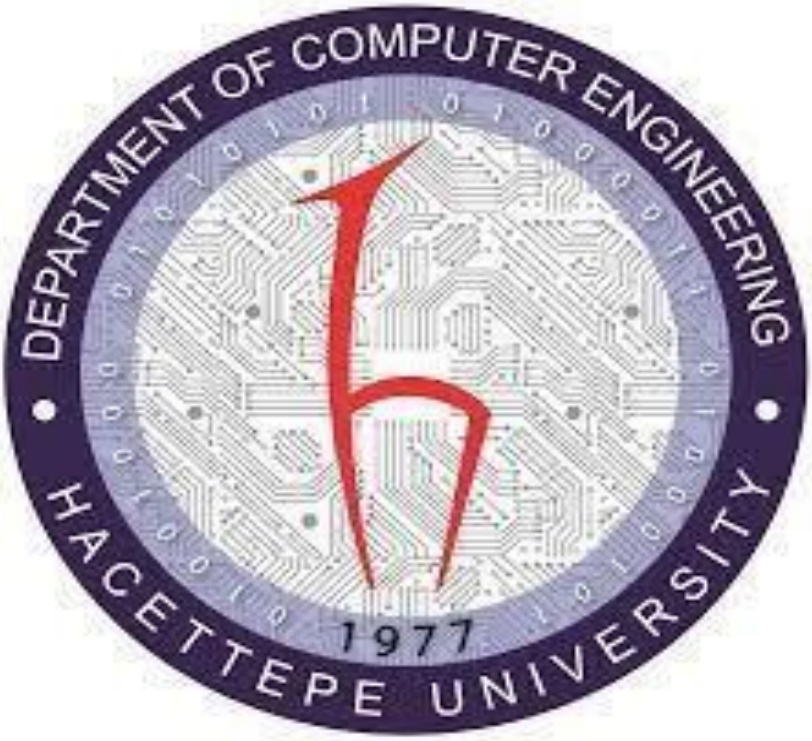
Hacettepe University

Department Of Computer Science

BBM204 Assignment 1 Report

Berke Abdullah Yıldız – 2210356100



PROBLEM STATEMENT

We aim to evaluate the performance of linear search and binary search algorithms on given random datasets with varying input sizes. Additionally, we plan to perform search operations on both randomly generated and sorted datasets. We will conduct each experiment 1000 times and compute the average running times of the algorithms in nanoseconds.

To articulate this problem statement comprehensively, we will include the sorting experiments as well:

1-Experiments on the Given Random Data:

We will generate random datasets with varying input sizes. For each dataset size, we will perform sorting algorithms such as insertion sort, merge sort, and counting sort. After sorting, we will conduct linear search, binary search, and other required search algorithms on the sorted datasets. Each search operation will be performed 1000 times, with a random target number selected for each iteration. The running times of the search algorithms will be recorded in nanoseconds, and the average running time will be computed.

2-Experiments on the Sorted Data: We will reuse the sorted datasets obtained from the previous experiments. Again, we will conduct linear search and binary search operations on these sorted datasets. Similar to the random data experiments, each search operation will be repeated 1000 times with randomly selected target numbers. The running times will be measured in nanoseconds, and the average running time for each algorithm will be calculated.

3-Experiments on the Reversely Sorted Data: We will reverse the sorted datasets obtained from the previous experiments to create reversely sorted data. Then, we will perform linear search and binary search operations on these reversely sorted datasets. Each search operation will be repeated 1000 times with randomly selected target numbers. The running times will be measured in nanoseconds, and the average running time for each algorithm will be calculated.

Search and Sorting Algorithms Java Codes

1 - Merge Sort Algorithm

```
public class MergeSort {
    public static void sort(Integer[] array) {
        mergeSort(array, 0, array.length - 1);
    }

    private static void mergeSort(Integer[] array, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSort(array, left, mid);
            mergeSort(array, mid + 1, right);

            merge(array, left, mid, right);
        }
    }

    private static void merge(Integer[] array, int left, int mid, int
right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        Integer[] L = new Integer[n1];
        Integer[] R = new Integer[n2];

        for (int i = 0; i < n1; ++i)
            L[i] = array[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = array[mid + 1 + j];

        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            } else {
                array[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }
}
```

