Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 30, 2023

Student name:
Mustafa Emre Yildirim

Student Number:
b2200356068

1 Problem Definition

Sorting data efficiently is crucial for improving the performance of other algorithms that rely on sorted input data, such as search and merge algorithms. With the exponential growth of digital information available on the internet, the ability to quickly and effectively search through large datasets has become essential for modern computing. Therefore, the efficiency of a sorting algorithm can be evaluated by testing its performance on datasets with varying sizes and different characteristics that need to be sorted. In summary, efficient sorting algorithms play a critical role in optimizing the performance of various computing tasks and enable us to handle large amounts of data effectively. In this assignment we will implement various algorithms from pseudocode, analyze how they will work on different inputs and make inferences.

2 Solution Implementation

2.1 Selection Sort

```
1  public class Factorial {
2       public static void sort(int[] elements,int dataNumber) {
3       int length = dataNumber;
4       for (int i = 0; i < length - 1; i++) {
5           int minPos = i;
6           int min = elements[minPos];
7           for (int j = i + 1; j < length; j++) {
8               if (elements[j] < min) {
9                   minPos = j;
10                   min = elements[minPos];
11                }
12            }
13              if (minPos != i) {
14                  elements[minPos] = elements[i];
15                  elements[i] = min;
16                }
17          }
18      }
19  }
```

2.2 Quick Sort

```
public static void quickSort(int[] array, int low, int high) {
        int stackLength = high - low + 1;
        int[] stackArr = new int[stackLength];
        int top = -1;
        stackArr[++top] = low;
        stackArr[++top] = high;
        while (top >= 0) {
            high = stackArr[top--];
            low = stackArr[top--];
            int pivot = partition(array,low,high);
            if (pivot-1 > low) {
                stackArr[++top] = low;
                stackArr[++top] = pivot-1;
            }
            if (pivot+1 < high) {
                stackArr[++top] = pivot+1;
                stackArr[++top] = high;
            }
        }
    }
    public 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++;
                swap(array, i, j);
            }
        }
        swap(array, i+1, high);
        return i+1;
    }
    public static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

2.3 Bucket Sort

```
public static void bucketSort(int[] A, int n) {
        int numberOfBuckets = n;
        ArrayList<Integer>[] buckets = new ArrayList[numberOfBuckets];
        for (int i = 0; i < numberOfBuckets; i++) {
            buckets[i] = new ArrayList<Integer>();
        }
        int max = A[0];
        for (int i = 1; i < n; i++) {
            if (A[i] > max) {
                max = A[i];
            }
        }
        for (int i = 0; i < n; i++) {
            int index = hash(A[i], max, numberOfBuckets);
            buckets[index].add(A[i]);
        }
        for (int i = 0; i < numberOfBuckets; i++) {
            Collections.sort(buckets[i]);
        }
        int index = 0;
        for (int i = 0; i < numberOfBuckets; i++) {
            for (int j = 0; j < buckets[i].size(); j++) {
                A[index++] = buckets[i].get(j);
            }
        }
    }
    public static int hash(int i, int max, int numberOfBuckets) {
        return (int) Math.floor(i / (double) max * (numberOfBuckets - 1));
    }
```

2.4 Linear Search

```
public static int linearSearch(int[] arr,int x) {

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

2.5 Binary Search

```
95  public static int binarySearch(int[] arr,int x) {
96          int low=0;
97          int up=arr.length-1;
98          while((up-low)>1) {
99              int mid=(up+low)/2;
100             if (arr[mid]<x) {
101                 low=mid+1;
102             }
103             else {
104                 up=mid;
105             }
106         }
107         if(arr[low]==x) {
108             return low;
109         }
110         else if (arr[up]==x) {
111             return up;
112         }
113         return -1;
114     }
```

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	0,47	1,02	1,61	3,57	8,57	29,09	112,4	505,45	1978,03	7661,75
Quick sort	0,15	0,17	0,27	0,48	0,81	1,41	3,97	11,97	31,92	62,49
Bucket sort	0,39	0,60	0,89	1,43	2,15	3,36	4,36	8,32	15,04	25,65
	Sorted Input Data Timing Results in ms									
Selection sort	0,41	0,34	0,45	1,78	7,18	28,88	119,45	462,65	1861,93	6907,29
Quick sort	0,90	0,39	0,53	1,90	6,29	23,59	91,53	372,36	1540,90	5754,57
Bucket sort	0,35	0,38	0,39	0,56	0,81	1,13	1,26	2,66	4,07	7,94
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0,48	0,93	1,52	6,01	23,95	96,02	383,67	1549,83	6207,16	23985,24
Quick sort	0,21	0,54	2,27	6,87	26,63	98,40	412,41	1521,10	5874,99	23190,73
Bucket sort	0,42	0,53	0,72	1,10	1,62	2,47	3,28	6,46	11,27	17,56

