Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 29, 2023

Student name:
Çağrı Çakıroğlu

Student Number:
b2200765005

1 Problem Definition

Compare the time and space complexity of different sorting algorithms for a given input size. Evaluate their performance in terms of the amount of memory required and the time taken to sort the input data. Provide a comparison of the algorithms to determine which algorithm is most efficient for the given input size

2 Solution Implementation

I used Selection Sort, Quick Sort, Bucket Sort, Binary Search, Linear Search algorithms in this problem. Implementations are given below

2.1 Selection Sort Algorithm

Selection sort is a basic sorting algorithm that repeatedly selects the smallest unsorted element and places it in its correct position in the sorted portion of the array. Despite its simplicity, making it inefficient for large input sizes.

```
1  public class Selection {
2      public static void sort(int[] array) {
3          for (int i = 0; i < array.length - 1; i++) {
4              int min = i;
5              for (int j = i + 1; j < array.length; j++) {
6                  if (array[j] < array[min]) {
7                      min = j;
8                  }
9              }
10             if (min != i) {
11                 int temp = array[i];
12                 array[i] = array[min];
13                 array[min] = temp;
14             }
15         }
16     }
17 }
```

And you can reference line **??** in the code like this.

2.2 Quick Sort Algorithm

Quicksort is a widely used sorting algorithm that operates by partitioning an array into two parts - elements smaller than a chosen pivot value and elements greater than the pivot value. The algorithm then recursively sorts the two partitions until the entire array is sorted. Quicksort is known for its efficiency, The key to Quicksort's efficiency is selecting an appropriate pivot value and partitioning the array efficiently. Quicksort is a popular choice for sorting large datasets and is often used as the default sorting algorithm in many programming languages.

```java
public class QuickSort {
    public static void sort(int[] array) {
        sort(array, 0, array.length - 1);
    }

    private static void sort(int[] array, int low, int high) {
        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;
            }
        }
    }

    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 divides an array into a set of buckets, sorts each bucket, and then combines the buckets to produce the sorted array. It has a time complexity of O(n+k) and is useful when the data is uniformly distributed over a known range. Bucket sort can be used as a preprocessing step for other sorting algorithms or as a standalone algorithm for small datasets.

```java
import java.util.*;

public class BucketSort {

    public static int[] sort(int[] array) {
        int numberOfBuckets = (int) Math.sqrt(array.length);
        ArrayList<ArrayList<Integer>> buckets = new ArrayList<>(
            numberOfBuckets);
        for (int i = 0; i < numberOfBuckets; i++) {
            buckets.add(new ArrayList<>());
        }
        int max = max(array);
        for (int i : array) {
            buckets.get(hash(i, max, numberOfBuckets)).add(i);
        }
        for (ArrayList<Integer> bucket : buckets) {
            Collections.sort(bucket);
        }
        int[] sortedArray = new int[array.length];
        int index = 0;
        for (ArrayList<Integer> bucket : buckets) {
            for (int integer : bucket) {
                sortedArray[index++] = integer;
            }
        }
        return sortedArray;
    }

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

    private static int max(int[] array) {
        int max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i] > max) {
                max = array[i];
            }
        }
        return max;
    }
```

```
}
```

2.4 Linear Search Algorithm

Linear search, also known as sequential search, is a basic search algorithm that checks each element of an array or list until the target element is found or the entire list has been searched. The algorithm starts at the first element of the list and checks each subsequent element until the target element is found or the end of the list is reached. Linear search has a worst-case time complexity of O(n), where n is the number of elements in the list. It is useful for small lists or when the target element is expected to be near the beginning of the list. However, for larger lists, more advanced search algorithms such as binary search may be more efficient.

