Hacettepe University

Computer Engineering Department

BBM204 Programming ASSIGNMENT I - 2023 Spring

Programming Assignment 1

March 30, 2023

Student name:
Furkan Doğmuş

Student Number:
b2200356014

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.

2 Solution Implementation

Sorting and Searching algorithms are implemented in this section.

2.1 BucketSort Algorithm

```

import java.util.Collections;
import java.util.Vector;

public class Bucket {

    static int[] sort(Integer arr[])
    {

        int max = -1;
        for(int i=0;i<arr.length;i++){
            max = Math.max(max,arr[i]);
        }
        int numberOfBuckets = (int) Math.sqrt(arr.length);
        Vector<Integer>[] buckets = new Vector[numberOfBuckets];

        for (int i = 0; i < numberOfBuckets; i++) {
            buckets[i] = new Vector<Integer>();
        }

        for (int i:arr){
            buckets[hash(i,max,numberOfBuckets)].add(i);
        }

        for (int i = 0; i < numberOfBuckets; i++) {
            Collections.sort(buckets[i]);
        }
        int index = 0;
        int[] sortedArr = new int[arr.length];
        for (int i = 0; i <numberOfBuckets ; i++) {
            for (int j = 0 ; j < buckets[i].size(); j++) {
```

```
                sortedArr[index++] = buckets[i].get(j);

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

2.2 QuickSort Algorithm

```
public class Quick  {
    public static void sort(int[] arr, int l, int r)
    {
        while(l<r)
        {
            int q=partition(arr,l,r);
            if (q-l <= r-(q+1))
            {
                sort(arr,l,q);
                l=q+1;
            }
            else
            {
                sort(arr,q+1,r);
                r=q;
            }
        }
    }
    private static int partition(int[] arr, int l, int r) {

        int x = arr[l],i=l,j=r;
        while (true) {
            do {
                i++;
            } while (i < r && arr[i] < x);
            do {
                j--;
            } while (j > l && arr[j] > x);

            if (i < j) {
                swap(arr,i,j);
                i++;
                j--;
```

```java
            } else {
                return j;
            }
        }
    }

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

2.3 SelectionSort Algorithm

```java

public class Selection {
    public static void sort(Integer[] arr,int n){
        int min;
        int max;
        for(int i=0;i<n-1;i++){
            min = i;
            max = i;
            for(int j=i+1;j<n;j++){
                if(arr[j] < arr[min]) {
                    min = j;
                }
                else if(arr[j] > arr[max]) {
                    max = j;
                }
            }

            if (min!=i){
                swap(arr,min,i);
            }

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

2.4 Searching Algorithm