2 - Insertion Sort Algorithm

```java
public class InsertionSort {
    public static void sort(Integer[] array) {
        int n = array.length;
        for (int i = 1; i < n; ++i) {
            int key = array[i];
            int j = i - 1;

            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }
}
```

3 - Counting Sort Algorithm

```java
public class CountingSort {

    public static void sort(Integer[] array, int max) {
        int[] count = new int[max + 1];

        for (int value : array) {
            count[value]++;
        }

        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }

        Integer[] sortedArray = new Integer[array.length];
        for (int i = array.length - 1; i >= 0; i--) {
            sortedArray[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }

        System.arraycopy(sortedArray, 0, array, 0, array.length);
    }
}
```

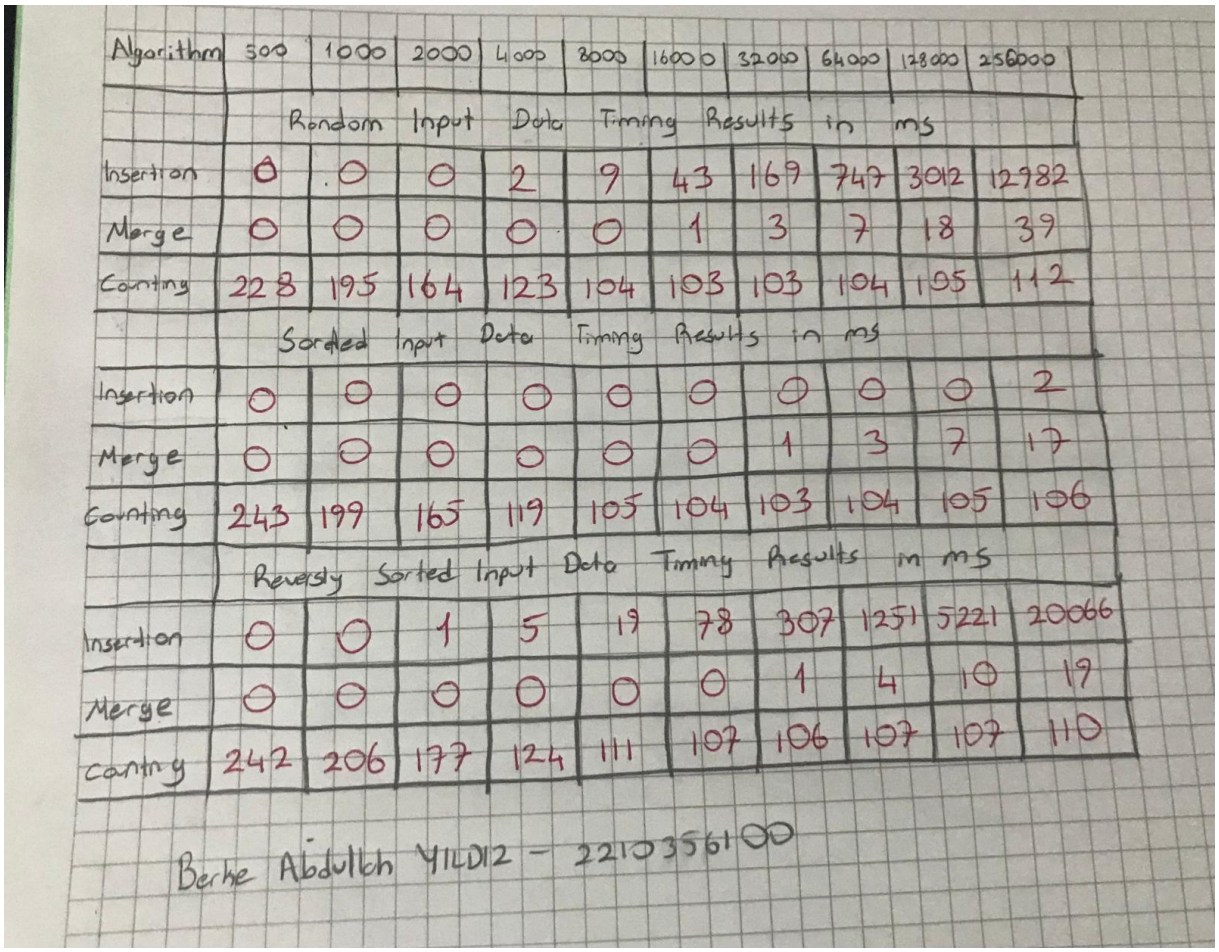
4 - Linear Search

```java
public class LinearSearch {

    public static int search(Integer[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

5 - Binary Search

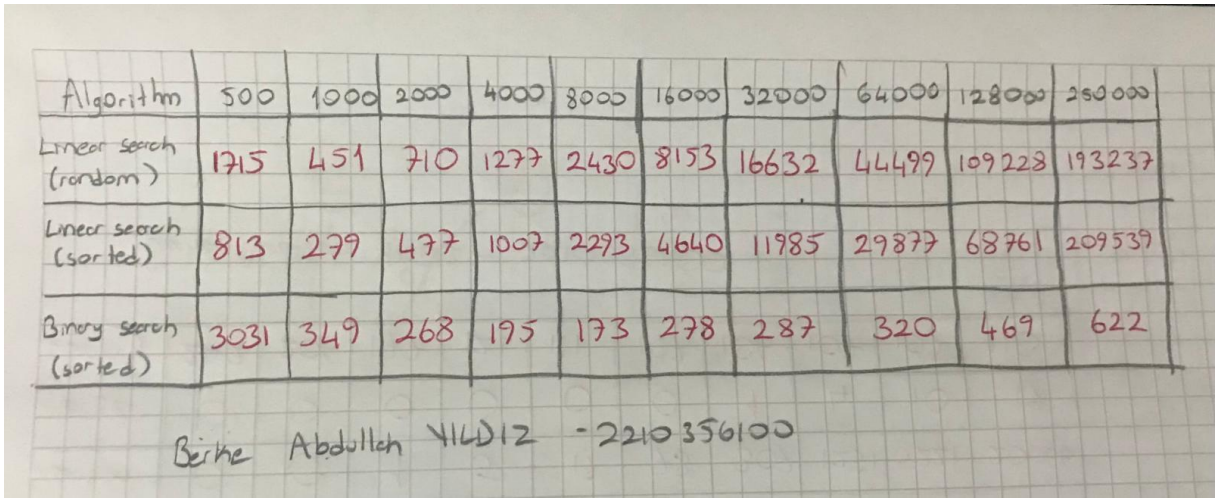
```java
public class BinarySearch {