```

public class LinearSearch {
    public static int linearSearch(int[] arr_for_temporary, int target) {
        for (int i = 0; i < arr_for_temporary.length; i++) {
            if (arr_for_temporary[i] == target) {
                return i;
            }
        }
        return -1; // Element not found
    }

}
```

2.5 Binary Search Algorithm

Binary search is a search algorithm that operates by dividing an ordered array or list into halves and repeatedly eliminating one half of the remaining elements until the target element is found or the search space is empty. The algorithm compares the target value to the middle element of the array or list and, depending on the result, discards the half that cannot contain the target element. Binary search has a time complexity of O(log n), where n is the number of elements in the list, making it much more efficient than linear search for larger lists. However, binary search only works on ordered lists, and its overhead of sorting the list or maintaining its order can make it less efficient for small lists.

```

public class BinarySearch {
    public static int binarySearch(int[] arr_for_temporary, int target) {
        int low = 0;
        int high = arr_for_temporary.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr_for_temporary[mid] == target) {
                return mid;
```

```
            } else if (arr_for_temporary[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1; // Element not found

}

}
```

3 Results, Analysis, Discussion

Your explanations, results, plots go in this section...

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	0	0	3	14	58	223	922	3645	13433
Quick sort	2	0	0	0	0	1	3	11	34	58
Bucket sort	2	1	0	1	2	5	5	9	17	34
	Sorted Input Data Timing Results in ms									
Selection sort	0	0	0	3	14	56	227	909	3667	13484
Quick sort	0	0	1	6	24	98	391	1566	6234	22991
Bucket sort	0	0	0	0	0	0	0	1	2	4