Running time test results for search algorithms are given in Table 2.
Complexity analysis tables to complete (Table 3 and Table 4):

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)	3036	2380	466	763	1589	3071	5989	12073	23828	47029
Linear search (sorted data)	137	227	562	735	1441	2803	5648	10560	23333	44997
Binary search (sorted data)	239	142	154	161	170	180	142	74	372	171

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	$O(n+k)$	$O(n)$	$\Omega(n \log n)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$\Omega(logn)$	$\Omega(logn)$

Table 4: Auxiliary space complexity of the given algorithms.

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

- QUESTION: What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted/searched?
  Selection Sort:
  - Best Case: $O(n^2)$ (when the array is already sorted)
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$ (when the array is reverse sorted)

  Quick Sort:
  - Best Case: $O(n \log n)$ (when the pivot element is always the median)
  - Average Case: $O(n \log n)$
  - Worst Case: $O(n^2)$ (when the array is already sorted or reverse sorted)

  Bucket Sort:
  - Best Case: $O(n+k)$ (when the array elements are uniformly distributed)
  - Average Case: $O(n+k)$

- Worst Case: $O(n^2)$ (when all the elements in the array fall in the same bucket)

Linear Search:

- Best Case: $O(1)$ (when the search element is at the first index of the array)
- Average Case: $O(n)$
- Worst Case: $O(n)$ (when the search element is not present in the array)

Binary Search:

- Best Case: $O(1)$ (when the search element is at the middle index of the sorted array)
- Average Case: $O(\log n)$
- Worst Case: $O(\log n)$ (when the search element is not present in the array)

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

  Yes, they generally match, but there may be some differences due to environmental factors.

4 Notes

- For the quick sort algorithm, if we choose the value randomly, this gives the opportunity to work faster for quick sort.
- The working times of the algorithms are not always the same, the main reasons for this are environmental factors. These may be the current state of the device, memory fullness or CPU usage.
- If we want our algorithm to be fast, we have to sacrifice memory. If we want to keep memory usage low, we should do the opposite.In short, we cannot keep both at a low level at the same time.

References

- https://www.geeksforgeeks.org/quick-sort
- https://en.wikipedia.org/wiki/Bucket-sort