    public static int search(Integer[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (array[mid] == target) {
                return mid;
            }
            else if (array[mid] < target) {
                left = mid + 1;
            }
            else {
                right = mid - 1;
            }
        }

        return -1;
    }
}
```

Experiment Results



Algorithm	500	1000	2000	4000	8000	16000	32000	64000	128000	256000
	Random Input Data Timing Results in ms									
Insertion	0	0	0	2	9	43	169	747	3012	12982
Merge	0	0	0	0	0	1	3	7	18	39
Counting	228	195	164	123	104	103	103	104	105	112
	Sorted Input Data Timing Results in ms									
Insertion	0	0	0	0	0	0	0	0	0	2
Merge	0	0	0	0	0	0	1	3	7	17
Counting	243	199	165	119	105	104	103	104	105	106
	Reversly Sorted Input Data Timing Results in ms									
Insertion	0	0	1	5	19	78	307	1251	5221	20066
Merge	0	0	0	0	0	0	1	4	10	19
Counting	242	206	177	124	111	107	106	107	107	110

Berke Abdullah YILDIZ - 2210356100



Algorithm	500	1000	2000	4000	8000	16000	32000	64000	128000	256000
Linear search (random)	1715	451	710	1277	2430	8153	16632	44499	109228	193237
Linear search (sorted)	813	279	477	1007	2293	4640	11785	29877	68761	209539
Binary search (sorted)	3031	349	268	195	173	278	287	320	469	622

