Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 29, 2023

Student name:
Ece Sena Etoğlu

Student Number:
b2210356016

1 Problem Definition

The main objective is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities.

2 Solution Implementations

2.1 Selection Sort

```java
public class SelectionSort {
    public static void sort(double[] arr) {
        for(int i = 0; i<arr.length-1;i++) {
            int min = i;
            for(int j = i+1; j<arr.length;j++) {
                if(arr[j] < arr[min]) {
                    min = j;
                }
            }
            //swap if min element has changed
            if(min != i) {
                double temp = arr[i];
                arr[i] = arr[min];
                arr[min] = temp;
            }
        }
    }
}
```

2.2 Quick Sort

```java
public class QuickSort {
    private static void sort(double[] input,int lowPointer,int highPointer) {
        // iterative quicksort using stack
        int[] stack = new int[highPointer - lowPointer + 1];
        int top = -1;
        int pivot;
        // put initial pointers
        stack[++top] = lowPointer;
        stack[++top] = highPointer;

        // keep popping until stack is empty
        while (top >= 0) {
            // Pop highPointer and lowPointer
            highPointer = stack[top--];
            lowPointer = stack[top--];
```

```
34
35              // put pivot in correct position
36              pivot = partition(input, lowPointer, highPointer);
37              // push elements left side of the pivot to the left of stack
38              if (pivot - 1 > lowPointer) {
39                  stack[++top] = lowPointer;
40                  stack[++top] = pivot - 1;
41              }
42              // push elements right side of the pivot to the right of stack
43              if (pivot + 1 < highPointer) {
44                  stack[++top] = pivot + 1;
45                  stack[++top] = highPointer;
46              }
47          }
48      }
49
50      private static int partition(double[] input,int low,int high) {
51          //pick last element as pivot
52          double pivot  = input[high];
53          int i = low -1;
54          for(int j = low; j<=high-1;j++) {
55
56              if(input[j] <= pivot) {
57                  i++;
58                  //swap input[i] with input[j]
59                  double temp = input[i];
60                  input[i] = input[j];
61                  input[j] = temp;
62              }
63          }
64          //swap input[i+1] input[high]
65          double temp = input[i+1];
66          input[i+1] = input[high];
67          input[high] = temp;
68
69          return i+1;
70      }
```

2.3 Bucket Sort

```
71  public class BucketSort {
72
73      public static double[] sort(double[] array) {
74          //take the floor of the result
75          int numOfBuckets = (int)Math.sqrt(array.length);
76          double max = findMax(array);
77
78          //2d list to store buckets
79          ArrayList<ArrayList<Double>> buckets =new ArrayList<>();
80          for(int i = 0; i<numOfBuckets;i++) {
81              buckets.add(new ArrayList<>());
82          }
83
84          //map every val in array to a bucket by a hash func.
85          for(double val:array) {
86              int id = hash(val,max,numOfBuckets);
87              buckets.get(id).add(val);
88          }
89
90          //sort every bucket
91          //combine all elements in sorted buckets to a final array
92          //do above operations together in a loop to avoid extra traversal of
                combining
93          double[] sortedArr = new double[array.length];
94          int counter = 0;
95
96          for(ArrayList<Double> bucket:buckets) {
97              Collections.sort(bucket);
98              for (Double val:bucket) {
99                  sortedArr[counter] = val;
100                 counter++;
101             }
102         }
103         return sortedArr;
104     }
105     private static int hash(double i,double max,int numOfBuckets) {
106
107         //takes the floor of the result by casting to int
108         return (int)(i/max * (numOfBuckets-1));
109     }
```

findMax function in line 76 returns the max value in the array. For simplification the code is not included.

