Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Ayşe Yaren Topgün

Student Number:
b2220356141

1 Problem Definition

In this assignment, it is intended that the performance of different sorting and searching algorithms under various conditions be explored. These algorithms will be analyzed based on the amount of time they consume and the memory they use when sorting or searching through data. The aim is to have the practical performance of these algorithms understood and demonstrated, with a comparison to their theoretical efficiency. The algorithms will be implemented in Java, tested on various sets of data, and their performance measured to determine if the actual results align with theoretical expectations.

2 Solution Implementation

2.1 Insertion Sort

Algorithm Description: Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.[1]

Code Description: Use a for loop to iterate from the second element of the array to the last element. For each element, store it in a temporary variable (key) and compare it with the elements to its left. If an element to the left is greater than the key, shift that element to the right. Once the correct position is found, insert the key value.

```
1
2  public class InsertionSort {
3
4      public static void insertionSort(int[] arr) {
5          for (int  j= 1; j < arr.length; j++) {
6              int key = arr[j];
7              int i = j - 1;
8
9              while (i >= 0 && arr[i] > key) {
10                 arr[i + 1] = arr[i];
11                 i = i - 1;
12             }
13             arr[i + 1] = key;
14         }
15     }
16 }
```

2.2 Merge Sort

Algorithm Description: Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.[2]

Code Description: The array is repeatedly halved until segments contain one or no elements. Sorted segments are then merged by a function that sequentially compares and combines them into a sorted array. This ensures each part is sorted before the final merge.

```
18
19  public class MergeSort {
20      public static void mergeSort(int[] arr){
21          if(arr.length <= 1){
22              return;
23          }
24          int middle = arr.length / 2;
25          int[] left = new int[middle];
26          int[] right = new int[arr.length - middle];
27
28          for(int i = 0; i < middle; i++){
29              left[i] = arr[i];
30          }
31          for(int i = middle; i < arr.length; i++){
32              right[i - middle] = arr[i];
33          }
34          mergeSort(left);
35          mergeSort(right);
36          merge(arr, left, right);
37      }
38
39      protected static void merge(int[] arr, int[] left, int[] right){
40          int i = 0; int j = 0; int k = 0;
41          while(i < left.length && j < right.length){
42              if(left[i] <= right[j]){
43                  arr[k++] = left[i++];
44              }
45              else {
46                  arr[k++] = right[j++];
47              }
48          }
49          while(i < left.length){
50              arr[k++] = left[i++];
51          }
52          while(j < right.length){
53              arr[k++] = right[j++];
54          }
55      }
56  }
```

2.3 Counting Sort

Algorithm Description: Counting Sort is a non-comparison-based sorting algorithm that works well when there is limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.[3]

Code Description: First, find the maximum element in the input array to determine the size of the counting array. For each element in the input array, increment the corresponding index in the counting array. Finally, use the counting array to construct the sorted array.

```
58
59
60  public class CountingSort {
61
62      public static  int[]  countingSort(int[] A){
63
64          int k = 0;
65          for (int value : A) {
66              if (value > k) {
67                  k = value;
68              }
69          }
70          int[] count = new int[k + 1];
71          int[] output = new int[A.length];
72          int size = A.length;
73
74          for(int i = 0; i < k; i++){
75              count[i] = 0;
76          }
77          for (int j : A) {
78              count[j] = count[j] + 1;
79          }
80          for(int i=1; i<k+1; i++){
81              count[i] = count[i] + count[i-1];
82          }
83          for(int i=size-1; i>=0; i--){
84              int j = A[i];
85              count[j] = count[j] - 1;
86              output[count[j]] = A[i];
87
88          }
89          return output;
90      }
91  }
```

2.4 Linear Search

Algorithm Description: Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.[4]

Code Description: Use a for loop to iterate from the start to the end of the array, checking each element. If the current element matches the target value, return the index of that element. If the end of the array is reached without finding the target, the search has failed.

```
93
94  public class LinearSearch {
95
96      public static int linearSearch(int[] A, int x){
97          int size = A.length;
98          for(int i=0; i< size; i++){
99              if(A[i] == x){
100                 return i;
101             }
102         }
103         return -1;
104     }
105 }
```

2.5 Binary Search

Algorithm Description: Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.[5]

