Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Mehmet Oğuz Kocadere

Student Number:
b2210356021

1 Problem Definition

This assignment delves into the analysis of sorting and searching algorithms, aims to compare their strengths and weaknesses. The primary goal is to implement these algorithms, measure their execution times, and juxtapose their performance. The process involves executing the algorithms, recording the time taken for sorting and searching operations, and presenting the results graphically for clearer interpretation. Notably, intrigued statistics emerged during the implementation of the merge sort algorithm, which will be elaborated upon following the presentation of the data.

2 Solution Implementation

Your answers, explanations, code go into this section.

2.1 Counting Sort

Counting Sort Java code:

```java
public class CS extends SortAlgorithm {

    @Override
    public int[] sortMet(int[] a) {
        int N = a.length;
        int[] o = new int[N + 1];
        int m = a[0];
        for (int i = 1; i < N; i++) {
            if (a[i] > m) {
                m = a[i];
            }
        }
        int[] c = new int[m + 1];
        for (int i = 0; i < m; ++i) {
            c[i] = 0;
        }
        for (int i = 0; i < N; i++) {
            c[a[i]]++;
        }
        for (int i = 1; i <= m; i++) {
            c[i] += c[i - 1];
        }
        for (int i = N - 1; i >= 0; i--) {
            o[c[a[i]] - 1] = a[i];
            c[a[i]]--;
        }
        System.arraycopy(o, 0, a, 0, N);
        return a;
    }
}
```

This is a non-comparison based sorting algorithm. Due to count and output arrays in commented lines, the algorithm creates some auxiliary spaces, thus the space complexity is increased.

2.2 Insertion Sort

Insertion Sort Java code:

```
32 public class IS extends SortAlgorithm {
33
34     private static void sw(int[] a, int i, int j) {
35         int t = a[i];
36         a[i] = a[j];
37         a[j] = t;
38     }
39
40
41     @Override
42     public int[] sortMet(int[] a) {
43         for (int i = 1; i < a.length; i++) {
44             for (int j = i; j > 0; j--) {
45                 if (a[j] < (a[j - 1])) {
46                     sw(a, j, j - 1);
47                 } else {
48                     break;
49                 }
50             }
51         }
52         return a;
53     }
54 }
```

Insertion sort is a comparison-based sorting algorithm known for its simplicity and efficiency, especially for small arrays or nearly sorted arrays. It operates by iteratively traversing through the array, comparing and inserting each element into its correct sorted position within the sorted portion of the array. This process continues until all elements are in their correct sorted positions. Insertion sort has a worst-case time complexity of $O(n^2)$ and a best-case time complexity of $O(n)$, with a space complexity of $O(1)$, indicating that it does not require any extra space beyond the array itself. While not as efficient as some other sorting algorithms like quicksort or mergesort, insertion sort is advantageous due to its simplicity and low overhead. It is often used in hybrid sorting techniques and is particularly effective for small or nearly sorted arrays.

2.3 Merge Sort

Merge Sort Java code:

```
56 public class MS extends SortAlgorithm{
```

```
57
58
59      public static int[] m(int[] A, int[] B) {
60          int[] C = new int[A.length + B.length];
61          int i = 0, j = 0, k = 0;
62          while (i < A.length && j < B.length) {
63              if (A[i] > B[j]) {
64                  C[k++] = B[j++];
65              } else {
66                  C[k++] = A[i++];
67              }
68          }
69          while (i < A.length) {
70              C[k++] = A[i++];
71          }
72          while (j < B.length) {
73              C[k++] = B[j++];
74          }
75          return C;
76      }
77
78      @Override
79      public int[] sortMet(int[] A) {
80          int n = A.length;
81          if (n <= 1)
82              return A;
83          int m = n / 2;
84          int[] l = new int[m];
85          int[] r = new int[n - m];
86          for (int i = 0; i < m; i++)
87              l[i] = A[i];
88          for (int i = m; i < n; i++)
89              r[i - m] = A[i];
90          l = sortMet(l);
91          r = sortMet(r);
92          return m(l, r);
93      }
94  }
```

This is a comparison based sorting algorithm. The mS does not affect the space or time, it is created for only to direct the function to avoid code duplication. The mergesort divides the array down the middle into two arrays, and the merge function combines these two arrays. Merge Sort algorithm uses extra space when it combines two arrays. In merge sort algorithm, function makes at most M + N - 1 comparisons (M: length of first array, N: length of second array). If all elements in first array are smaller than elements of the second array, function makes M comparisons. If vice versa, function makes N comparisons. It means, sorted or reverse sorted data mean approxiamtely same way for merge sort algorithm and running will take less time.

