Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 18, 2023

Student name:
Burak Kurt

Student Number:
b2200765010

1 Problem Definition

Lots of data has been created and stored every day. To do some operations like sorting, searching on that data, efficient algorithms are needed on both time and space aspects.

2 Solution Implementation

Selection Sort , Quick Sort and Bucket Sort algorithms for sorting, Linear and Binary Search algorithms for searching have been implemented and their running time performances have been compared.

2.1 Selection Sort

```
1  public static int[] SelectionSort(int[] arr){
2      int[] sorted_arr = arr;
3      int Size = sorted_arr.length;
4      for(int i = 0; i< Size-1;i++){
5          int min = i;
6          for(int j = i+1 ; j < Size;j++){
7              if(sorted_arr[j] < sorted_arr[min]){
8                  min = j;
9              }
10         }
11         if ( min != i){
12             swap(sorted_arr,min,i);
13         }
14     }
15         return sorted_arr;
16     }
17 private static void swap(int[] arr,int i,int j){
18         int temp = arr[j];
19         arr[j] = arr[i];
20         arr[i] = temp;
21     }
```

2.2 Quick Sort

```
22 public static int[] QuickSort(int[] arr){
23         int[] sorted_arr = arr;
24
25         QuickSort(sorted_arr, 0, arr.length-1);
26
27         return sorted_arr;
28     }
29     private static void QuickSort(int[] arr,int low,int high){
```

```
30          int Size  = high - low +1;
31          int[] stack = new int[Size];
32          int top = -1;
33          stack[++top] = low;
34          stack[++top] = high;
35          while (top>=0){
36              high = stack[top--];
37              low = stack[top--];
38              int pivot = Partition(arr, low, high);
39              if(pivot -1 > low){
40                  stack[++top] = low;
41                  stack[++top] = pivot-1;
42              }
43              if(pivot+1 < high){
44                  stack[++top] = pivot+1;
45                  stack[++top] = high;
46              }
47          }
48      }
49
50      private static int Partition(int[] arr , int low , int high){
51          int pivot = arr[high];
52          int i = low-1;
53          for(int j = low ; j < high ; j++){
54              if(arr[j] <=pivot){
55                  i++;
56                  swap(arr, i, j);
57              }
58          }
59          swap(arr, i+1, high);
60          return i+1;
61      }
```

2.3 Bucket Sort

```
62  public static int[] BucketSort(int[] array) {
63          int[] sorted_arr = BucketSort(array, 0);
64
65          return sorted_arr;
66      }
67
68      private static int[] BucketSort(int[] array, int n) {
69          int Bucket_Num = (int) Math.sqrt((double) array.length);
70          ArrayList<Integer>[] buckets = new ArrayList[Bucket_Num];
71
72          for (int i = 0; i < Bucket_Num; i++) {
73              buckets[i] = new ArrayList<Integer>();
```

```
        }

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

        for (int i = 0; i < array.length; i++) {
            int hash_value = Hash(i, max, Bucket_Num);
            buckets[hash_value].add(array[i]);
        }

        for (int i = 0; i < Bucket_Num; i++) {
            Collections.sort(buckets[i]);
        }

        int[] sorted_arr = new int[array.length];
        int index = 0;
        for (int i = 0; i < Bucket_Num; i++) {
            for (int j = 0; j < buckets[i].size(); j++) {
                sorted_arr[index++] = buckets[i].get(j);
            }
        }

        return sorted_arr;

    }

    private static int Hash(int i, int max, int Bucket_Num) {
        return (int) Math.floor(i / max * (Bucket_Num - 1));
    }
```

2.4 Linear Search

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

2.5 Binary Search

```
115 public static int BinarySearch(int[] array, int value) {
116         int low = 0;
117         int high = array.length - 1;
118 
119         while (high - low > 1) {
120             int mid = (high + low) / 2;
121             if (value > array[mid]) {
122                 low = mid + 1;
123             } else {
124                 high = mid;
125             }
126         }
127         if (array[low] == value) {
128             return low;
129         }
130         if (array[high] == value) {
131             return high;
132         }
133         return -1;
134     }
```