Code Description: Initially, set the lower and upper bounds of the search range. Since the array is sorted, check the middle element of the range and compare it with the target value. If the middle element is less than the target, continue the search in the right half of the array. If greater, continue in the left half. Repeat this process until the target value is found or the search range is exhausted.

```
107
108 public class BinarySearch {
109
110     public static int binarySearch(int[] A , int x){
111         int low = 0;
112         int high = A.length - 1;
113
114         while(high-low>1){
115             int mid = (high+low)/2;
116             if(A[mid] < x){
117                 low = mid+1;
118             }
119             else{
120                 high = mid;
121             }
122         }
123
124         if(A[low] == x){
125             return low;
126         }
127         else if(A[high] == x){
128             return high;
129         }
130         else{
131             return -1;
132         }
133     }
134 }
```

3 Results, Analysis, Discussion

3.1 Results

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	1	1	4	21	52	214	1314
Merge sort	0	0	1	2	3	7	10	11	24	38
Counting sort	313	244	247	241	246	244	247	247	252	255
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	2
Merge sort	0	0	0	1	1	2	4	8	16	33
Counting sort	261	246	267	250	253	243	251	247	255	256
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	4	15	55	210	802	3193
Merge sort	0	0	0	1	1	3	8	15	24	33
Counting sort	257	251	281	252	253	245	250	249	250	260

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)	47733	55029	53312	11510	26834	44928	88858	170591	336085	590035
Linear search (sorted data)	3422	4303	9085	17267	34143	61579	123084	249191	527227	994113
Binary search (sorted data)	18873	4512	4201	3554	2071	2371	3205	3044	3909	2743