2.4 Linear Search

Linear Search Java code:

```
public class Searching {
    public static int LS(int[] a, int x) {
        for (int i = 0; i < a.length; i++)
            if (a[i] == x) return i;
        return -1;
    }
}
```

Linear Search algorithm is running with $O(n)$ time on the both of sorted dataset and random dataset. However, the results on the random dataset is a little better from the sorted dataset in my graphics. I think it cause from the elements that I take randomly are distributed more nearly to beginning of my random data set. When I sort them, they moved away from the beginning of sorted dataset. It's auxiliary space complexity is $O(1)$.

2.5 Binary Search

Binary Search Java code:

```
public class Searching {
    private static int BS(int[] a, int x) {
        int l = 0, r = a.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;
            if (x == a[m]) return m;
            if (x > a[m]) l = m + 1;
            else r = m - 1;
        }
        return -1;
    }
}
```

Binary Search algorithm is running with $O(\log n)$ time complexity. It is the best searching algorithm with a big difference in the graph. It can be work with just sorted data. It's auxiliary space complexity is $O(1)$.

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									
Insertion sort	0.0	0.0	0.0	1.0	5.0	21.0	85.0	365.0	1450.0	5482.0
Merge sort	0.0	0.0	0.0	0.0	1.0	1.0	3.0	13.0	19.0	31.0
Counting sort	187.0	157.0	158.0	158.0	158.0	158.0	160.0	157.0	159.0	166.0
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Merge sort	0.0	0.0	1.0	0.0	1.0	2.0	3.0	8.0	23.0	37.0
Counting sort	163.0	158.0	157.0	155.0	167.0	158.0	159.0	157.0	159.0	157.0
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	2.0	9.0	38.0	175.0	743.0	2667.0	10019.0
Merge sort	0.0	0.0	0.0	0.0	1.0	2.0	5.0	9.0	14.0	37.0
Counting sort	208.0	165.0	160.0	161.0	159.0	157.0	155.0	159.0	160.0	150.0

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)	1562.0	587.0	327.0	565.0	1057.0	1929.0	3469.0	6341.0	9960.0	13371.0
Linear search (sorted data)	717.0	147.0	229.0	412.0	747.0	1372.0	2714.0	5542.0	10842.0	20553.0
Binary search (sorted data)	258.0	140.0	152.0	113.0	112.0	116.0	125.0	125.0	140.0	163.0

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**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n+1)$	$\Theta(n+k)$	$O(n+k)$
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**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