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	0	1	5	19	78	296	1056	4087	16160
Quick sort	0	0	0	1	9	41	155	602	2754	10555
Bucket sort	0	0	0	0	0	1	1	2	7	6
	Sorted Input Data Timing Results in ms									
Selection sort	0	0	1	4	16	64	263	1054	4180	15960
Quick sort	0	0	0	3	13	48	167	623	2774	10646
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	5	19	65	253	1029	4193	15815
Quick sort	0	0	0	2	12	42	157	604	2726	10343
Bucket sort	0	0	0	0	0	0	1	0	1	4

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)	1139	1608	5898	15644	5737	2503	3912	5542	5895	7071
Linear search (sorted data)	130	190	447	734	1353	1956	3895	5086	4879	5993
Binary search (sorted data)	122	355	306	557	1124	2096	3690	5031	6430	7136

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	$O(n)$	$O(n)$	$O(n^2)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

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	$O(1)$

In selection sort, no matter what data order is (sorted or not) complexity is same because algorithm always compares right of the selected element for every element in array. As seen in Fig. 4 , running time is same for all data orders. Since Selection Sort is a inplace algorithm, it does swap operations inside of the array. so algorithm does not require any extra space.

Worst case of Quick Sort happens when the partition element is highest or lowest element in the array. It could be happen in sorted or reversely sorted arrays. Average and best case happens when partition element is middle element of the array. Then it can successfully divide array into 2 equal part and so on, log(n) terms comes from dividing part. Quick sort uses extra array to sort elements in it, it needs extra (n) space to run.

If all the elements are stored in few buckets (means that range is low in dataset) and selection sort is used to sort elements in the bucket, worst case happens. Otherwise, if elements are distributed homogeneous in buckets, it's complexity becomes linear time. With this advantage, Bucket Sort runs faster than other two sorting algorithms. It runs faster when array is sorted. For space complexity, n element is stored in extra k bucket, so space complexity is O(n+k).

Linear Search begins searching from first element of array to the last. Best case happens when searched item is in the first index. In the worst case, item is in the last index of the array. It does not require any space because it does search operation on the original array.

Binary Search divides array into half in each iteration. With this property, it runs faster than linear search. Best case happens when searched element is in the middle of the array. Space complexity is O(1) because it does search operation on array same as linear search.

4 Notes

In Fig.5, there is no difference between running times, although it should be. Quick Sort is expected to run slower in sorted and reversely sorted data. That error might be happen from computer's computational power at that moment. Also, linear search runs faster than binary search for input size larger than 64000 in Fig.7 and plots are not smooth except binary search plot. This inconsistency comes from randomness in experiment.

References

- https://www.geeksforgeeks.org/quick-sort/
- https://www.scaler.com/topics/data-structures/bucket-sort/
- https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm

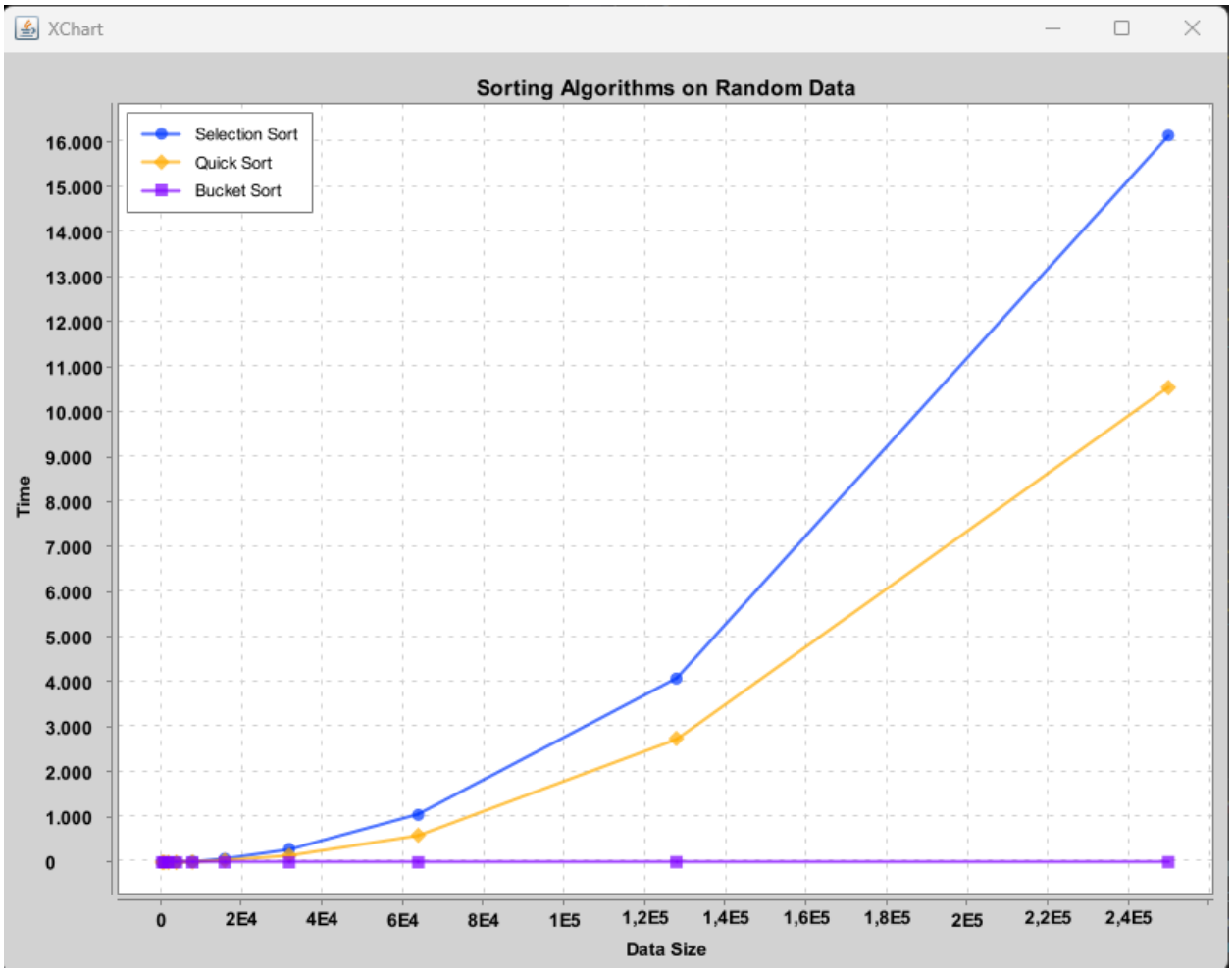

Figure 1: Sorting algorithms performance on random input.