```java
public class SearchAlgorithms {
    public static int linearSearch(int[] arr,int target){
        int res = -1;
        for(int i=0;i<arr.length;i++){
            if (arr[i] == target){
                res = i;
                break;
            }
        }
        return res;
    }
    public static int binarySearch(int[] arr, int target){
        int low = 0,high=arr.length-1;
        while(high-low>1){
            int mid = (high+low) /2;
            if (arr[mid]<target)
                low = mid +1;
            else
                high = mid;
        }
        if(arr[low] == target)
            return low;
        else if(arr[high]== target)
            return high;
        return -1;
    }
    }
```

3 Results, Analysis, Discussion

Running time test results for sorting algorithms are given in Table 1.

Running time test results for search algorithms are given in Table 2.

Complexity analysis tables to complete (Table 3 and Table 4):

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	1.46616	0.80382	2.28082	10.25	31.44	108.95	435.39	1696.9	18488	90133
Quick sort	0.381	0.13232	0.55725	7.37	5.13	45.90	343.95	533	2131.77	7568
Bucket sort	0.4973	0.74596	0.66938	1.15446	2.68465	7.069	8.0093	23.0	36.02	66.54
	Sorted Input Data Timing Results in ms									
Selection sort	0.32978	1.11157	4.0192	16.48	78.90	221.5	603.7	2631	24728	
Quick sort	0.86990	0.78001	2.49959	13.77	29.49	143.77	785.21	2365.55	13097	
Bucket sort	0.75618	0.94080	0.91958	1.95871	4.28903	10.05	11.15	29.30	46.26	84.94
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	1.27138	4.14386	4.56518	13.08	45.82	177.3	789.2			
Quick sort	0.99196	1.15199	4.70	25.34	75.00	310.03	1561.76	6585.46	24042	
Bucket sort	1.11343	1.30607	1.59335	3.27436	9.3013	14.27	18.44	41.39	68.02	120.14

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)	6193	11011	12371	14186	17330	23660	39225	251798	851015	1728966
Linear search (sorted data)	261	670	1381	2697	5260	14007	25718	174858	682852	1620274