Berke Abdullah YILDIZ - 2210356100

Table 1: Computational Complexity

Algorithm	Best Case	Average Case	Worst Case	Justification
Linear Search	O(1)	O(n/2)	O(n)	In the worst case, linear search traverses through all elements once.
Binary Search	O(1)	O(log n)	O(log n)	Binary search operates on sorted data, halving the search space with each iteration.
Insertion Sort	O(n)	O(n^2)	O(n^2)	In the worst case, insertion sort requires comparing and shifting elements for each element in the array.
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Merge sort divides the array into halves recursively and merges them back together, resulting in a time complexity of O(n log n).
Counting Sort	O(n + k)	O(n + k)	O(n + k)	Counting sort counts the frequency of each element and then reconstructs the array, making it a linear time complexity algorithm.

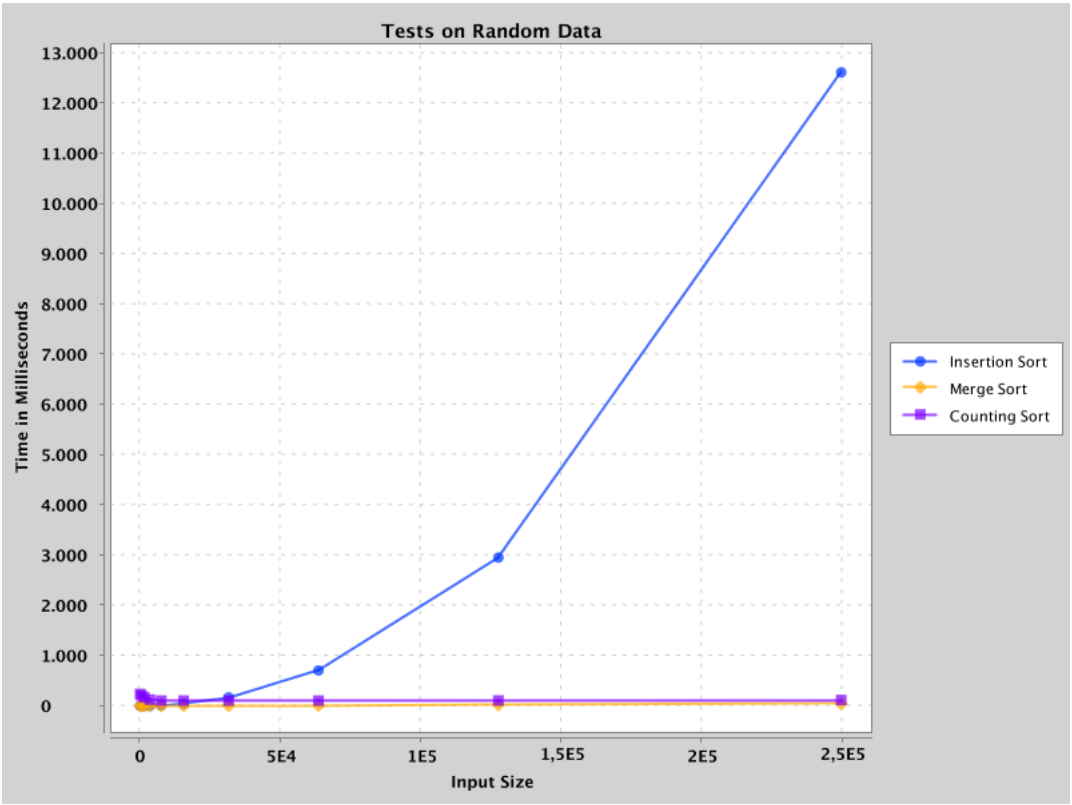
Table 2: Auxiliary Space Complexity

Algorithm	Worst Case	Justification
Linear Search	O(1)	Linear search does not require any additional space as it operates directly on the input array.
Binary Search	O(1)	Binary search operates recursively, but the space complexity is constant as it doesn't require additional space per iteration.
Insertion Sort	O(1)	Insertion sort operates in-place, requiring only a constant amount of additional space for variables.
