.nanoTime();

        //Return runtime in nanoseconds
        return (end - start);
    }

    private static int partition(int array[], int low, int high) {
        int pivot = array[high];
        int i = low - 1;


```

```
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }
}
```

2.3 Bucket Sort Algorithm

Bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets based on their values, and then sorting each bucket individually using another sorting algorithm. Bucket sort has a time complexity of $O(n + k)$, where n is the number of elements in the array and k is the number of buckets.

```
public static long sort(int[] array) {

  //Start timer
        long start = System.nanoTime();

        //Perform sorting
        int bucketAmount = (int) Math.sqrt(array.length);
        int maxInt = getMax(array);

        @SuppressWarnings("unchecked")
        List<Integer>[] buckets = new List[bucketAmount];
        for (int i = 0; i < bucketAmount; i++) {
            buckets[i] = new LinkedList<Integer>();
        }

        for(int num : array)
        buckets[hash(num, maxInt, bucketAmount)].add(num);

        for(List<Integer> bucket : buckets)
        Collections.sort(bucket);

        int index = 0;

```

```
        for(List<Integer> bucket : buckets)  {
            for(int num : bucket)  {
                array[index++] = num;
            }
        }

        //End timer
        long end = System.nanoTime();

        //Return runtime in nanoseconds
        return (end - start);

    }

    private static int hash(int num,int max, int bucketAmount)  {
    return (int) Math.floor((num/max*(bucketAmount-1))) ;
    }
}
```

2.4 Linear Search Algorithm

Linear search is a simple search algorithm that works by checking each element in an array until a match is found or the end of the list is reached. It has a time complexity of O(n)

```
public static int linear(int[] array, int x) {
        int size = array.length;

        for (int i=0; i < size; i++) {
            if (array[i] == x)
                return i;
        }
        return -1;
}
```

2.5 Binary Search Algorithm

Binary search is a search algorithm that works by dividing a sorted array into halves and narrowing down the search interval until the target element is found or the interval becomes empty. It has a time complexity of O($logn$)

```
public static int binary(int[] array, int x) {

        int low = 0;
        int high = array.length-1;


```

```
        while (high - low > 1) {
            int mid = (high+low)/2;

            if (array[mid] < x)
                low = mid + 1;
            else
                high = mid;
        }
        if (array[low] == x)
            return low;
        else if (array[high] == x)
            return high;