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

3.2 Analysis

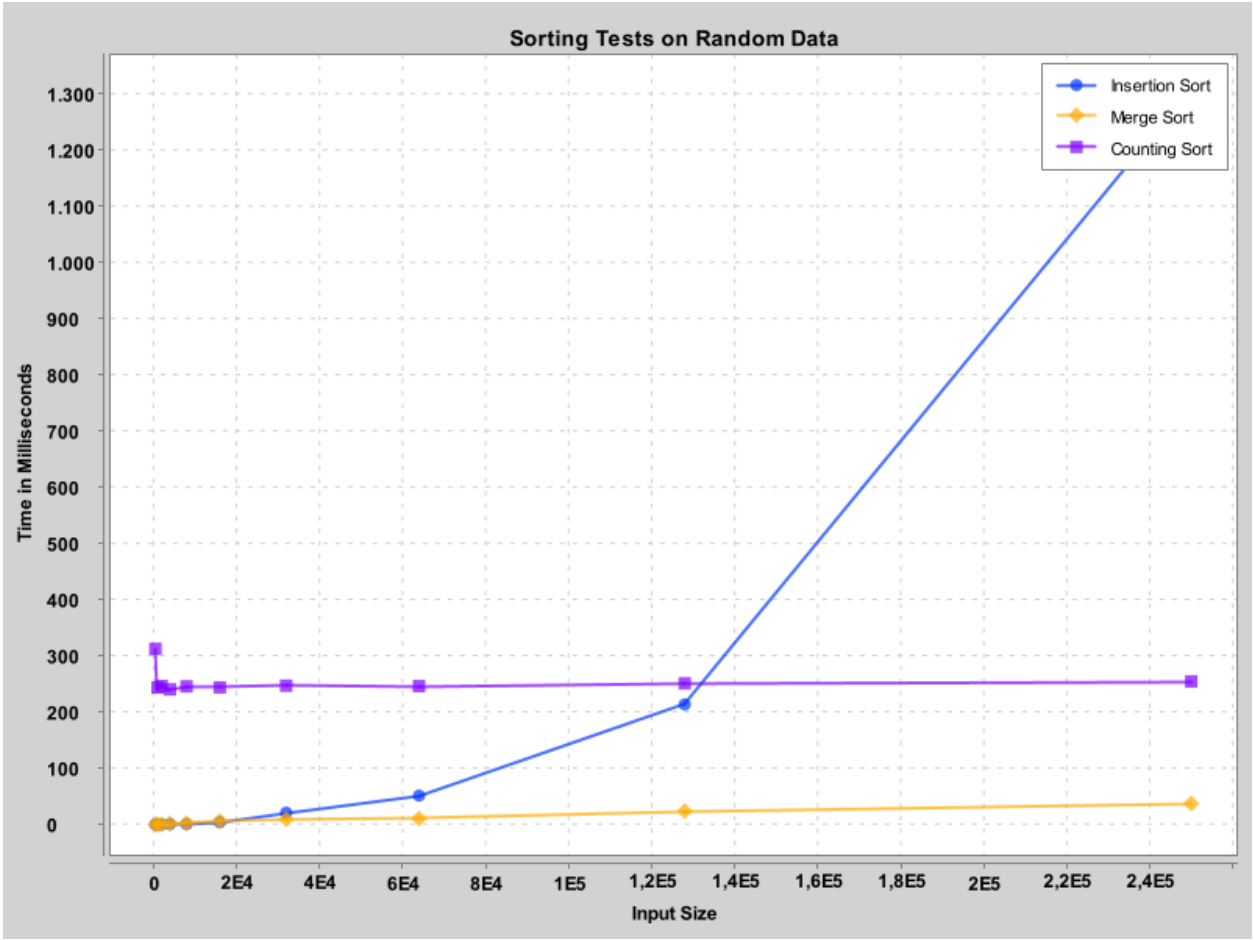

Figure 1: Sorting Tests on Random Data

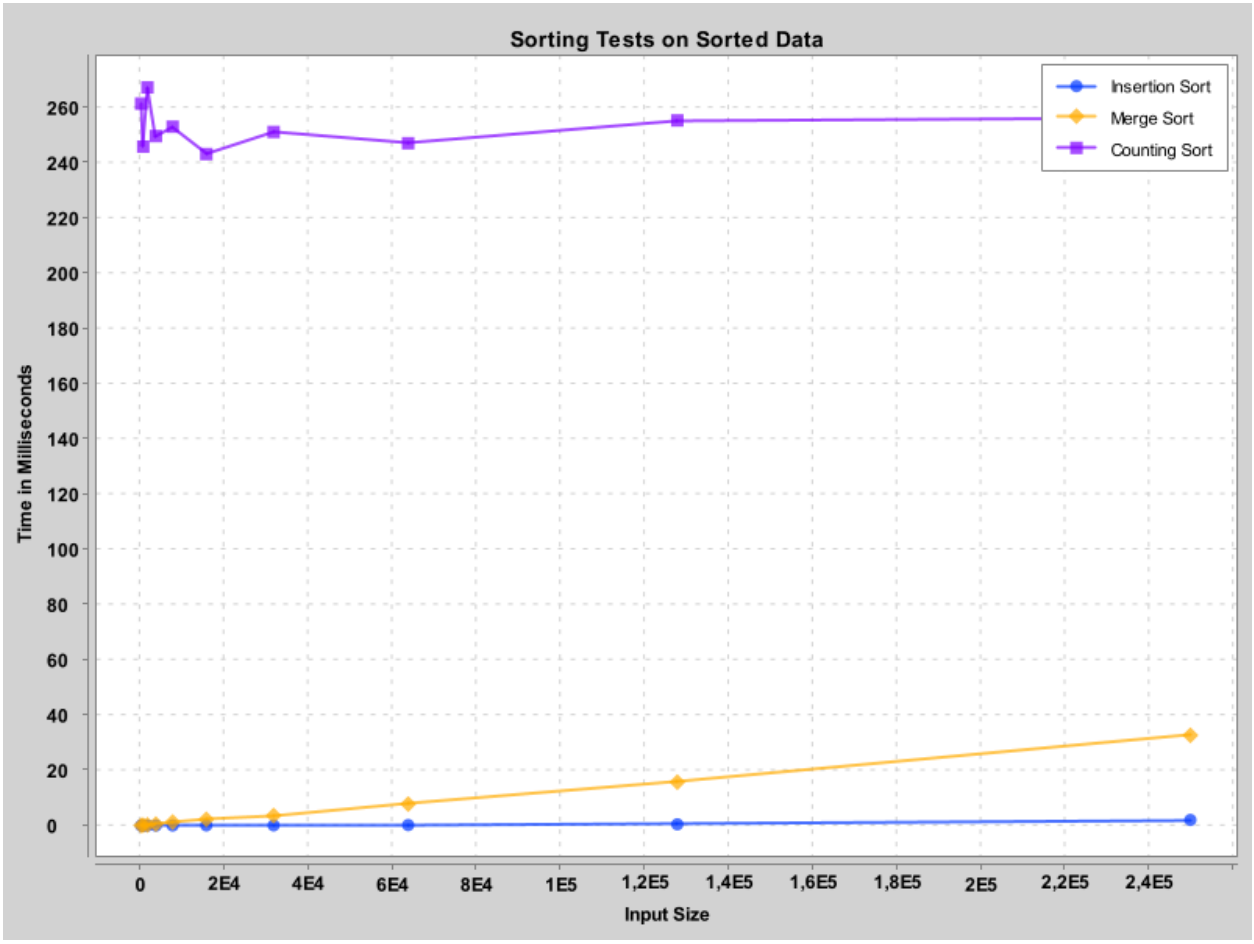

Figure 2: Sorting Tests on Sorted Data

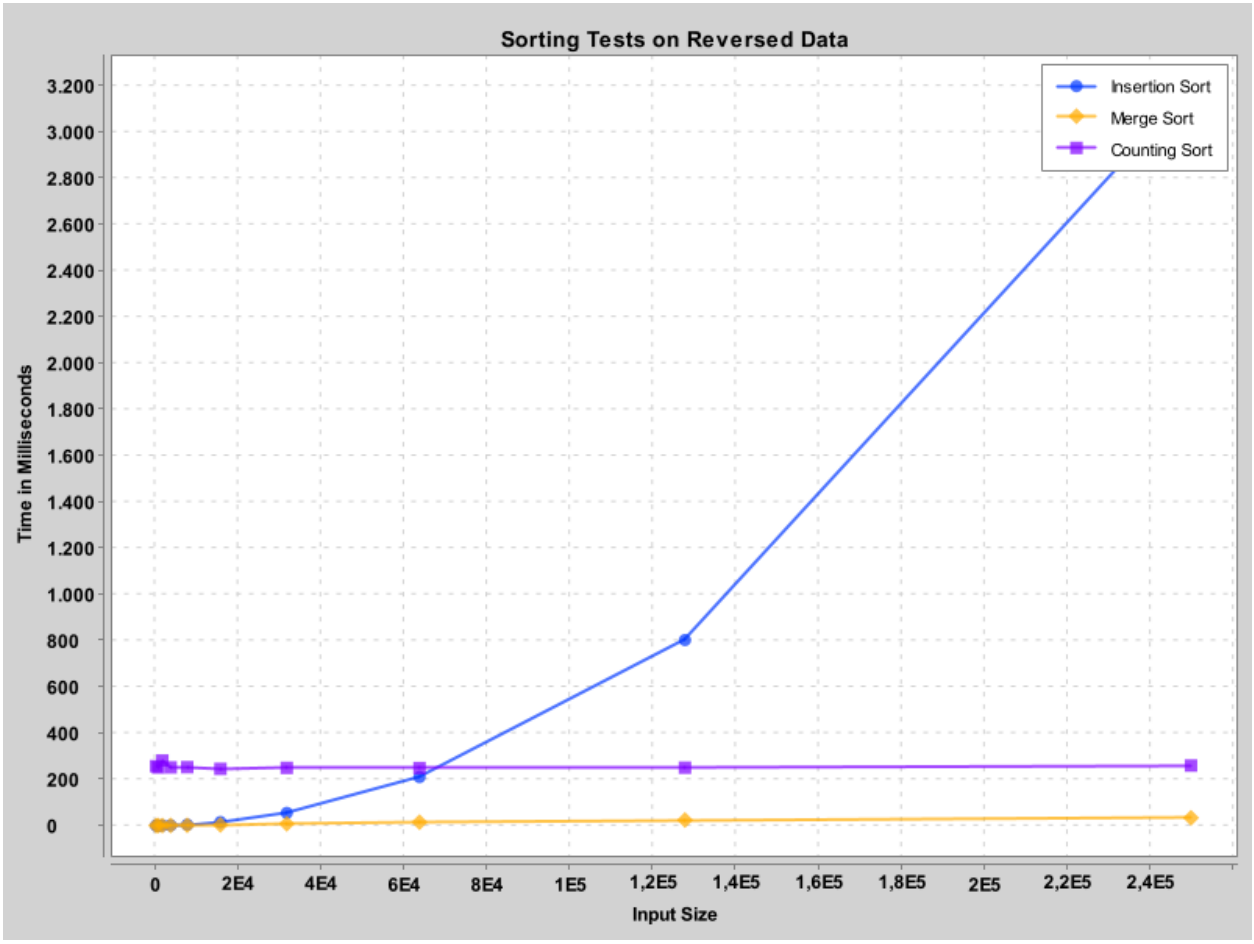

Figure 3: Sorting Tests on Reversed Data

