Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 29, 2023

Student name:
İsmail Ulaş Ünal

Student Number:
b2200356001

1 Problem Definition

In this assignment, we will examine how fast 3 different sorts(Selection sort, quick sort, bucket sort) and 2 different search(linear search, binary search) algorithms work in different sized arrays and different array distribution types (unsorted, sorted, reverse sorted), and how their complexity changes the working speed.

2 Sort Solution Implementation

Sorting algorithms;

2.1 Selection Sort

```
1  public class SelectionSort implements ISort{
2      public long sort(int[] arr) {
3          long startTime = System.nanoTime();
4          int n = arr.length;
5          for (int i = 0; i < n-1; i++) {
6              int minIndex = i;
7              for (int j = i+1; j < n; j++) {
8                  if (arr[j] < arr[minIndex]) { // found min
9                      minIndex = j;
10                 }
11             }
12             swap(arr, minIndex, i);
13         }
14         long endTime = System.nanoTime();
15         return endTime - startTime;
16     }
17 }
```

2.2 Quick Sort

```java
import java.util.Stack;

public class QuickSort implements ISort{
    @Override
    public long sort(int[] arr) {
        long startTime = System.nanoTime();
        if (arr == null || arr.length == 0) {
            return 0;
        }
        Stack<Integer> stack = new Stack<>();
        stack.push(0);
        stack.push(arr.length - 1);

        while (!stack.isEmpty()) {
            int end = stack.pop();
            int start = stack.pop();
            int pivotIndex = partition(arr, start, end);
            if (pivotIndex - 1 > start) {
                stack.push(start);
                stack.push(pivotIndex - 1);
            }
            if (pivotIndex + 1 < end) {
                stack.push(pivotIndex + 1);
                stack.push(end);
            }
        }
        long endTime = System.nanoTime();
        return endTime - startTime;
    }
    private 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;
    }
}
```

2.3 Bucket Sort

```java
import java.util.ArrayList;
import java.util.Collections;

public class BucketSort implements ISort{
    @Override
    public long sort(int[] arr) {
        long startTime = System.nanoTime();

        int numOfBuckets = (int) Math.sqrt(arr.length);
        ArrayList<Integer>[] buckets = new ArrayList[numOfBuckets];

        for (int i = 0; i < numOfBuckets; i++) {
            buckets[i] = new ArrayList<>();
        }
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            max = Math.max(arr[i], max);
        }
        for (int element : arr) {
            int bucketIndex = hash(element, max, numOfBuckets);
            buckets[bucketIndex].add(element);
        }

        int index = 0;
        for (int i = 0; i < numOfBuckets; i++) {
            Collections.sort(buckets[i]);
            for (int element : buckets[i]) {
                arr[index++] = element;
            }
        }
        long endTime = System.nanoTime();

        return endTime - startTime;
    }

    private int hash(double i, double max, int numOfBuckets) {
        return (int) (i / max * (numOfBuckets - 1));
    }
}
```

3 Search Solution Implementation

Search algorithms;

3.1 Linear Search

```
99  public class LinearSearch implements ISearch{
100     @Override
101     public int search(int[] arr, int num) {
102         for (int i = 0; i < arr.length; i++) {
103             if (arr[i] == num) {
104                 return i;
105             }
106         }
107         System.out.println("Cannot find");
108         return -1;
109     }
110 }
```

3.2 Binary Search

```
111 public class BinarySearch implements ISearch{
112     @Override
113     public int search(int[] arr, int num) {
114         int low = 0;
115         int high = arr.length - 1;
116 
117         while (low <= high) {
118             int mid = low + (high - low) / 2;
119 
120             if (arr[mid] == num) {
121                 return mid;
122             }
123 
124             if (num > arr[mid]) {
125                 low = mid + 1;
126             }
127             else {
128                 high = mid - 1;
129             }
130         }
131         System.out.println("cannot find");
132         return -1;
133     }
134 }
```

4 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	0.2	0.2	0.8	3.2	12.4	48.9	195.1	787.7	3167.8	11761.4
Quick sort	0.2	0.1	0.2	0.2	0.4	1.5	5.9	18.5	57.0	102.5
Bucket sort	0.2	0.2	0.2	0.4	1.0	1.9	4.0	5.6	11.5	25.1
	Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.2	0.7	3.0	12.1	48.5	194.3	783.0	3150.0	11822.9
Quick sort	0.3	0.4	1.2	7.2	42.0	167.5	669.2	2708.6	10789.5	41289.6
Bucket sort	0.0	0.0	0.0	0.1	0.2	0.4	0.6	0.5	0.8	1.6
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.2	1.0	4.0	16.0	63.0	249.8	1033.2	4539.7	20974.6
Quick sort	0.2	0.3	0.9	3.4	12.6	26.7	92.9	360.4	1360.7	2876.4
Bucket sort	0.1	0.1	0.0	0.1	0.3	0.5	0.7	0.8	1.7	3.4