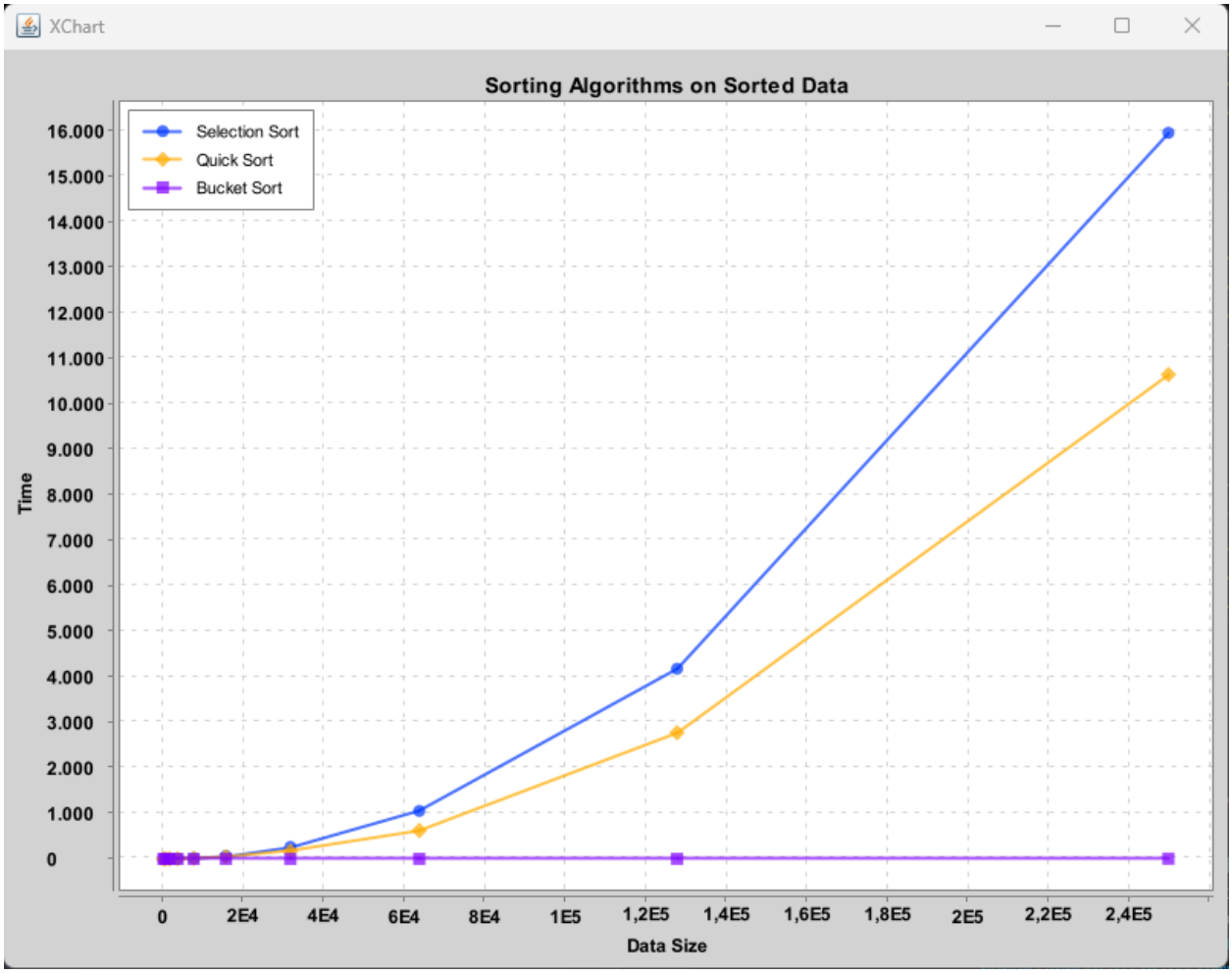

Figure 2: Sorting algorithms performance on sorted input.