Merge Sort	O(n)	Merge sort uses additional space proportional to the size of the input array for merging operations.
Counting Sort	O(n + k)	Counting sort requires additional space for the count array and the output array, which are both proportional to the size of the input array and the range of input values.

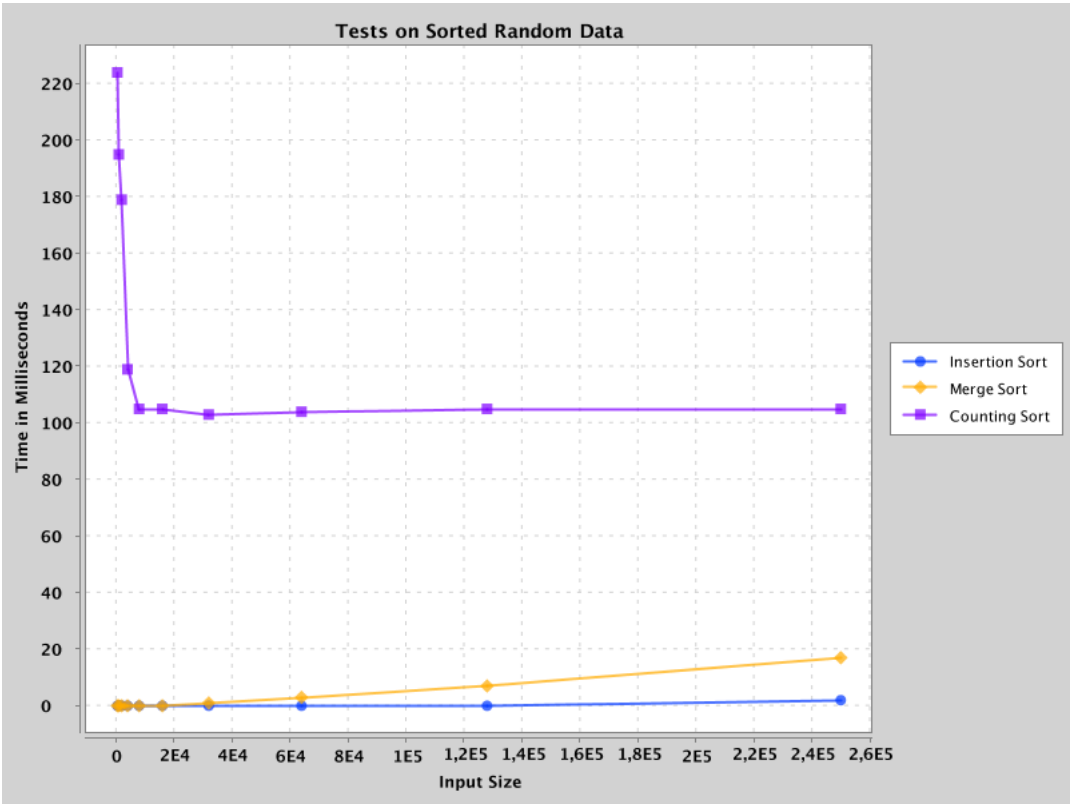
I created these tables via reference https://hackr.io/blog/big-o-notation-cheat-sheet

Plots of Sort and Search Algorithms

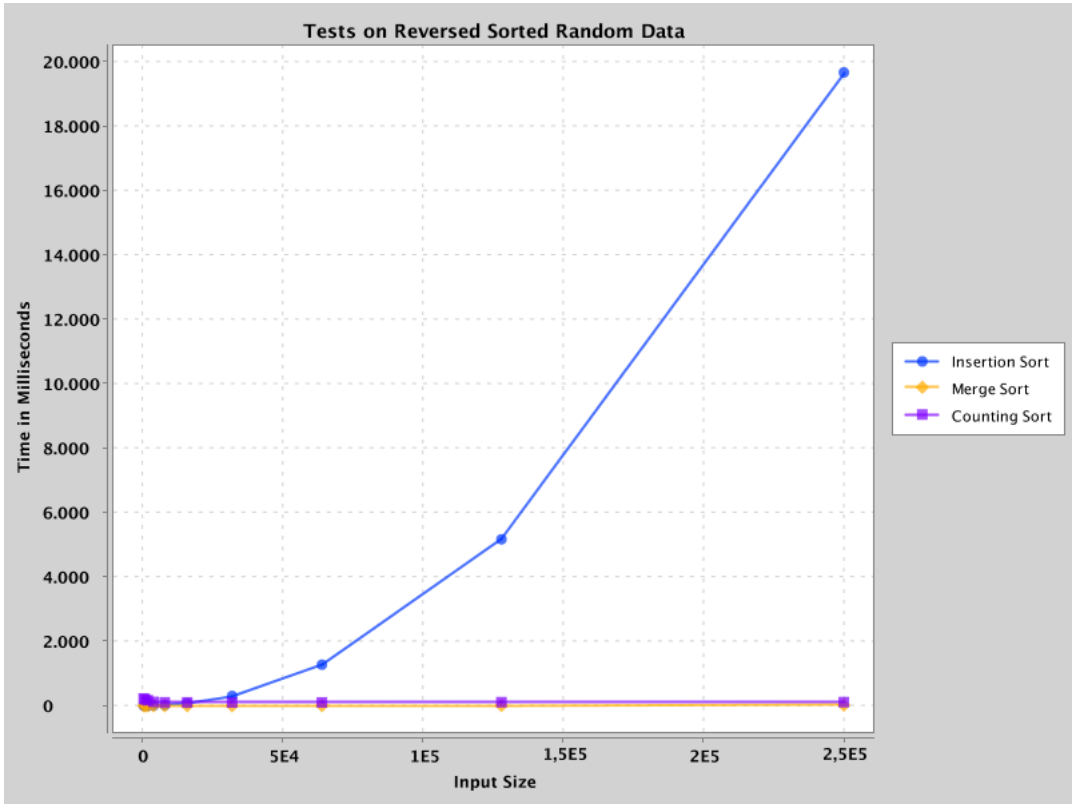
1 - Random Data Sorts Plot



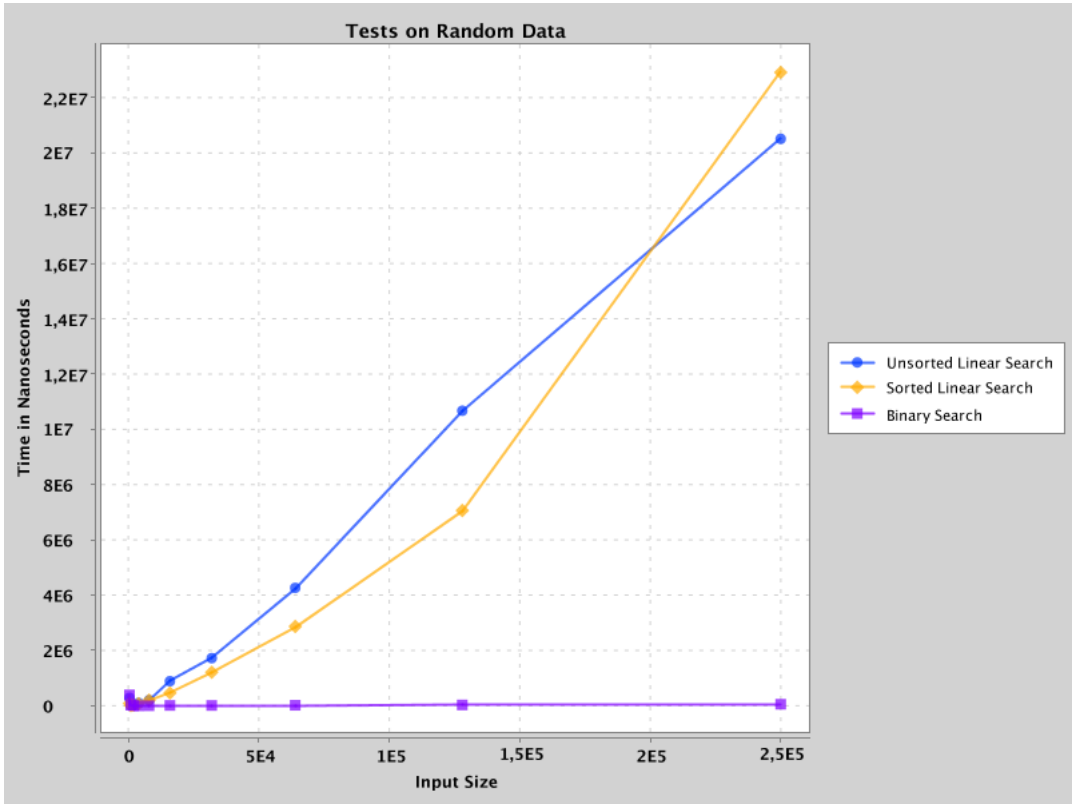
2 - Sorted Data Plot



3 - Reversly Sorted Data Plot



4 - Binary and Linear Search Plot



Result

- 1- What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted/searched?

Answer:

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n/2)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Counting Sort	O(n + k)	O(n + k)	O(n + k)

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

Answer:

Yes it matches. We can easily observe that as the number of data increases, the match rate increases. However, as the number of data decreases, the distance to theoretical data increases as the error that may occur increases.