2.4 Linear Search

```
110 public class LinearSearch {
111     public static  int search(double[] arr, double key) {
112         for(int i = 0; i<arr.length;i++) {
113             if(arr[i] == key) {
114                 return i;
115             }
116         }
117         return -1;
118     }
119 }
```

2.5 Binary Search

```
120 public class BinarySearch {
121     public static  int search(double[] arr,double key) {
122         int low = 0;
123         int high = arr.length-1;
124 
125         while(high - low > 1) {
126             int mid = (low + high) / 2;
127 
128             //go right
129             if(arr[mid] < key) {
130                 low = mid + 1;
131             }
132             else {
133                 high = mid;
134             }
135         }
136         if(arr[low] == key) {
137             return low;
138         }
139         else if (arr[high] == key) {
140             return high;
141         }
142         return -1;
143     }
144 }
```

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	0.29611	0.32304	1.52029	5.03513	20.2018	85.72659	283.29476	1034.2413	4098.95503	15642.79438
Quick sort	0.01113	0.0275	0.08326	0.21744	0.52058	0.99949	3.36597	11.80281	34.69637	63.01536
Bucket sort	0.05246	0.11575	0.25823	0.54573	1.11607	2.33628	6.01066	10.91496	20.54261	49.37731
	Ascending Sorted Input Data Timing Results in ms									
Selection sort	0.07465	0.25052	0.99011	3.99672	17.32428	63.94661	256.43328	1024.08108	4236.12948	15642.70498
Quick sort	0.11134	0.41291	1.59897	6.53834	25.85067	101.45846	403.15027	1615.163	6444.01584	24647.04894
Bucket sort	0.0114	0.02531	0.03968	0.05116	0.10084	0.22307	0.80755	1.03695	2.10559	5.03332
	Descending Sorted Input Data Timing Results in ms									
Selection sort	0.08704	0.35876	1.27155	5.17059	19.85287	78.59419	329.84229	1357.89204	5800.70461	28229.90813
Quick sort	0.07409	0.23416	0.77882	2.47713	7.2278	14.27705	46.01527	200.42163	768.84952	1630.90066
Bucket sort	0.00974	0.07702	0.10077	0.15981	0.25428	0.61209	0.91011	1.53813	3.23207	9.17847

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)	3903.1	3164.8	1949.0	1308.5	2684.2	3677.8	6666.7	11371.6	24379.9	107303.7
Linear search (sorted data)	604.4	873.6	1416.2	2257.6	4193.7	7798.1	12591.1	20412.4	36774.5	118599.6
Binary search (sorted data)	450.8	450.7	474.5	439.5	517.4	529.1	653.8	747.6	657.6	871.2

There are several factors affecting the runtime of an algorithm such as:

- How large is the CPU cache.
- Which programming language is used.
- Which operating system is used.

Hence the obtained measurements might be different in different runs or from other measurements found online. To make sense of the datas, datas are examined through plots.

Complexity analysis tables are given in 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	$\Omega(n)$	$\Theta(n + n^2/k + k)$	$O(n \log n)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

- **Bucket Sort:** For complexity calculations see Bucket Sort. Note, algorithm's worst case is dominated by the algorithm used to sort each bucket 97
- **Linear Search:** For complexity calculations see Linear Search Best case occurs when the searched element is the first element of the list.
- **Binary Search:** For complexity calculations see Binary Search. Best case occurs when the searched element is the middle element of the list.

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	$O(1)$

- **Bucket Sort:** See lines in the code 93, 81 for space complexity calculation.
- **Linear Search, Binary Search:** Iterative algorithms hence O(1) space complexity.