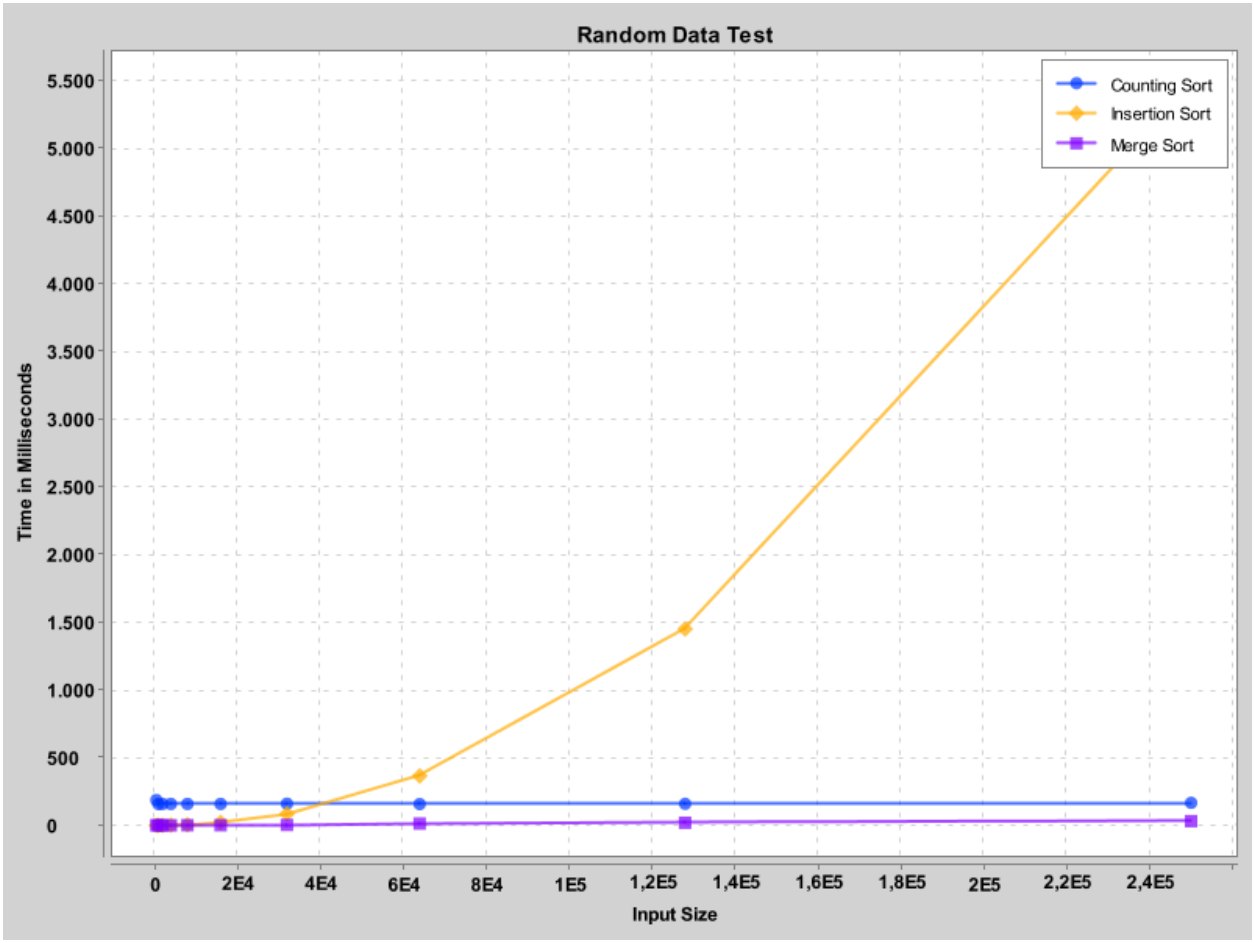

Figure 1: Plot of the sorting algorithms functions on the random dataset. The merge sort is working better than counting sort and insertion sort. Because the k factor of counting sort is making greater the complexity of counting sort from $O(nlogn)$. For insertion sort, it is working worstly for this case. Because it is working with nearly $O(n^2)$ time. The merge sort is working with $O(nlogn)$ for all dataset cases.