Binary search (sorted data)	693	860	1038	1215	1401	1594	1798	2009	2633	3175

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

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	$O(1)$

4 Notes

When reporting algorithms on a computer, there are several factors that can affect the runtime or execution time of the algorithm. Some of these factors include:

Input size: The size of the input data can significantly affect the time taken to execute the algorithm. As the size of the input increases, the time taken to execute the algorithm also increases.

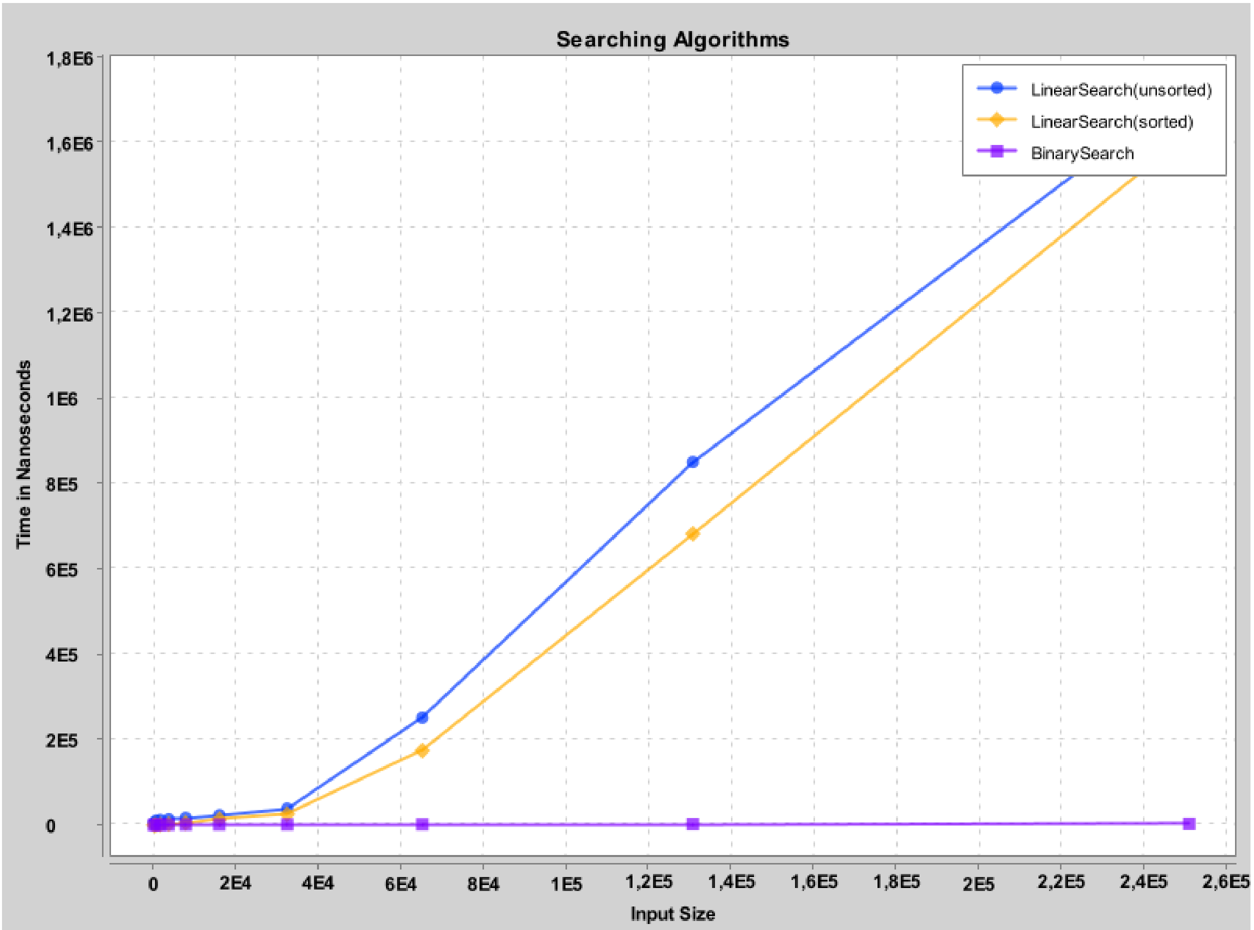

Figure 1: Plot of the searching algorithms.

Algorithmic complexity: The complexity of the algorithm, usually measured in terms of time complexity or space complexity, also affects the runtime. Algorithms with higher time complexity take longer to execute.

Hardware specifications: The specifications of the computer hardware used to run the algorithm also play a role in the runtime. Faster processors and more memory can lead to faster execution times.

Multi-tasking: When running the algorithm, if there are other processes or tasks running simultaneously, this can impact the runtime of the algorithm.

Programming language and implementation: The choice of programming language and the implementation of the algorithm can also affect runtime. Some languages or implementations are faster than others for certain types of algorithms.

It is important to note that during the writing of the report, simultaneous tasks or operations performed can affect the runtime of the algorithm, leading to misleading or inaccurate results.

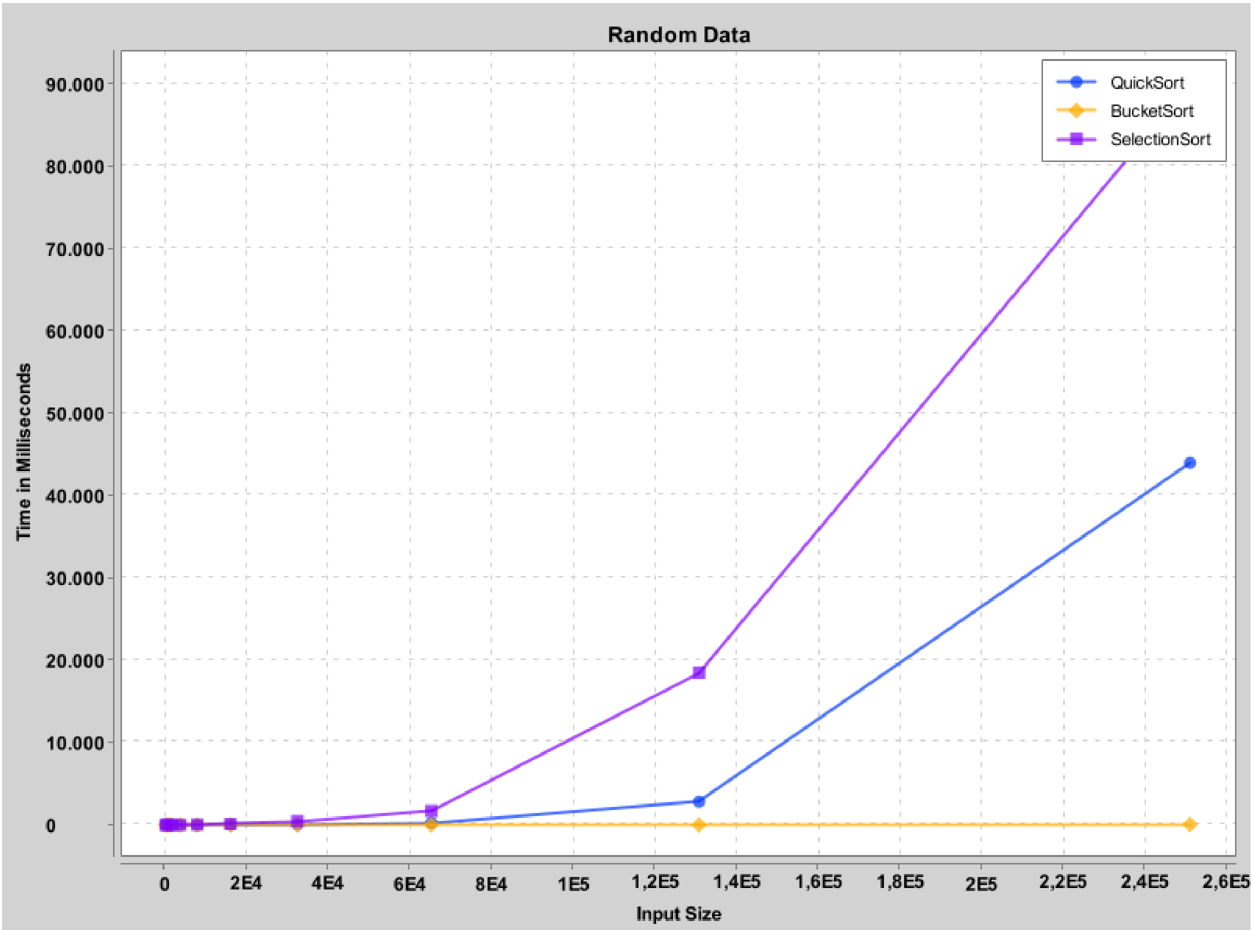

Figure 2: Plot of the Sorting algorithms on random data.

Therefore, it is important to isolate the algorithm and ensure that it is the only process running during testing to obtain accurate results.

References

- https://chat.openai.com/
- https://stackoverflow.com/questions/33884057/quick-sort-stackoverflow-error-for-large-arrays
- My friends posts on Piazza