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0	0	1	4	16	66	267	1114	4882	22196
Quick sort	0	0	1	5	20	80	311	1212	4710	1656
Bucket sort	0	0	0	0	0	0	0	1	4	5

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)	1995	1146	2688	614	643	1377	2048	4696	8384	15218
Linear search (sorted data)	36	78	99	305	472	1007	2755	8574	12158	23367
Binary search (sorted data)	2326	50	41	47	48	395	72	84	124	157

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

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	$O(1)$

Results analysis, explanations...

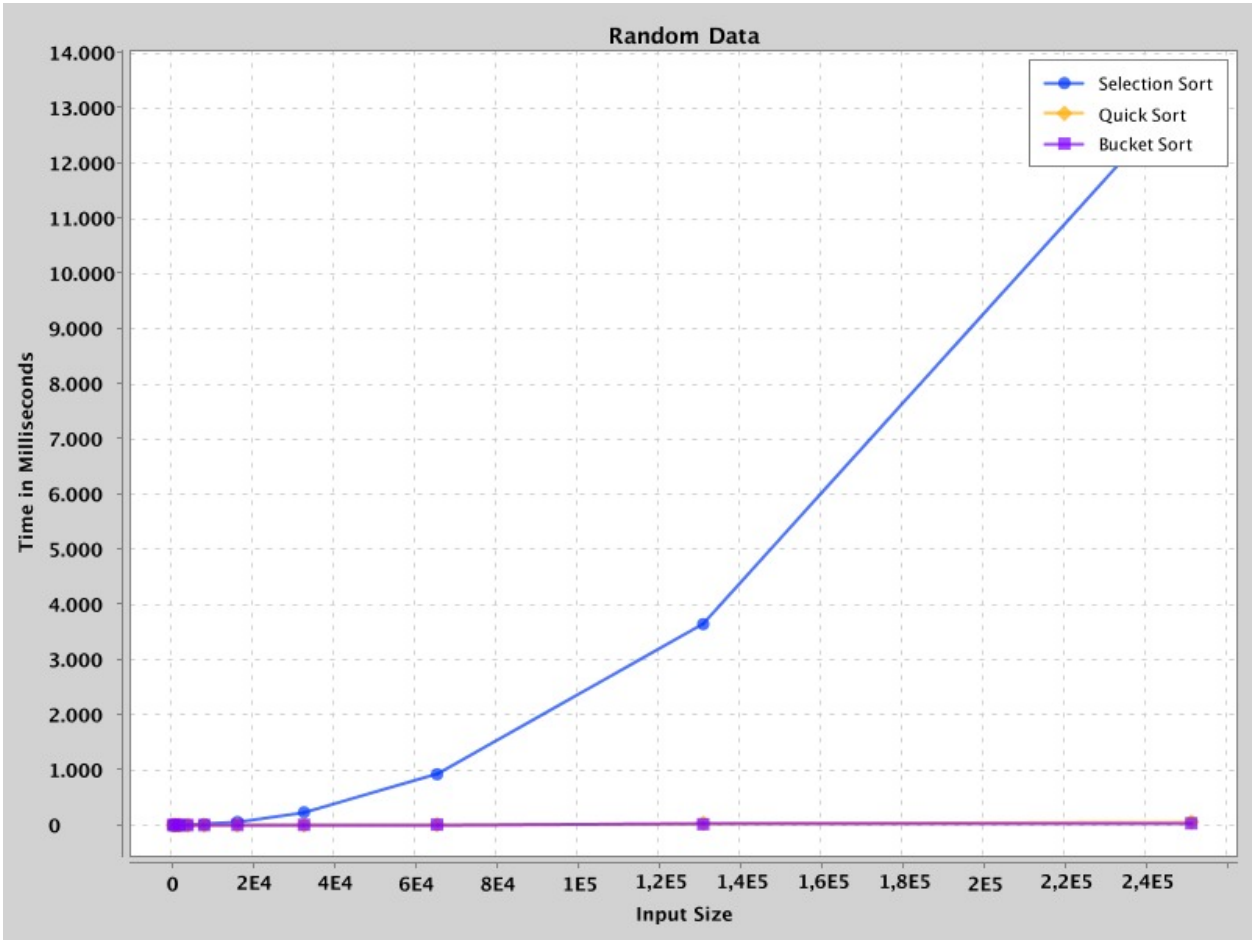

Figure 1: Sorting algorithms on random Data.

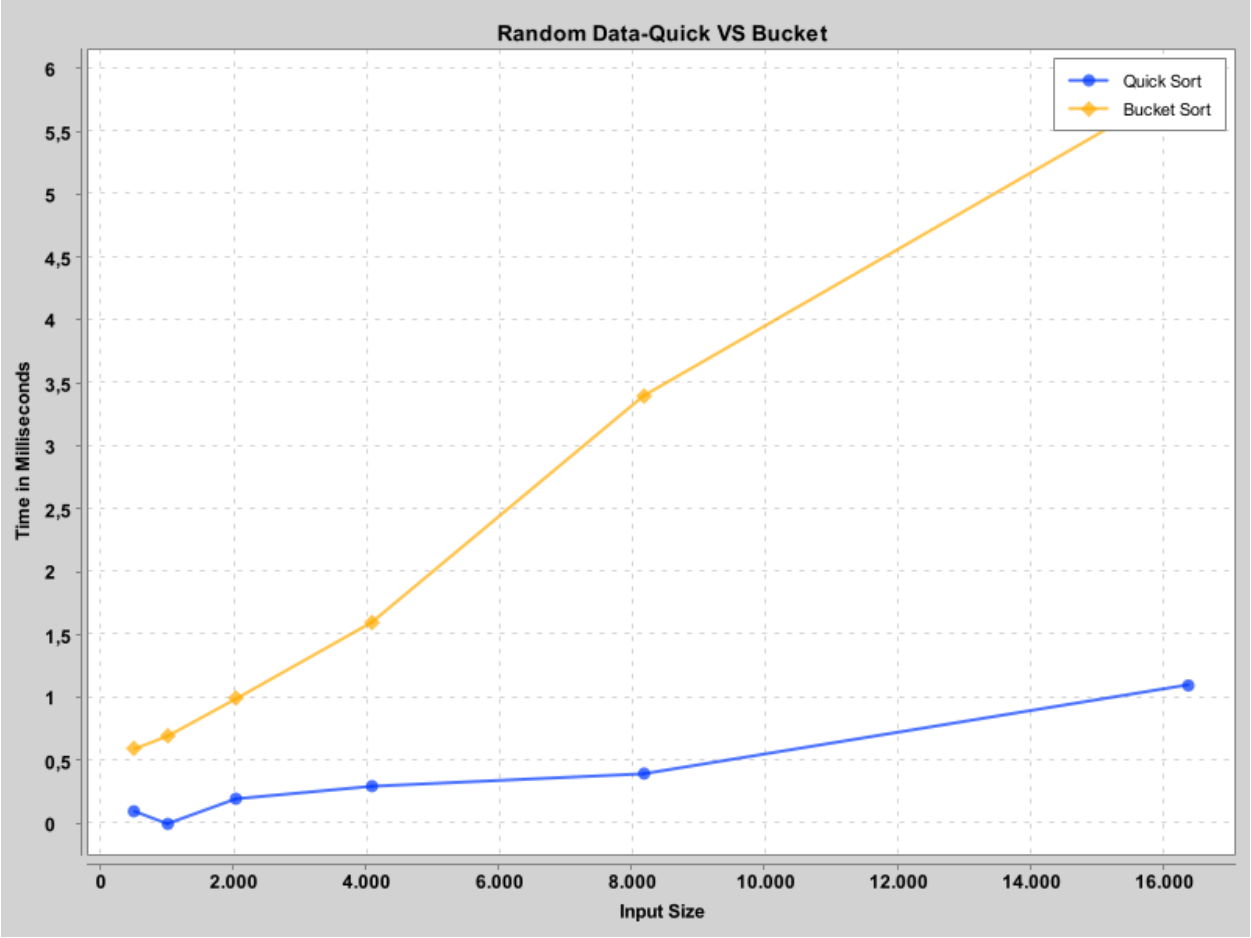

Figure 2: Sorting algorithms on random Data.(Quick Sort vs Bucket Sort

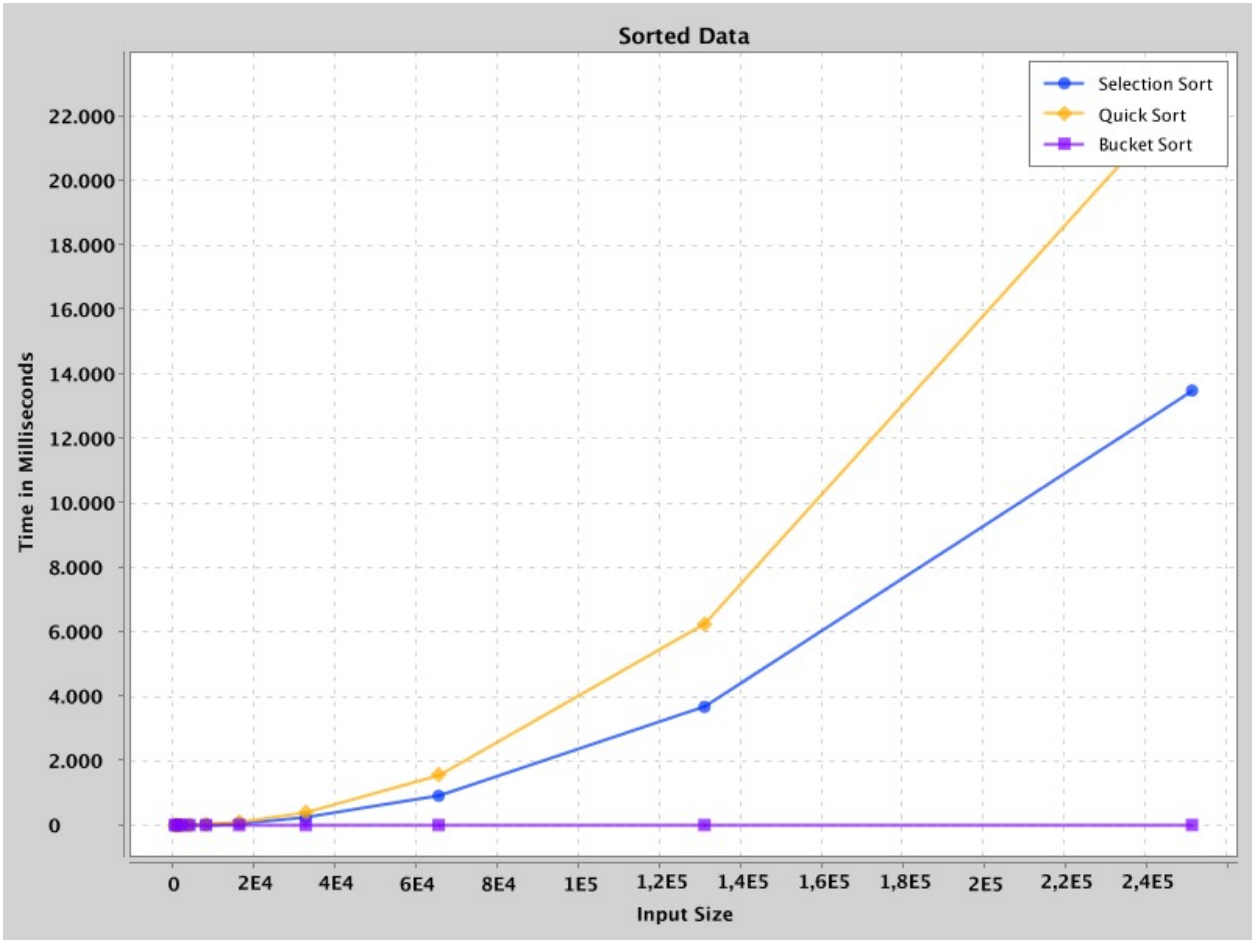

Figure 3: Sorting algorithms on sorted Data.

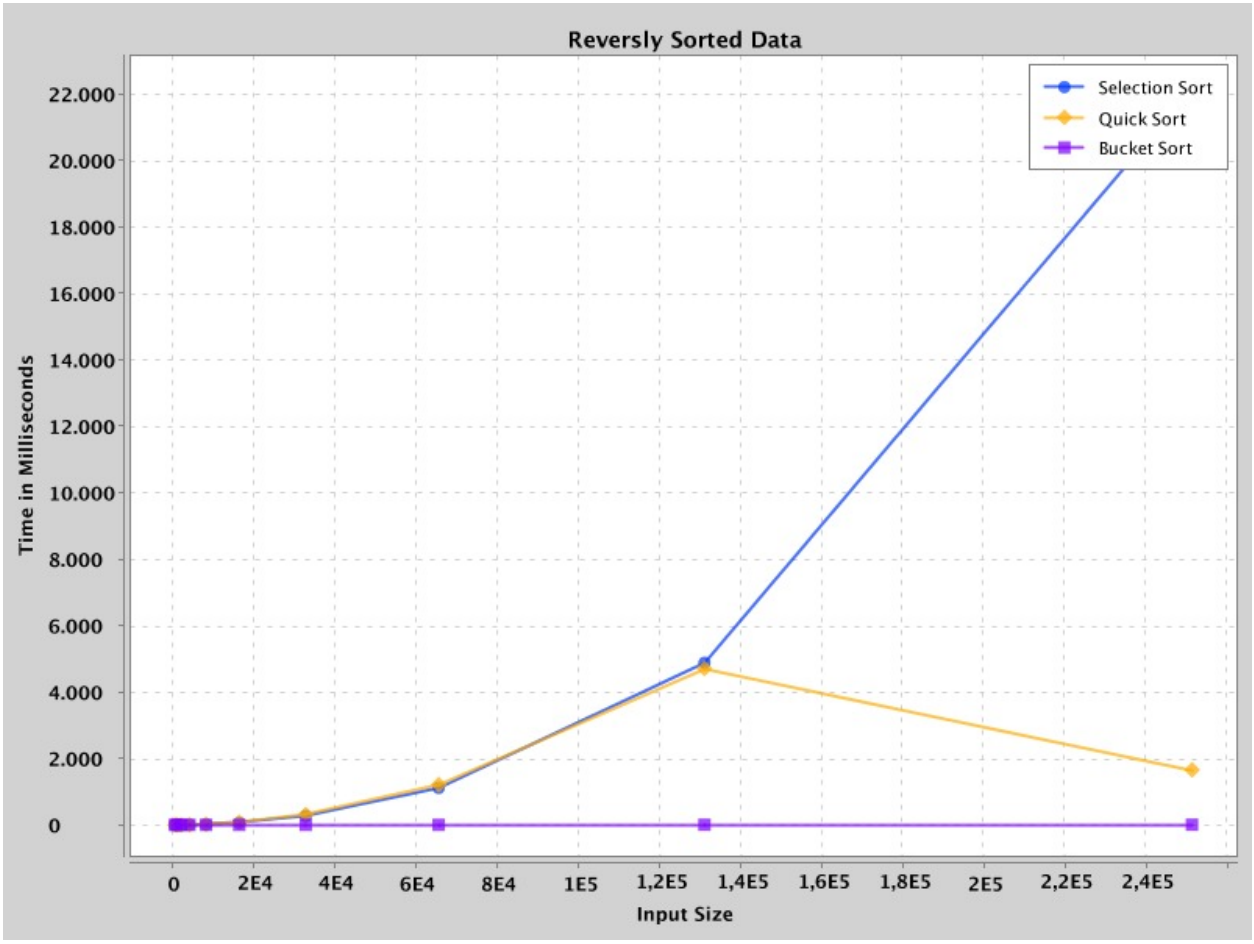

Figure 4: Sorting algorithms on reversely sorted Data.

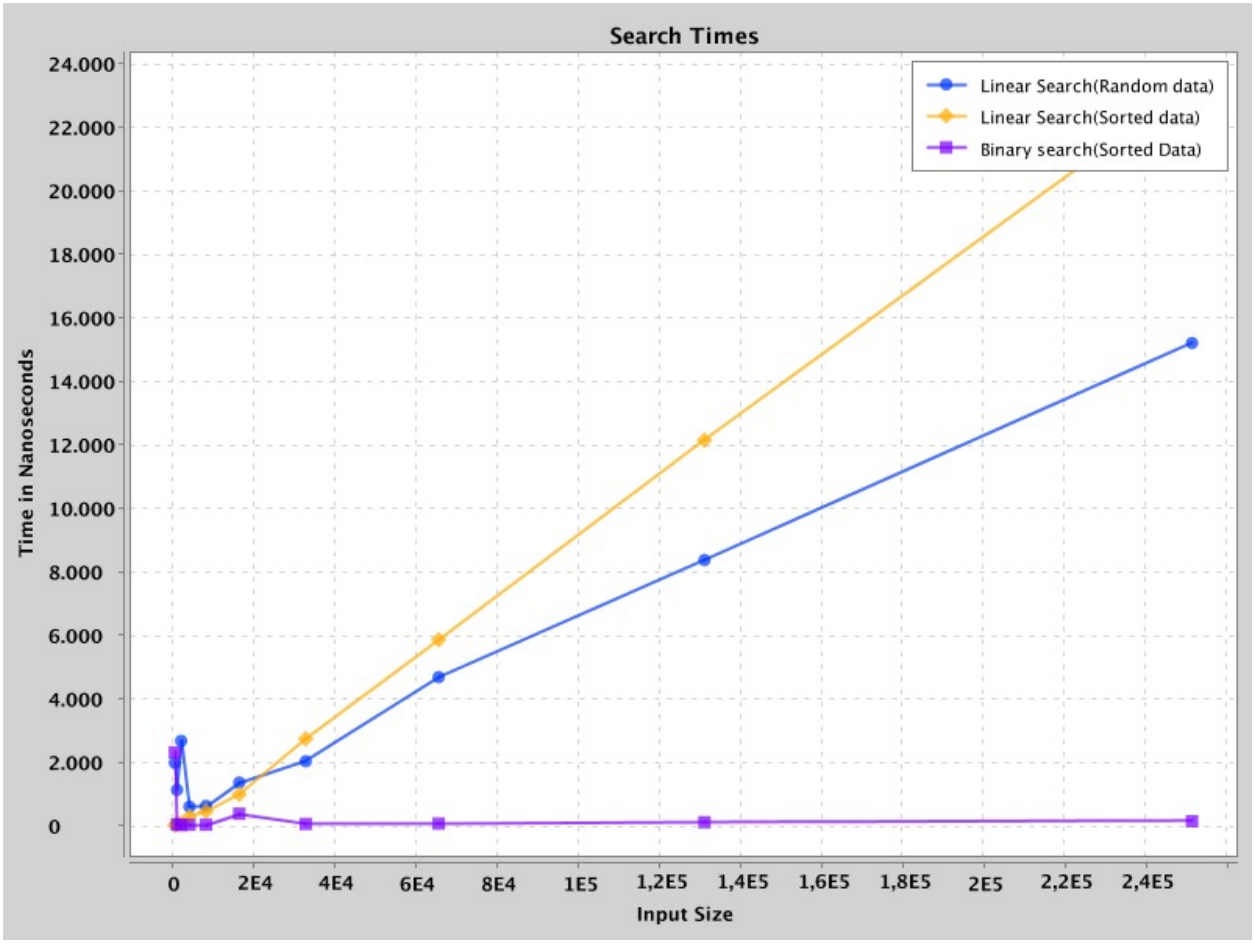