        return -1;
    }
```

3 Results, Analysis, Discussion

3.1 Complexity Analysis

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	0.47	0.40	1.01	3.00	11.08	43.18	172.20	705.59	2723.02	10148.08
Quick sort	0.07	0.09	0.19	0.34	0.42	0.89	2.87	10.35	31.40	53.22
Bucket sort	0.23	0.15	0.30	0.80	1.43	5.26	5.87	8.91	19.51	48.44
	Sorted Input Data Timing Results in ms									
Selection sort	0.46	0.17	0.66	2.59	10.25	40.99	162.19	652.94	2623.87	10162.03
Quick sort	0.09	0.37	1.44	5.75	22.77	80.35	325.14	1293.16	5209.23	19836.29
Bucket sort	0.01	0.01	0.03	0.05	0.11	0.22	0.45	0.90	2.08	10.32
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.43	0.33	0.40	0.30	0.42	0.69	1.22	2.47	6.99	17.17
Quick sort	0.08	0.10	0.11	0.14	0.24	0.64	1.99	1.95	4.42	10.57
Bucket sort	0.02	0.03	0.06	0.09	0.16	0.31	2.40	1.80	4.62	9.63

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)	1453	1272	427	394	645	1143	2145	4456	8560	14538
Linear search (sorted data)	54	94	120	315	497	1072	2811	6194	12377	24978
Binary search (sorted data)	194	79	84	76	71	85	113	140	185	228

Complexity analysis tables are given in Table 3 and Table 4

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	$\Omega(n)$	$\Theta(n+k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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	$O(1)$

3.2 Result Plotting and Analysis

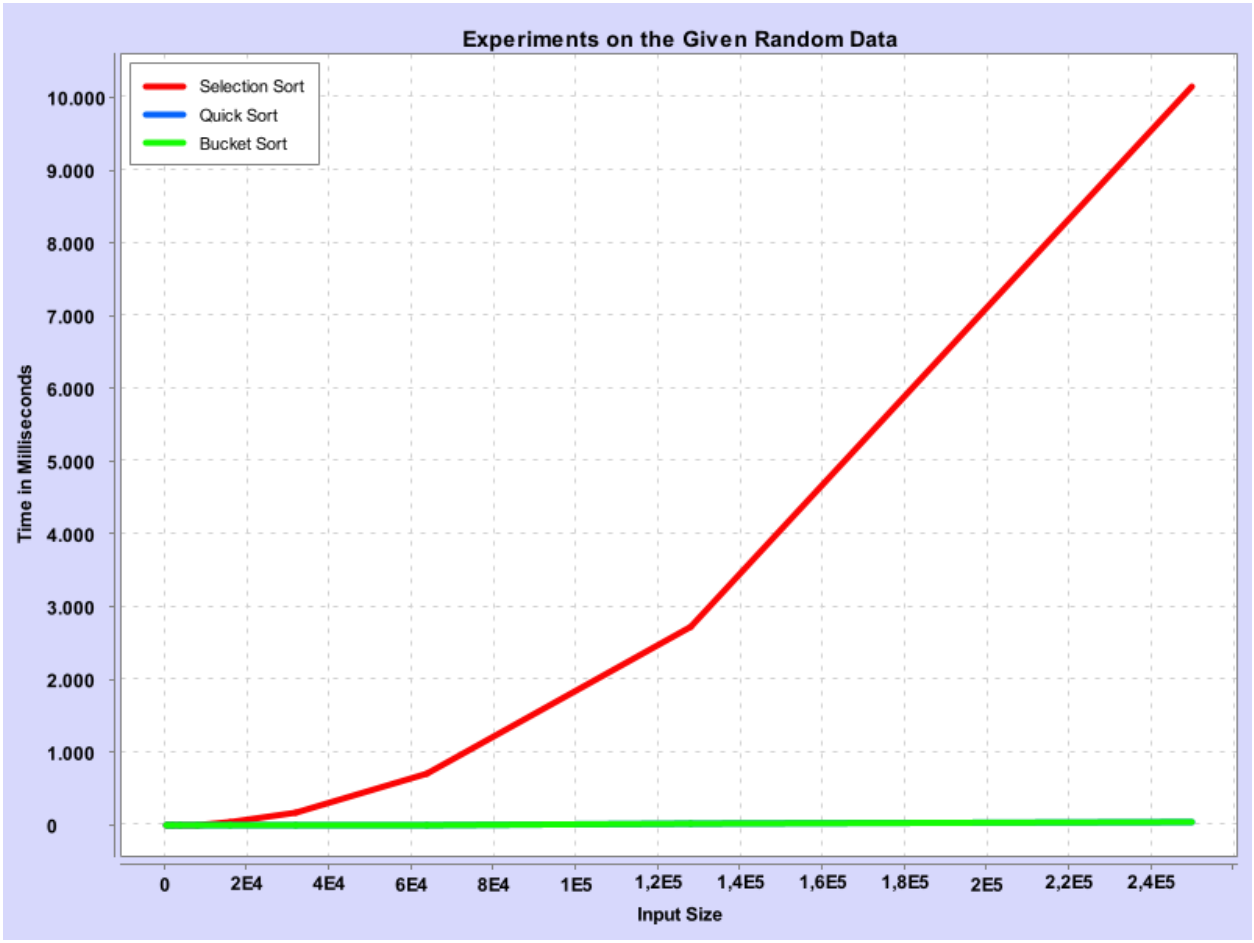

Figure 1: Experiments on the given random data.