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)	59.512	101.698	175.949	320.825	583.21	1023.721	2213.542	4406.294	7978.697	14461.593
Linear search (sorted data)	58.579	106.562	191.368	379.161	737.116	1490.209	3156.87	6144.848	11718.383	23489.544
Binary search (sorted data)	39.347	46.525	50.29	58.038	60.159	62.93	68.231	82.071	86.968	91.729

Complexity analysis tables to complete (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	$\Omega(n + k)$	$\Theta(n + n^2/k + k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

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	$O(1)$

5 Images

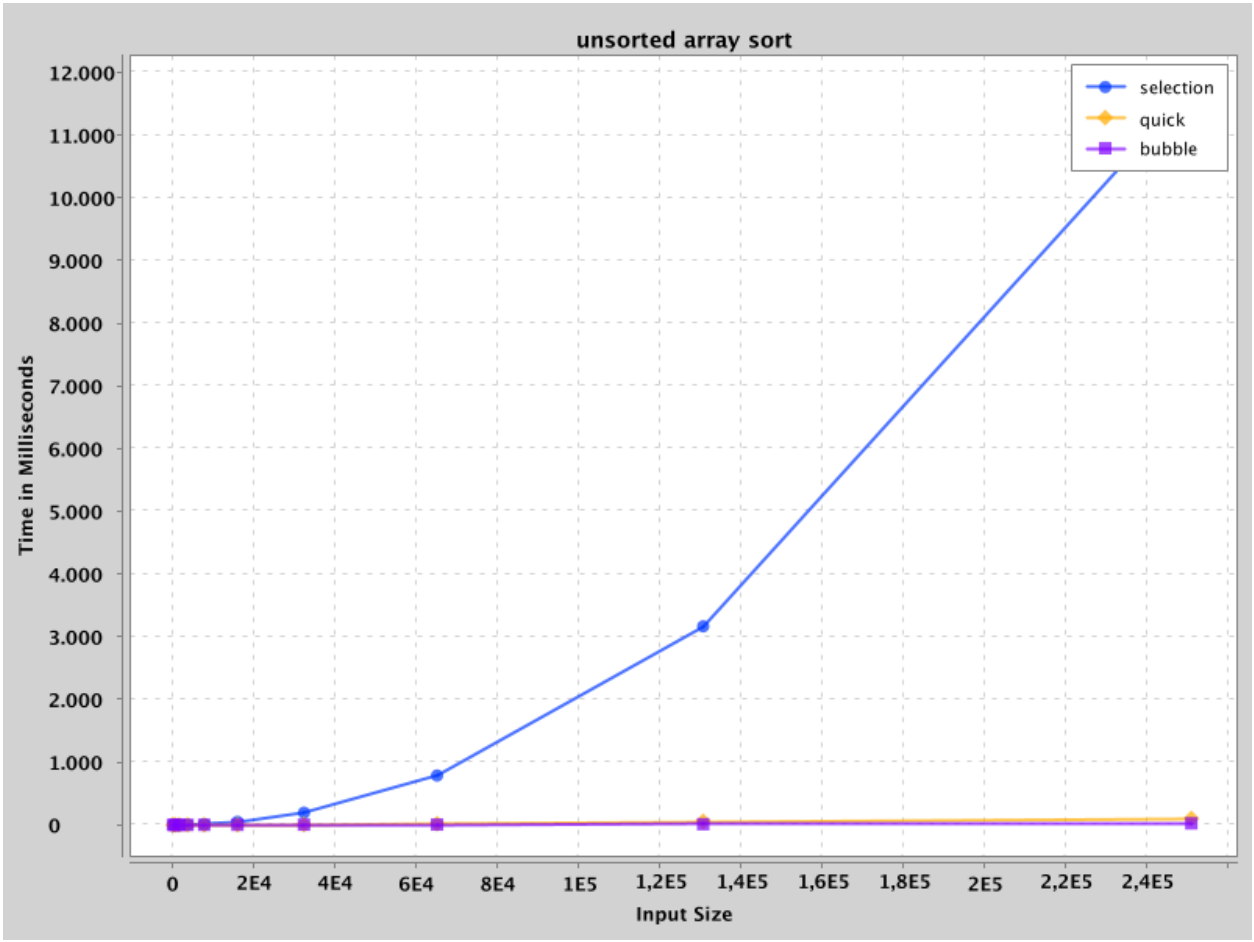

Figure 1: Tests on Random Data.