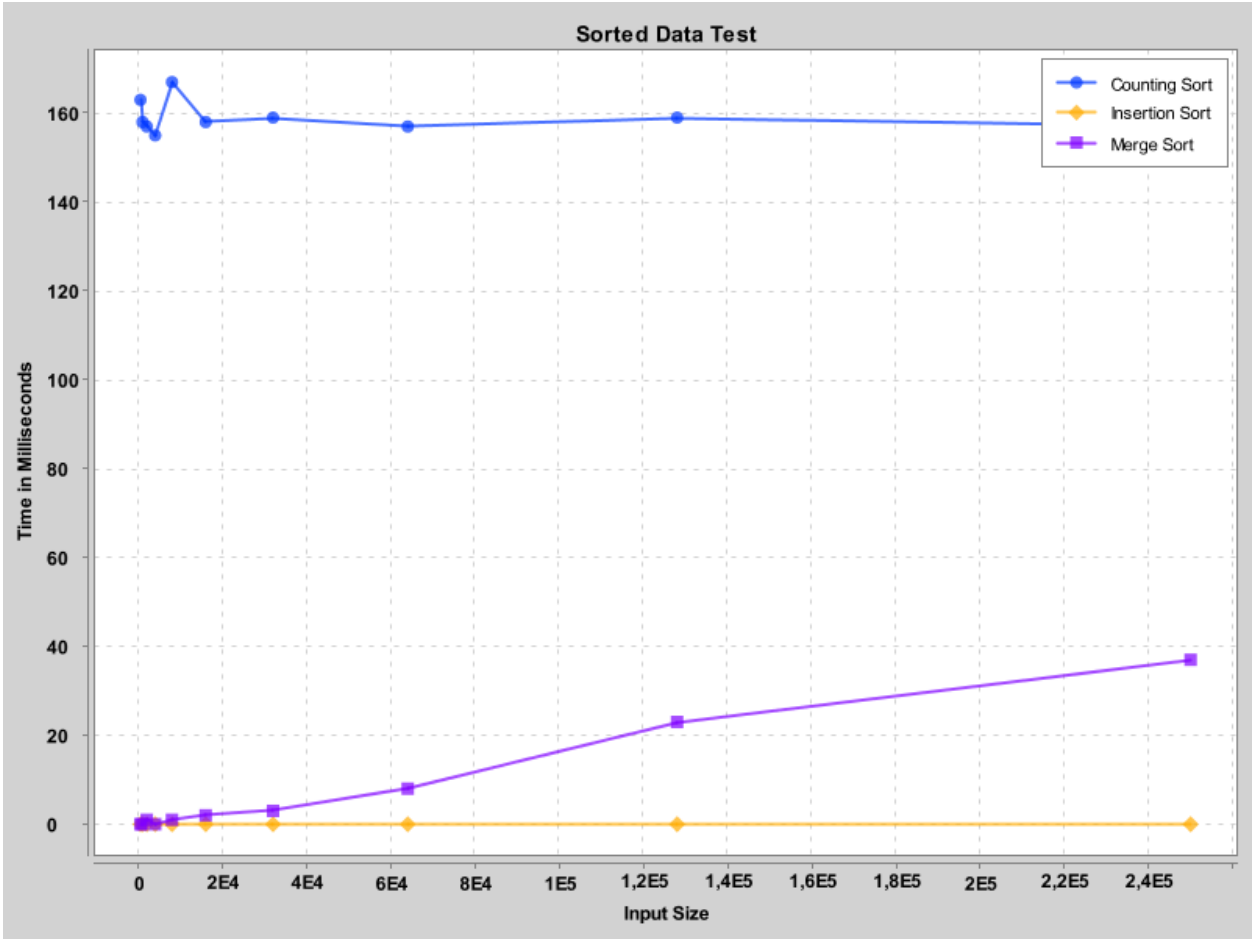

Figure 2: Plot of the sorting algorithm functions on the sorted dataset. The insertion sort is working best for this dataset case. Because for already sorted dataset insertion sort is working with $O(n)$ time complexity. After that, the merge sort is better than counting sort and a little worse from insertion sort with $O(nlogn)$ time complexity. The worst performance is belonging to counting sort. Because of that is k factor cost for the time complexity of $O(n+k)$.