This is an average case scenario for all the sorting algorithms. So it is expected to see a $O(n^2)$ complexity for Selection Sort, and a $O(nlogn)$ complexity for both Quick Sort and Bucket Sort. As seen in the figure 1 the obtained results from the experiment match their theoretical asymptotic complexities.

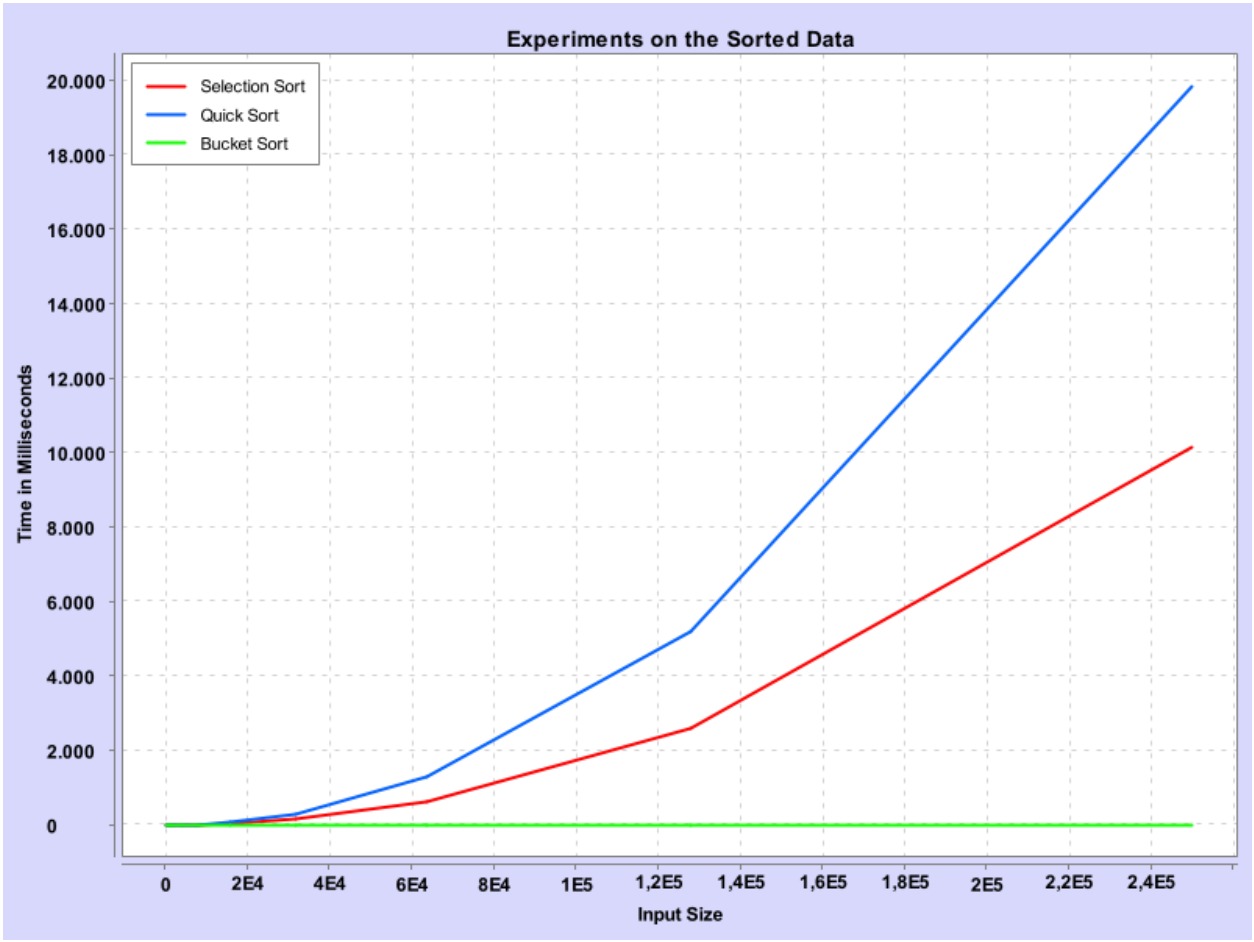

Figure 2: Experiments on the sorted data.

This is an average case scenario for Bucket Sort and Selection Sort but a worst case scenario for Quick Sort. So it is expected to see a $O(n^2)$ complexity for both Selection Sort and Quick Sort, and a $O(nlogn)$ complexity for Bucket Sort. As seen in the figure 2 the obtained results from the experiment match their theoretical asymptotic complexities.

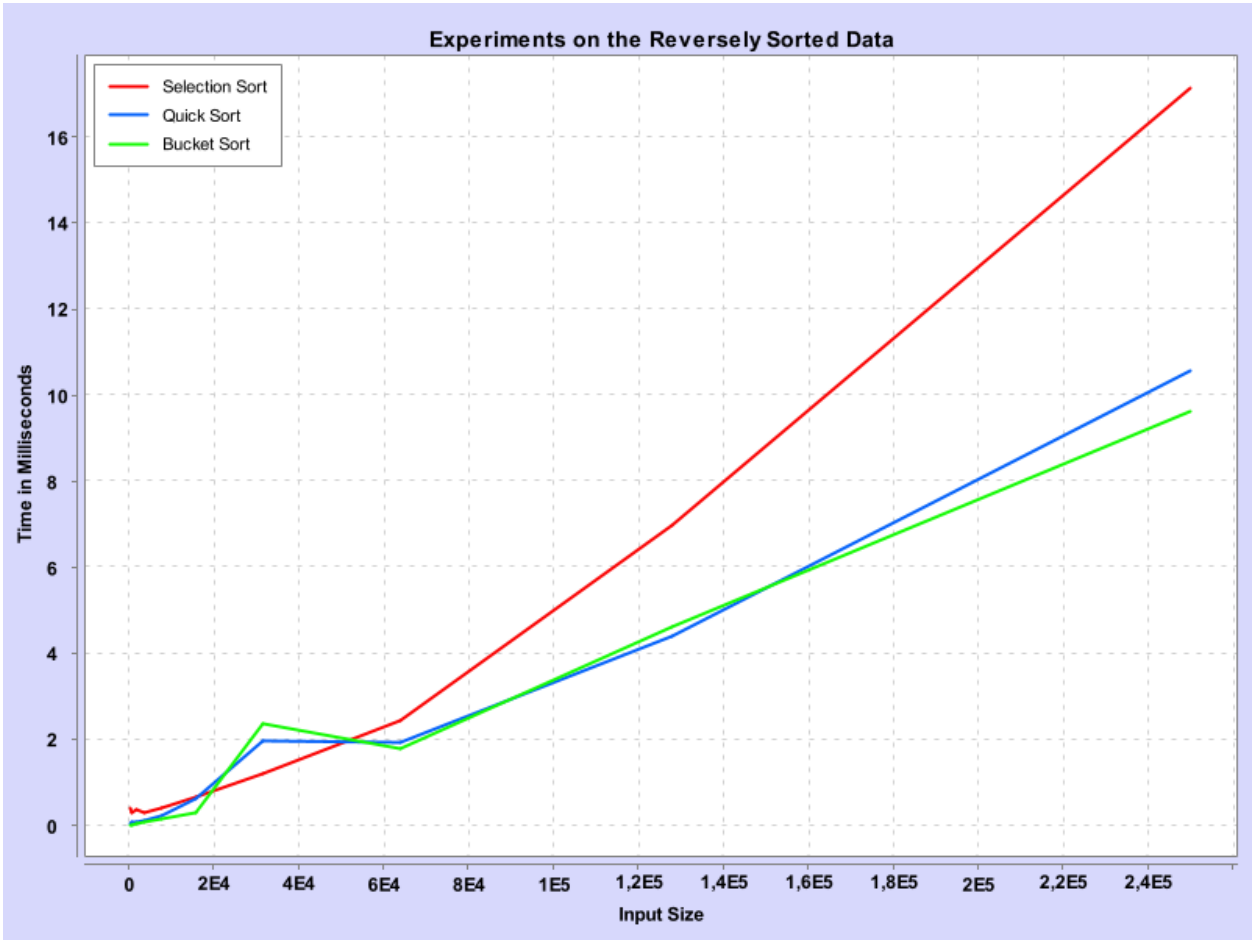

Figure 3: Experiments on the reversely sorted data.