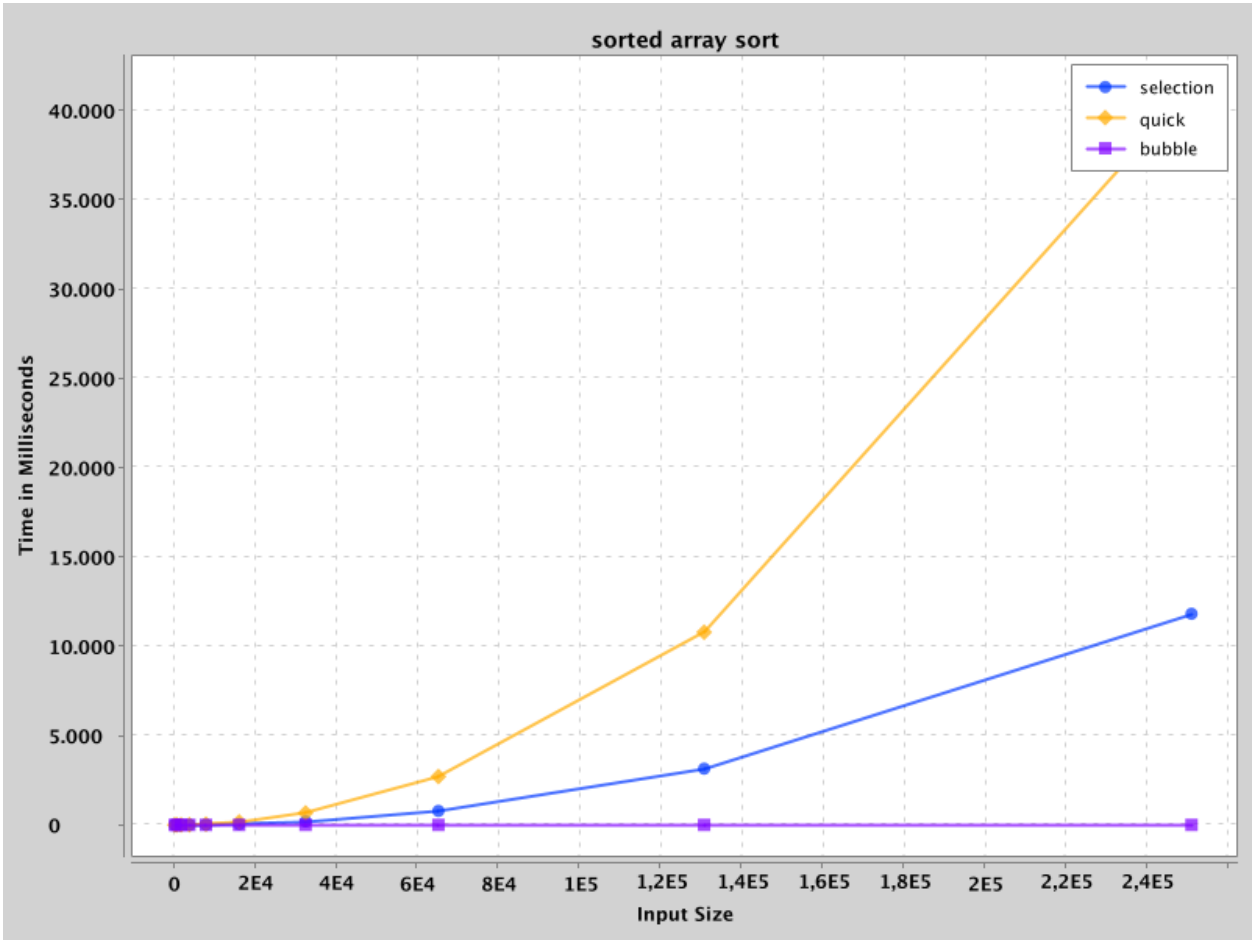

Figure 2: Tests on Sorted Data.