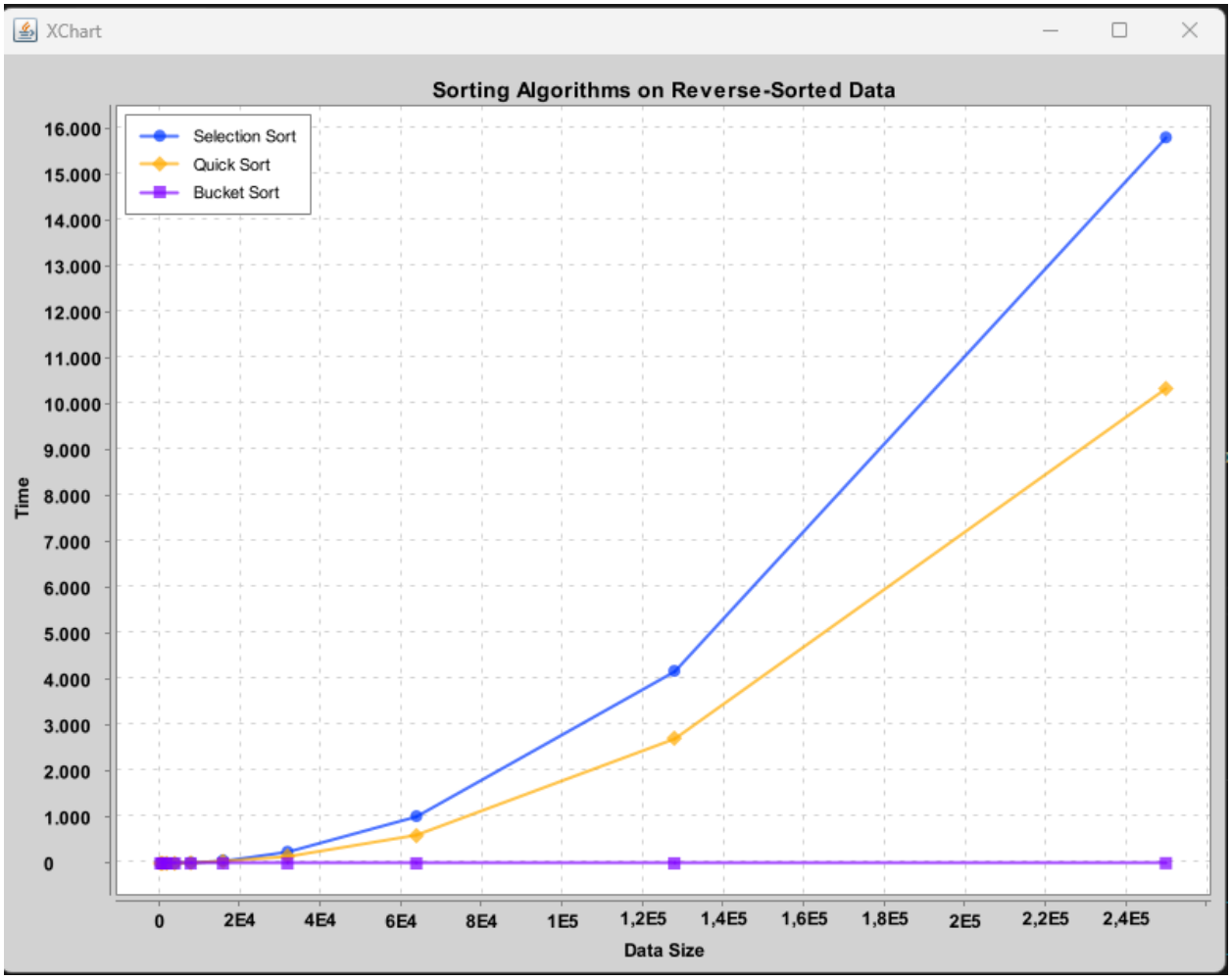

Figure 3: Sorting algorithms performance on reversely-sorted input.