This is an average case scenario for all the sorting algorithms. So it is expected to see a $O(n^2)$ complexity for Selection Sort, and a $O(nlogn)$ complexity for both Quick Sort and Bucket Sort. As seen in the figure 3 the obtained results from the experiment match their theoretical asymptotic complexities.

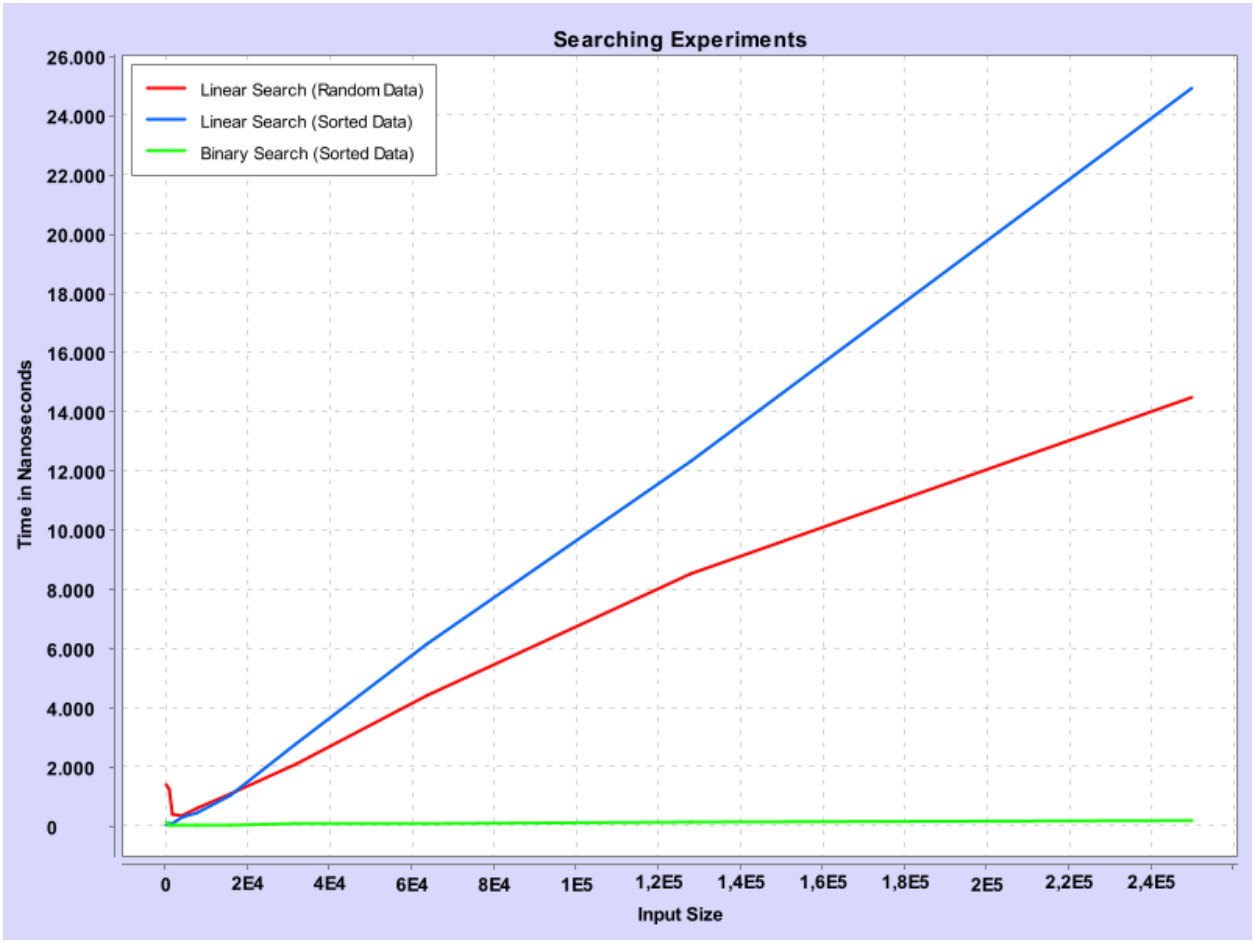

Figure 4: Searching Experiments.

In case of a Linear Search it is expected to see a $O(n)$ complexity both on random data and sorted data. For a Binary Search on sorted data it is expected to see a $O(nlogn)$ complexity. As seen in the figure 4 the obtained results from the experiment match their theoretical asymptotic complexities.

References

- https://www.geeksforgeeks.org/sorting-algorithms/?ref=lbp
- https://knowm.org/open-source/xchart/