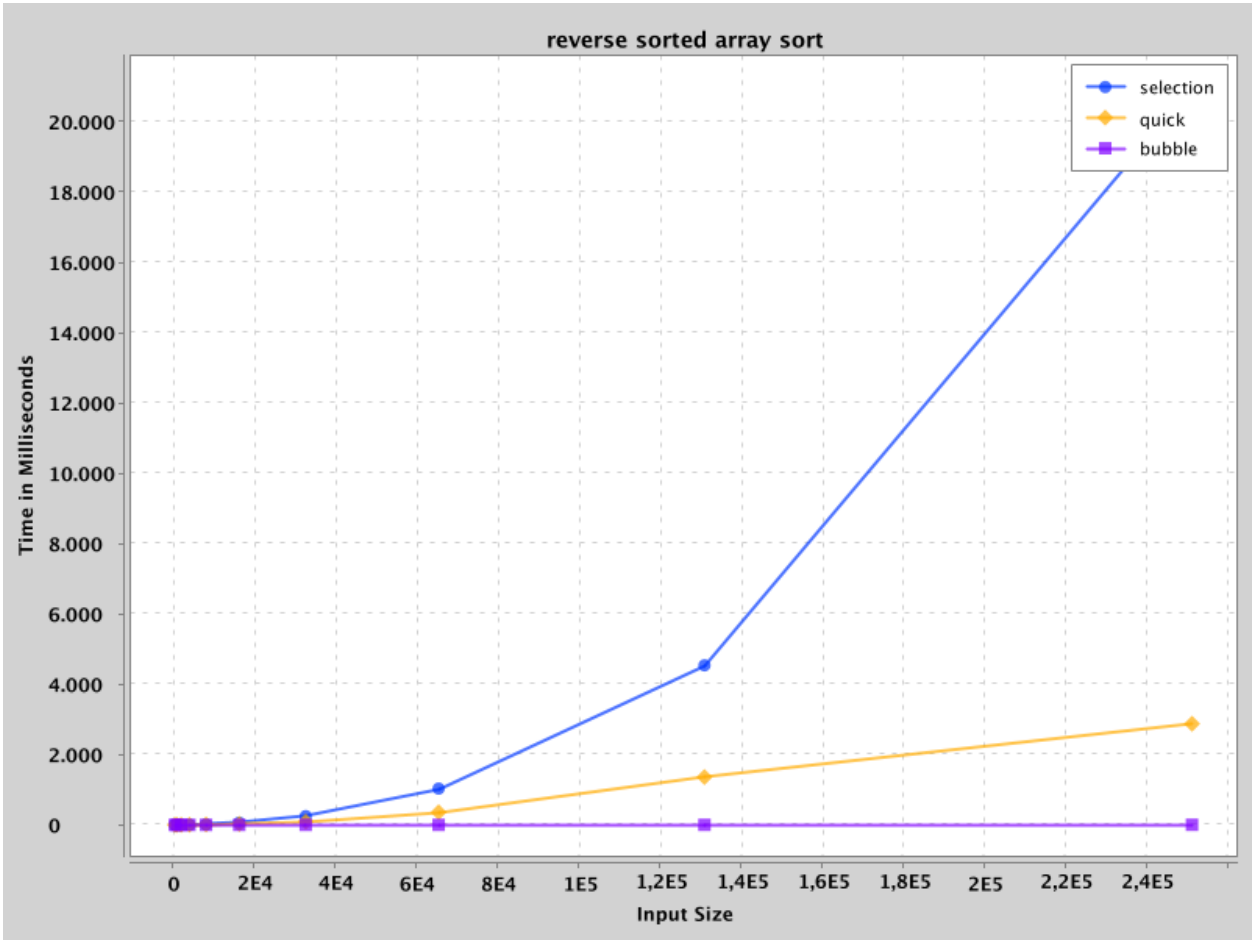

Figure 3: Tests on Reverse Sorted Data.