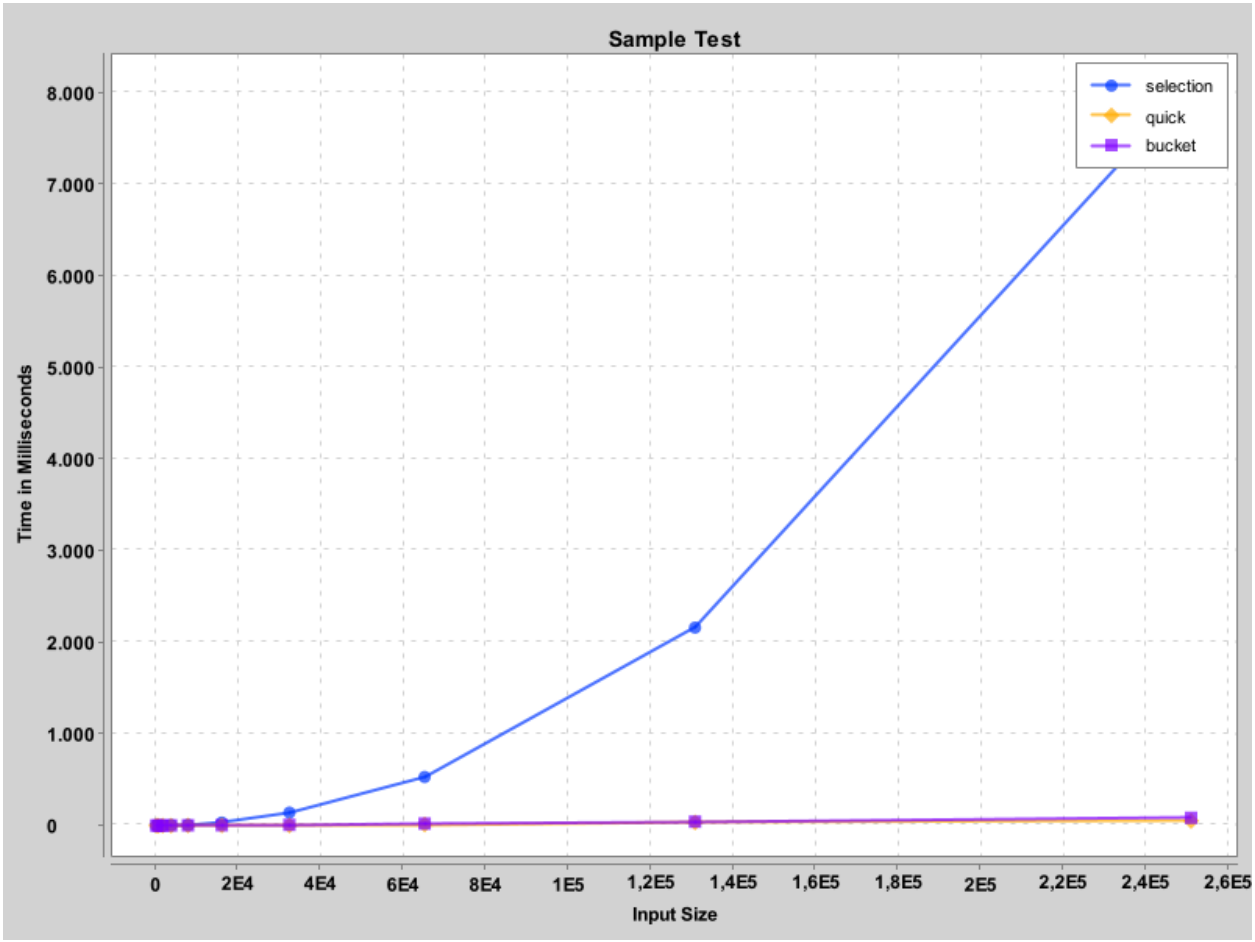

Figure 1: Sorting on Random Data.