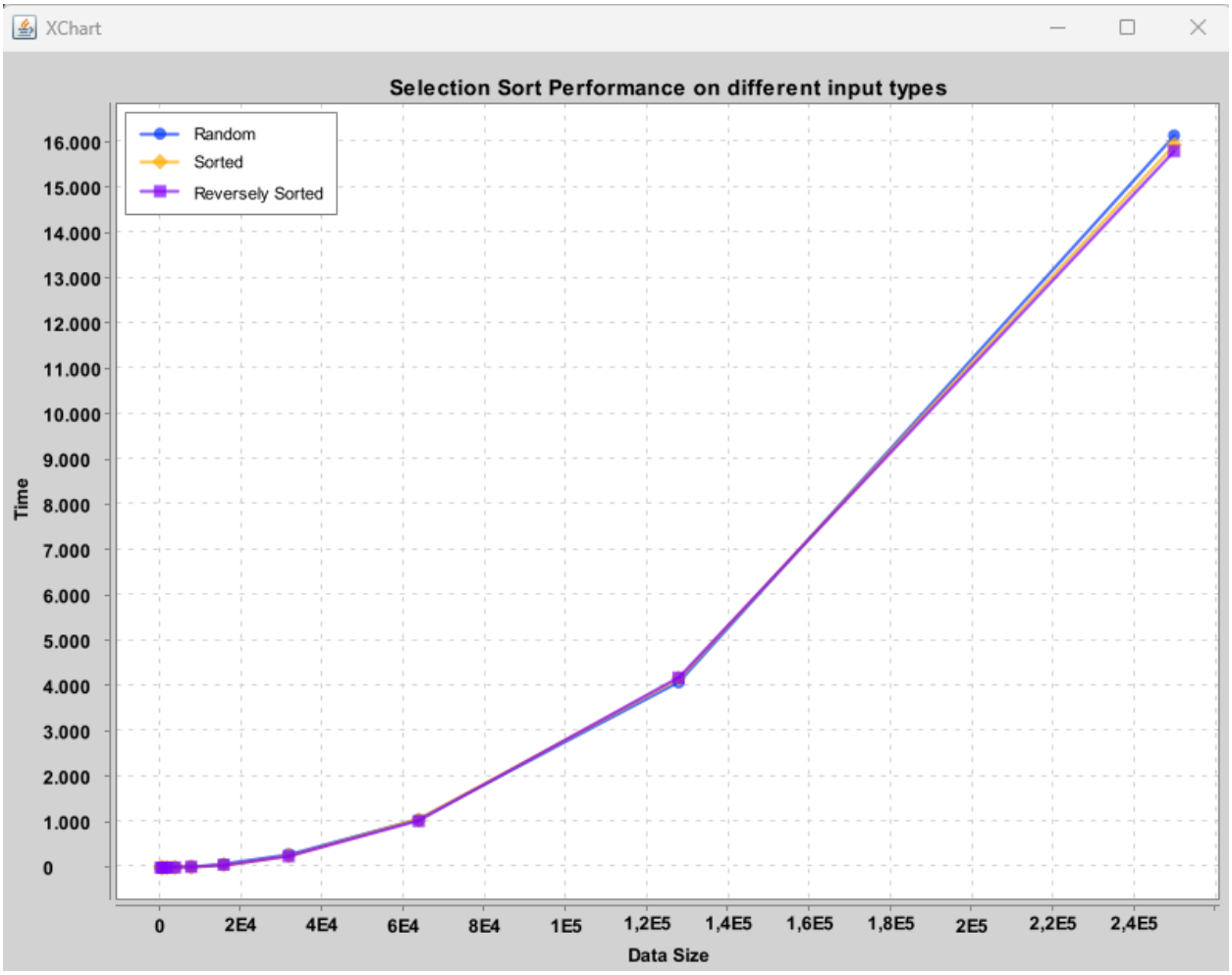

Figure 4: Selection sort performance depending on input type.