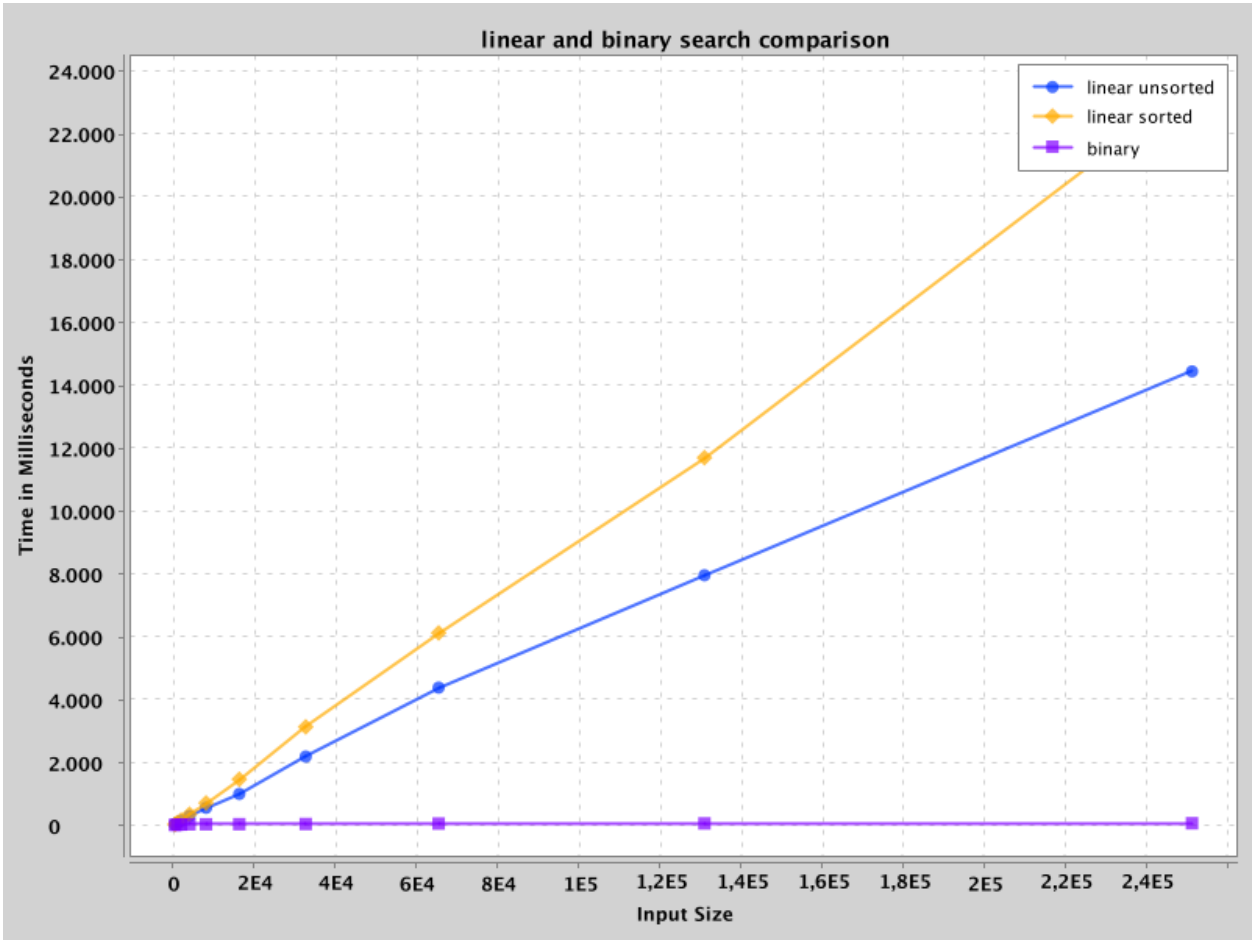

Figure 4: Tests on Search Algorithms.

6 Notes

Selection Sort works in $O(n^2)$ complexity because of the fors in line 5 and line 7. Since this algorithm finds the smallest element in the dataset by traversing the entire array, its complexity is $O(n^2)$ in all cases. But selection sort doesn't use any extra space. When comparing the theoretical result with the practical result, the algorithm works at n^2. If the array size is doubled, $(2n)^2 = 4n^2$, so it should work 4 times slower. As we can see from the table, 0.2 - 0.8 - 3.2 - 3.2 - 12.4 - 48.9 - 195.1 - 787.7 - 3167.8 - 11761.4, that is, it increases about 4 times in each step. So the results are largely similar and each case(reverse, sorted, random) runs at similar speeds. It fits with theorical results.

In order for the quick sort algorithm to be in the best case, the numbers smaller and larger than the pivot must be equal each time, that is, we need to divide the array into two equal parts, as can be seen from the table and graph, this usually happens, but if the array comes in a sorted form Since one piece will have almost no elements and the other will have all the elements, our level count goes from $O(\log n)$ to $O(n)$, and our complexity goes from $O(n \log n)$ to $O(n^2)$. reverse sorted data runs faster than sorted data, but both grow about 4 times larger at each step with some exceptions.

The complexity of the bucket sort algorithm is generally determined by the hash method, not how the array is sorted. If each element falls into a bucket, it will be the worst case, but if the elements are evenly distributed across the buckets, it will be the best case. Space complexity is n + k, k = number of buckets. And It works really well on every type ol data because it's worst case is not bad as quick sort's worst case.

If we compare these 3 sort methods from the table, in an unsorted array it is much faster than quick and bubble selection, but if we try to sort a sorted array again, quick sort will reach the worst case and work almost at the same speed as selection. But the bucket works at almost the same speed in all of them, regardless of how the array is sorted.

When it comes to comparing search algorithms, binary search works much faster because it grows with $O(\log n)$ as in the theoretical result. The reason why binary search is $O(\log n)$ is that it divides the array into two equal parts each time and discards half of them, so it works much faster than going one by one.