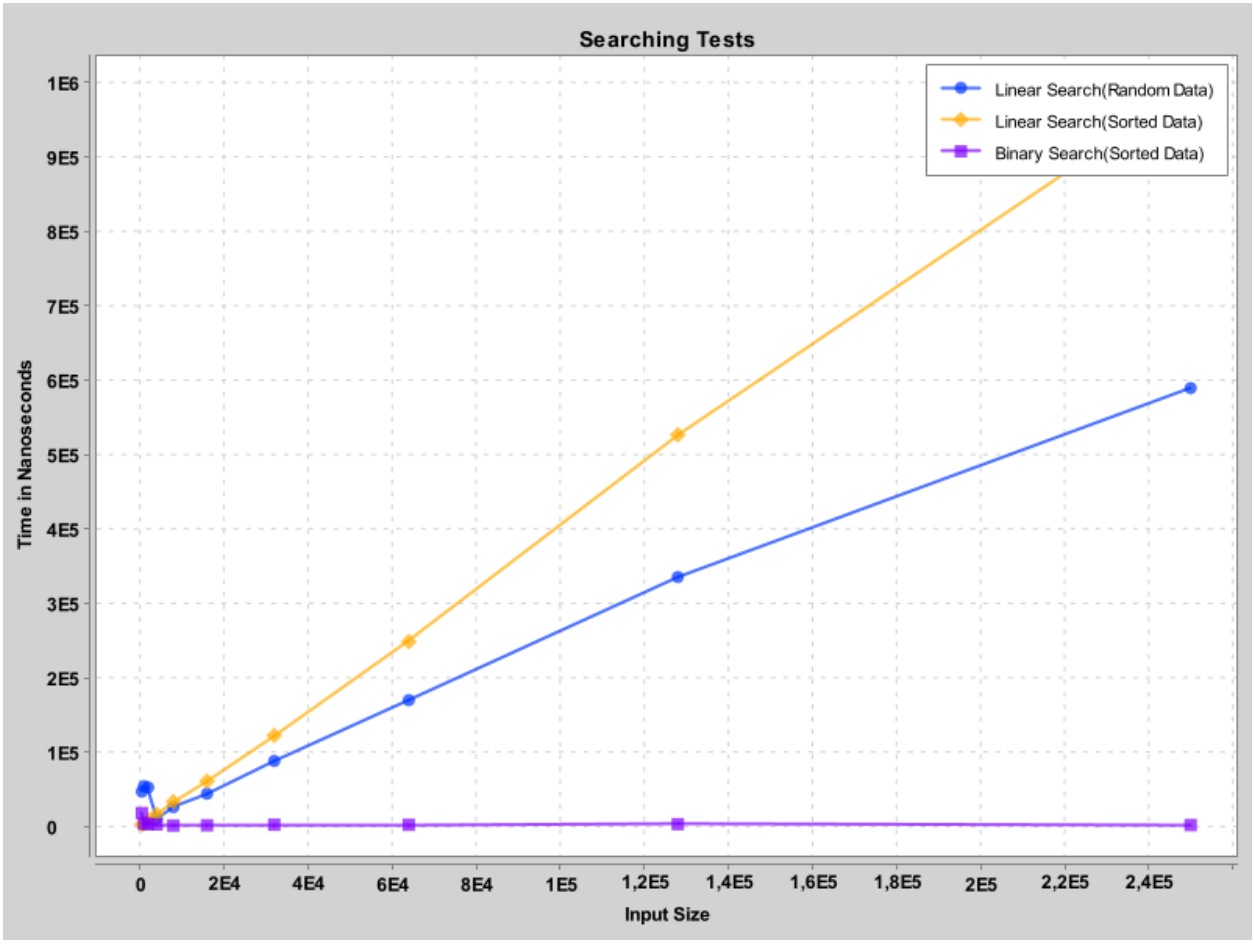

Figure 4: Searching Tests

3.3 Discussion

3.3.1 Insertion Sort

Insertion Sort Best Case: It is observed that the insertion sort, which theoretically has a best case complexity of O(n), behaves much faster than expected, even as O(1), by looking at the data in the table and graph. A difference was observed in the working process only for data with a size of 500000. These results may be affected by factors such as data caching, branch prediction, and testing conditions. O(n) value can be observed when more precise time measurements are made and larger data sets are used. 2