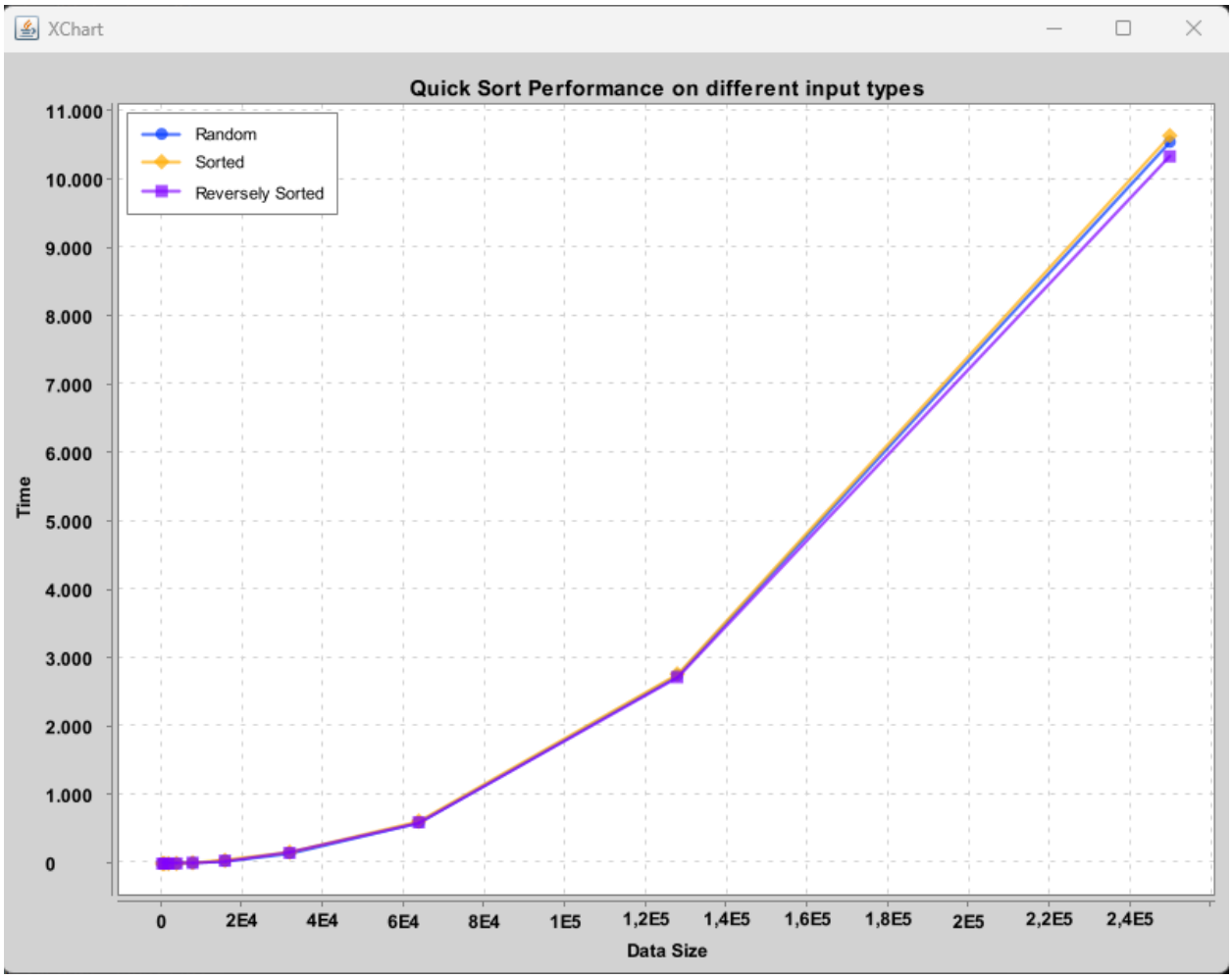

Figure 5: Quick sort performance depending on input type.

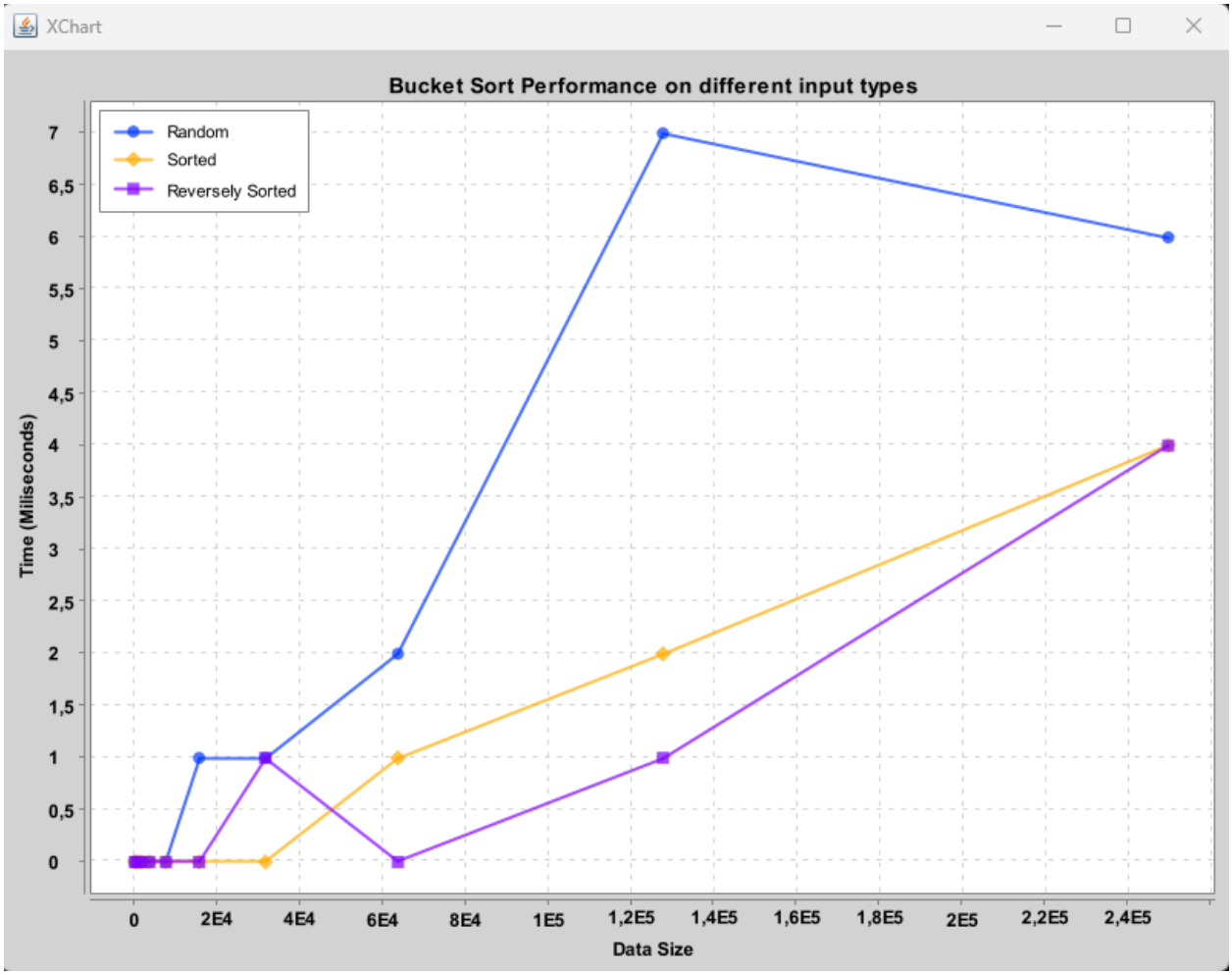

Figure 6: Bucket sort performance depending on input type.