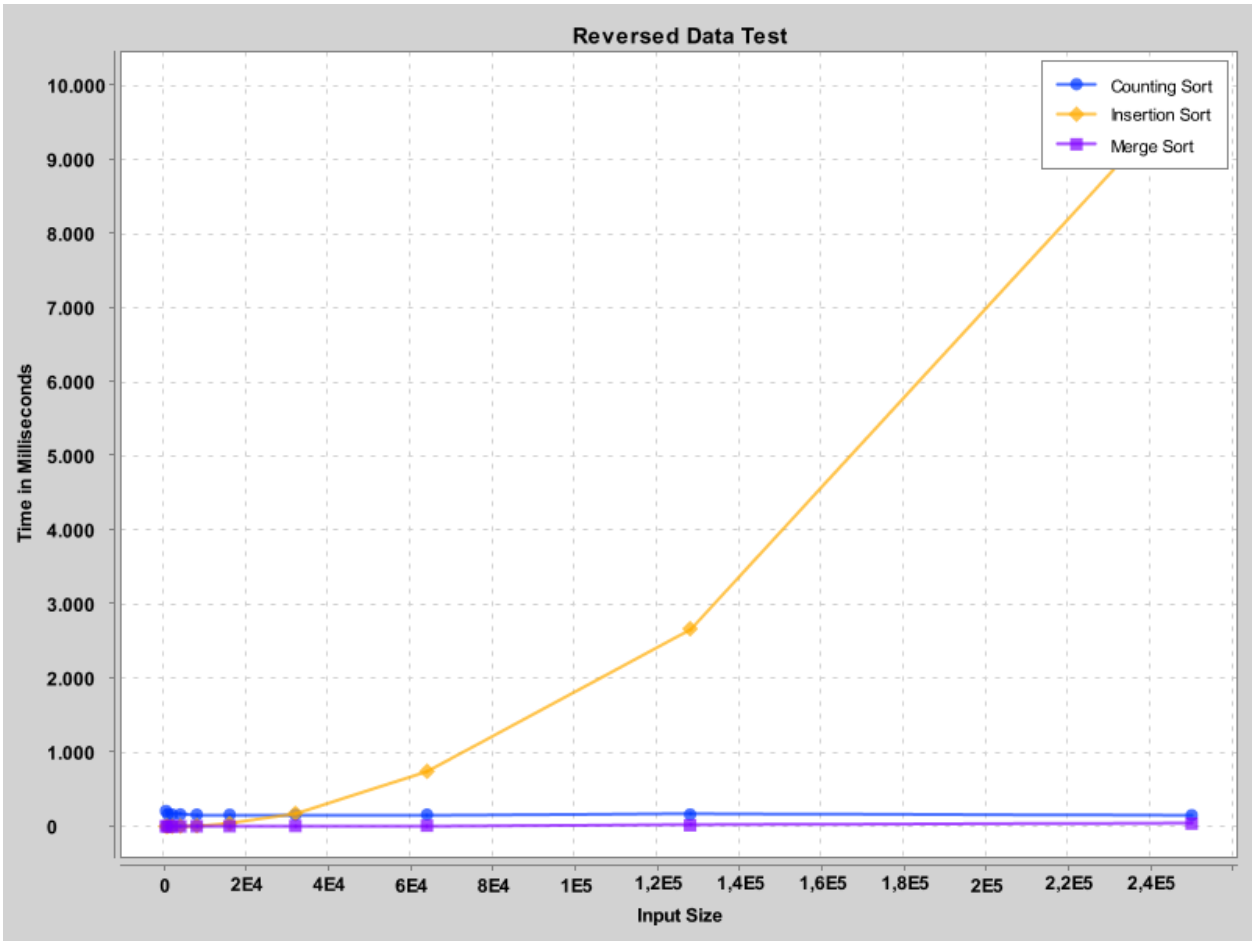

Figure 3: Plot of the sorting algorithm functions on the reversed dataset. The merge sort is working best for this dataset case. Because for reversed dataset insertion sort is working with $O(n^2)$ time complexity with worst case and counting sort is working with a little extra cost from merge sort. It cause from the k factor cost.