Insertion Sort Average Case: When we look at the test graph (number) of the insertion sort made with random data, which theoretically has an average case complexity of O(n^2), a graph similar to O(n^2) is observed. Likewise, when we look at the operating times in the table, although it is not exactly O(n^2), an increase close to it is observed. A closer result can be obtained when more tests are performed using larger data sets and with more precise measurements. 1

Insertion Sort Worst Case: When we look at the test graph made on the reverse sorted list, we observe a graph similar to the theoretical value of O(n^2), which is the insertion sort time complexity value. In the table, there are values closer to O(n^2) according to the working times with random data. According to the graphic and table results, it can be easily said that the worst case time complexity value is O(n^2) insertion space complexity. 3

Insertion Sort Space Complexity: When looking at the insertion sort code block, it can be said that the space complexity value is O(1) since all operations are performed within the list itself and no extra data structure is used.

3.3.2 Merge Sort

Merge Sort All Cases: Looking at the increase values in the table for merge sort, it is observed that there is a similar increase in running time for random, sorted and reversely sorted data. Likewise, when we look at the test graphs, we see a similar graph for all three data types. Although the best, worst and average case time complexity value $O(n \log n)$ cannot be observed exactly in the graph, it can be said that it is a graph close to it. The $O(n \log n)$ value can be observed much more clearly in the case of more precise graph drawing, more precise measurements and larger and more comprehensive tests. 1 2 3

Merge Sort Space Complexity: When looking at the code block for merge sort, it is observed that the list to be sorted is divided into two parts in the middle each time and 2 new lists are created. Since an extra data structure is used and the total size of the two lists created is the size of the original list, the space complexity value can be said to be O(n).

3.3.3 Counting Sort

Counting Sort All Cases: Counting sort table values are quite similar for all three data types. Normally, as the value of n increases, the duration is expected to increase, but it generally tends to remain more constant. When looking at the graphs, the exact O(n+k) value could not be obtained. While a more constant slope was observed for reversely sorted and random data graphs, a rather bumpy curve was observed for sorted data. The reasons for these may be that the k value is small (if the k value is small, it will not affect the duration much). Closer values can be obtained when more precise measurements and more comprehensive tests are made. 1 2 3

Counting Sort Space Complexity: When we look at the code block for counting sort, we see that 2 extra lists are created, the size of the list to be sorted and the size of the k value, and sorting is done with them. Therefore, it can be said that the space complexity value is O(n+k), which is the sum of the n value, which is the actual list length, and the k value, which is the largest element.

3.3.4 Linear Search & Binary Search

Linear Search & Binary Search Time Complexity: The best case time complexity for linear search is O(1), which occurs when the searched value is at the top of the list. The worst case is $O(n)$, which happens if the searched value is at the end of the list, because linear search searches from the beginning of the list to the end. Therefore, the average case value is $O(n)$.

For binary search, the average case time complexity is $O(\log_2 n)$ since a midpoint in the list is determined each time and half of the list is looked at depending on whether the searched value is greater or less than this midpoint. Best case is when the searched value is the first determined midpoint, and in this case the complexity is O(1). The result for the worst case is the same as the average case.

When looking at the search test graph, it can be observed that as the input size increases, the elapsed time increases linearly for both random and sorted data, as expected for linear search. Linear search is not affected by whether the data is sorted or not, so no improvement is observed over the sorted data graph. On the contrary, the times were longer in sorted data than in random data. 4

For binary search, sorted data is required and it can be concluded that it is a much better graphic than linear search. A graph with an average case time complexity value close to $O(\log_2 n)$ was observed. For both search algorithms, deviations in the graph and table may be due to the random values searched. 4

Linear Search & Binary Search Space Complexity: When looking at the code blocks of both search algorithms, the space complexity values are O(1) since no extra data structure is used for search.

References

[1] `https://www.geeksforgeeks.org/insertion-sort/`.

[2] `https://www.geeksforgeeks.org/merge-sort/`.

[3] `https://www.geeksforgeeks.org/counting-sort/`.

[4] `https://www.geeksforgeeks.org/linear-search/`.

[5] `https://www.geeksforgeeks.org/binary-search/`.