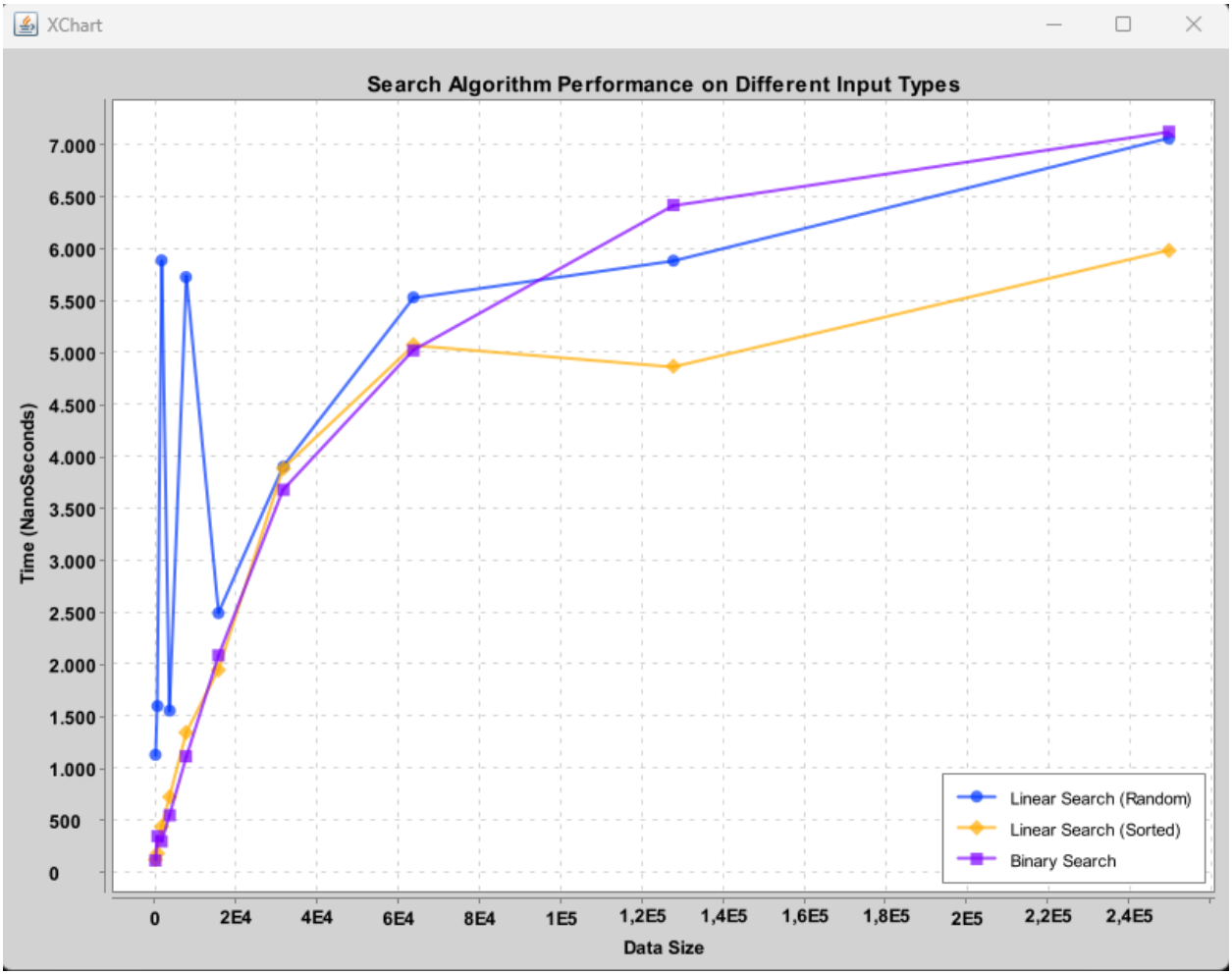

Figure 7: Search algorithms performance.