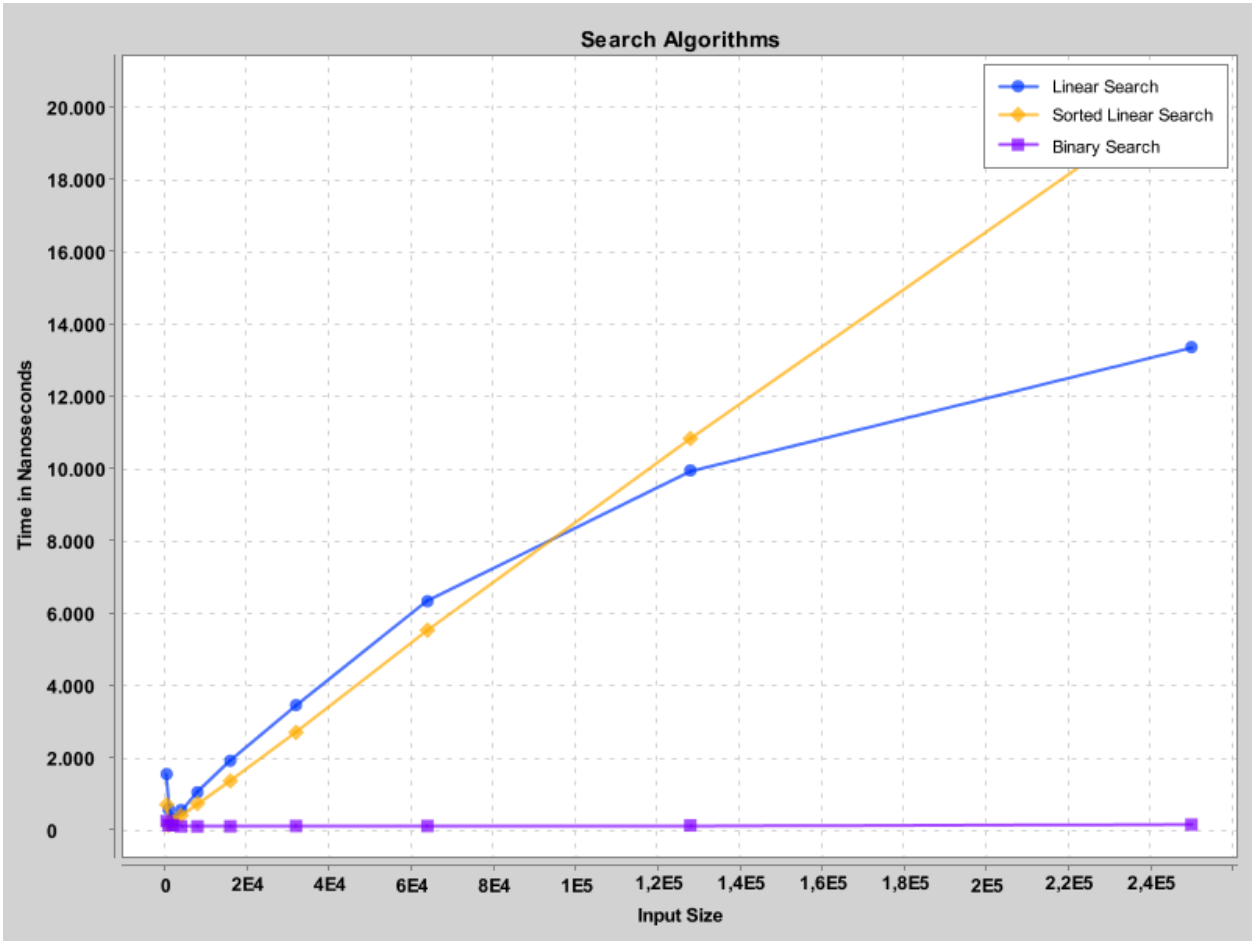

Figure 4: Plot of the search algorithms functions. The binary search has the best performance for this test. Becasue it is working with $O(logn)$ time complexity in the worst case of our tests. After that, the linear search with the random dataset is a little bit better than linear search with sorted dataset case. This situation is just depends on the selected random element from the array. I selected a random element for each iteration of my test cases and the distribution of them is near to beginning of the random dataset in my tests.

Results analysis, explanations...

To conclusion we can say that if we have data the length of around 2000 or less, we should use insertion sort algorithm to avoid spending time and using more space. Otherwise, if the length of data is greater than 2000, we should definitely use merge sort algorithm. So, what if we have billions of data? What algorithm is more effecient? Probably due to running in linear time, counting sort and pigeonhole sort will sort in less time.

3 Notes

Merge Sort algorithm worked faster than that I expected. However, it is not a problem and ratio of working times are suitable according to expectations and each other.

Also, I need to say this: In my searching algorithm tests I selected different random elements for all iterations of my searching algorithm test with the aim of providing a realistic chart. Firstly, I tried with the just one random element however it made my chart very bad distributed. I can show with a better way the behaviours of searching algorithms with this application.

References

- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.programiz.com/dsa/counting-sort