Figure 5: Binary on Sorted data, Linear on both Random and Sorted Data

In terms of sorting algorithms, bucket sort proves to be the most efficient across all scenarios, with a time complexity of O(n + k), where n represents the number of items and k denotes the number of buckets used. However, it requires additional space to store the buckets, which makes it less desirable than in-place sorting algorithms. Selection sort, on the other hand, consistently performs the poorest in all cases except when dealing with already sorted data due to its time complexity of O(n2). Quick sort performs well when dealing with randomly sorted data but performs poorly on already sorted data, worse even than selection sort. It shows average performance when handling reverse sorted data, with time complexities of O(log n) for best and average cases and O(n2) for the worst case. The algorithms generally perform as expected based on their theoretical time complexities, with the exception of quick sort's unexpectedly good performance on reverse sorted data, which may be attributed to the pivot selection method.

As for searching algorithms, binary search proves to be the most efficient, with a time complexity similar to O(log n), but it requires the array to be sorted. Linear search performs the worst when dealing with sorted arrays, with an exact time complexity of O(n). On the other hand, linear search exhibits better performance when handling randomly sorted data, although still not as good as binary search, with a similar time complexity of O(n). Both algorithms perform in accordance with their theoretical time complexities.

In conclusion, bucket sort is the most efficient sorting algorithm, whereas selection sort is the least efficient for randomly and reverse sorted data. Quick sort performs poorly on already sorted data. Binary search is the most efficient searching algorithm, but only if the array is sorted. Linear search, on the other hand, performs poorly when dealing with sorted data, and its performance is only marginally better on randomly sorted data. The algorithms generally perform in accordance with their theoretical time complexities, with the exception of quick sort's unexpectedly good performance on reverse sorted data, which may be attributed to the pivot selection method.

4 Notes

IN REVERSELY SORTED DATA When dealing with reverse sorted data, at the beginning, both selection sort and quick sort perform similarly in terms of efficiency. However, as the size of the data increases, quick sort exhibits better performance. This is because as the data size grows, the time complexity of selection sort increases at a faster rate , causing selection sort to become significantly less efficient.

References

- For Sorting Algorithms: https://www.geeksforgeeks.org/sorting-algorithms/
- For Searching Algorithms: https://www.geeksforgeeks.org/searching-algorithms/
- BBM202 and BBM204