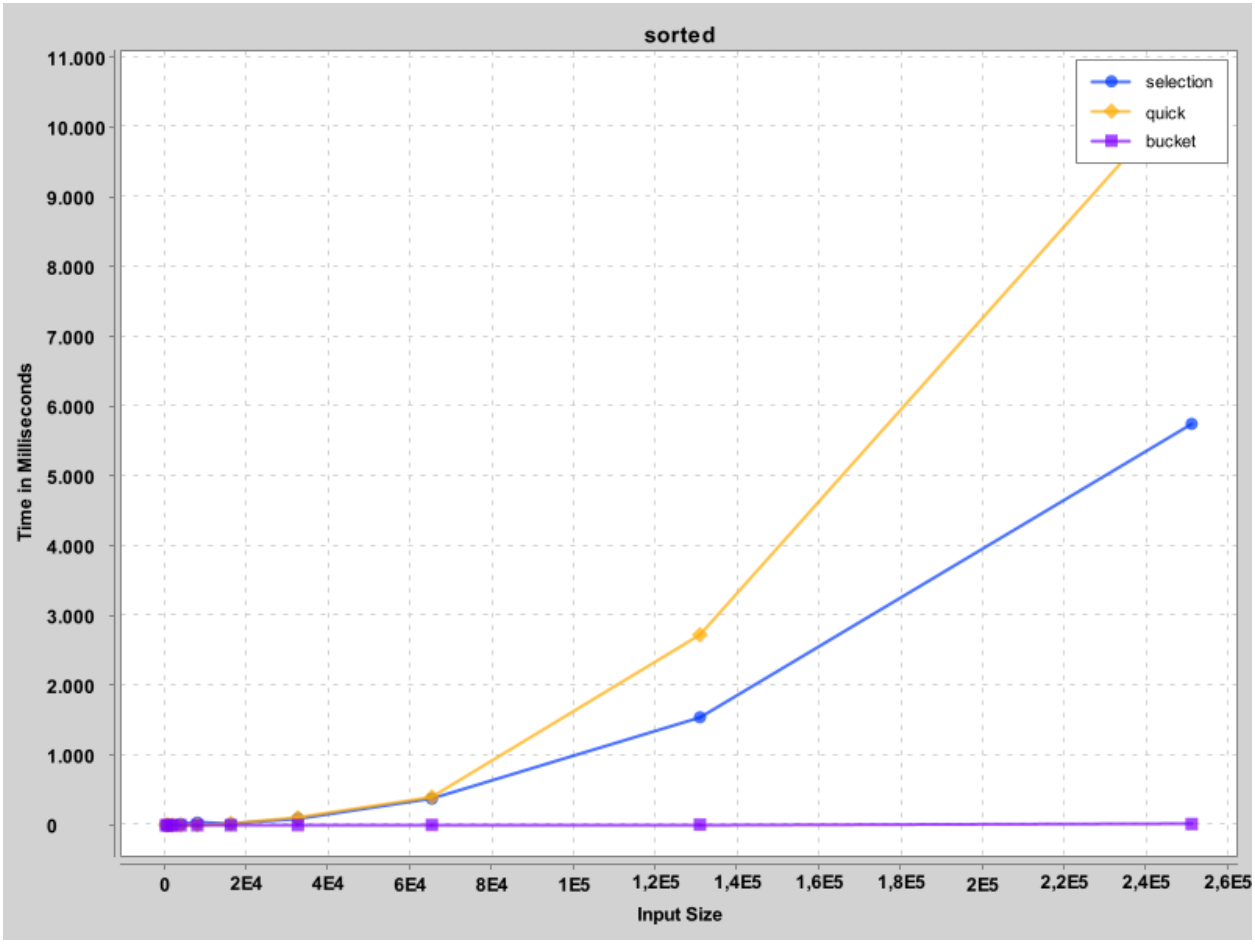

Figure 2: Sorting on Sorted Data..

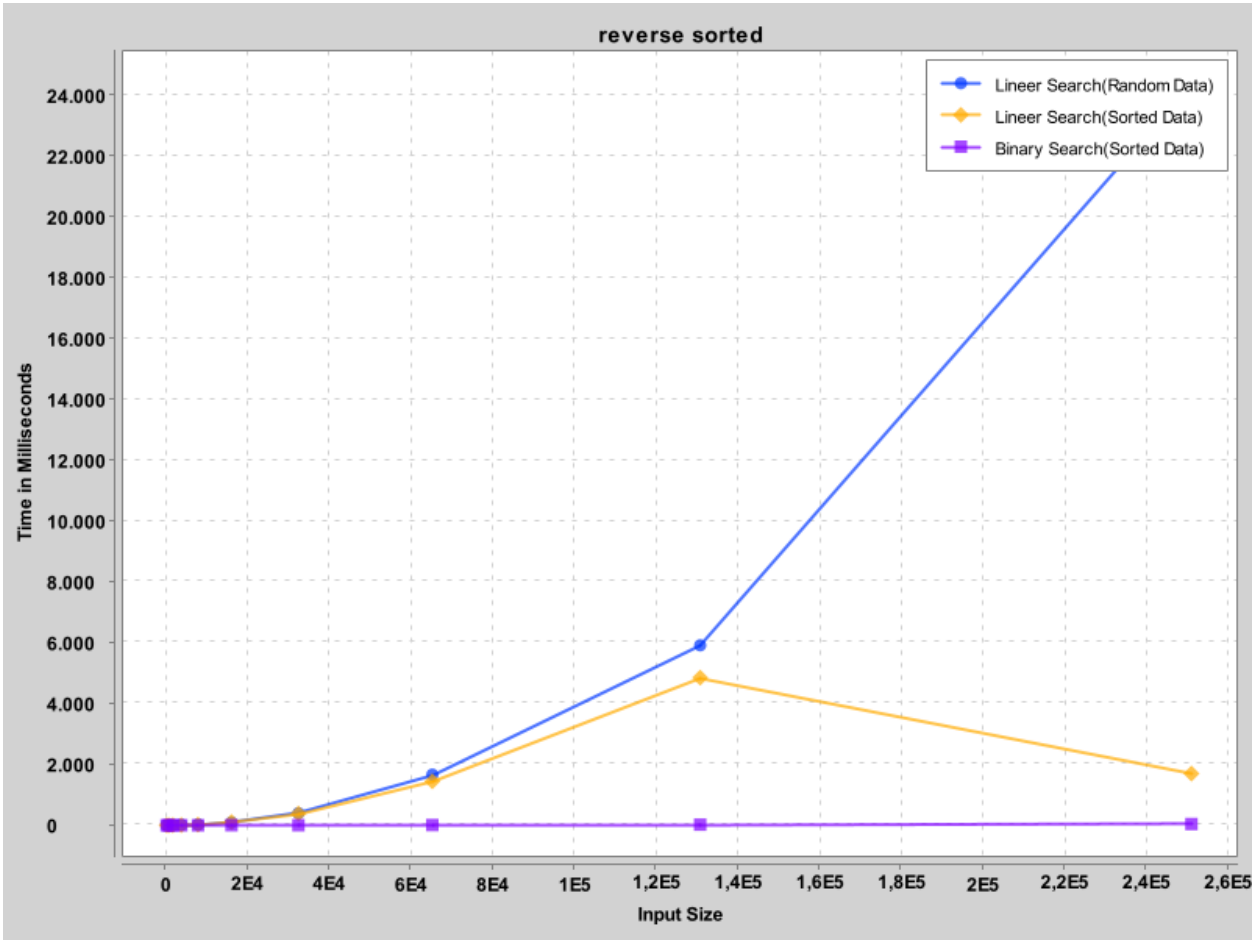

Figure 3: Sorting on Reversed Sorted Data..

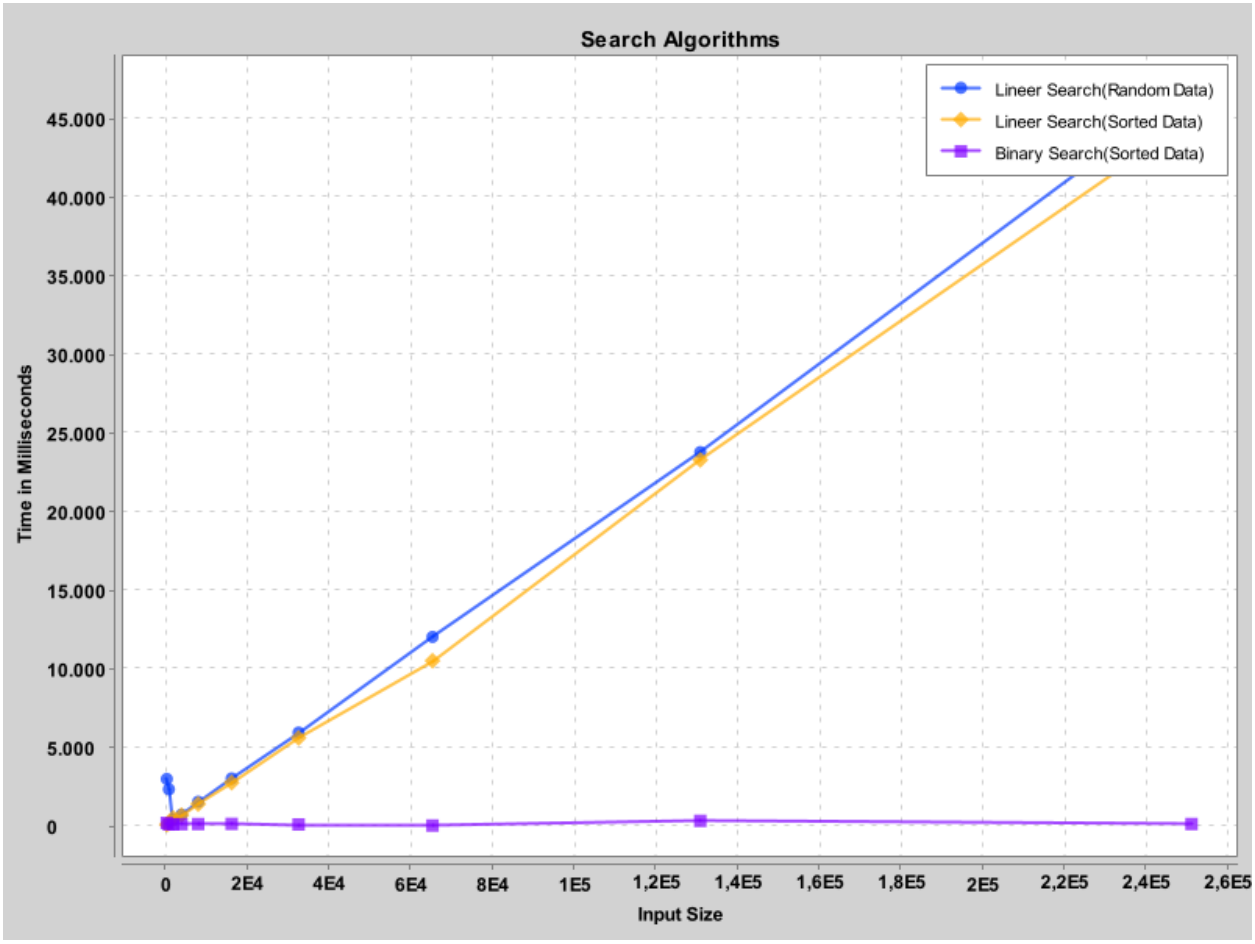

Figure 4: Searching Algortihms.