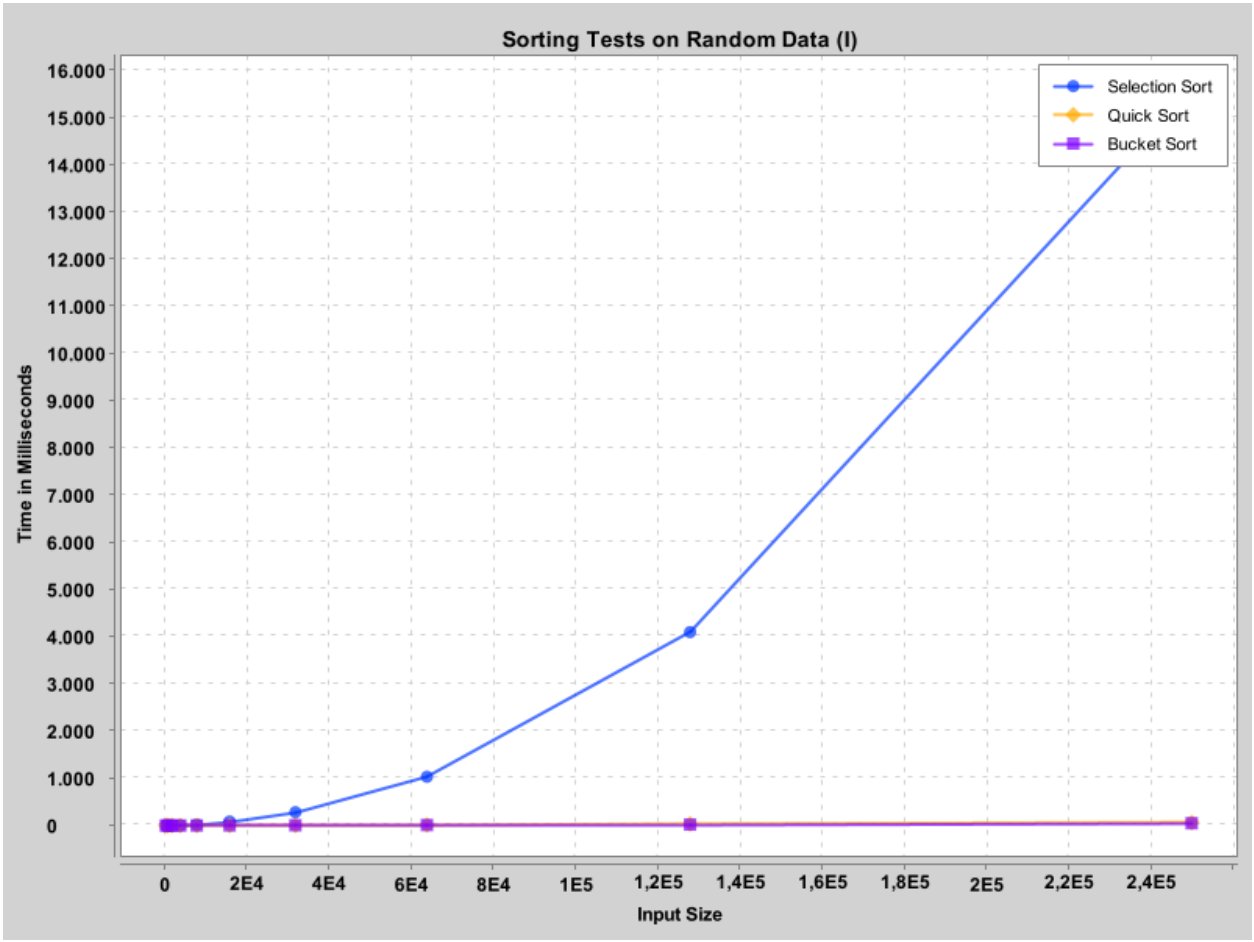

Figure 1: Plot of the Sorting Tests On Random Data

On random data Quick Sort partitions tends to be balanced hence random data is the best case for Quick Sort. In the plot it is seen that Selection Sort has worse complexity than Quick Sort. Matches with the theory since Selection Sort has worse complexities in all cases of Quick Sort except for the worst case of Quick Sort.

See figure 2 to see comparison of Quick Sort and Bucket Sort.

Balanced and Unbalanced Partitions: In Quick Sort, unbalanced partitioning is when n sized array partitions into (n-1) sized array, disabling the advantage of logarithmic growth. Unbalanced partitioning occurs when the picked pivot is greater than, less than or equal to all elements.

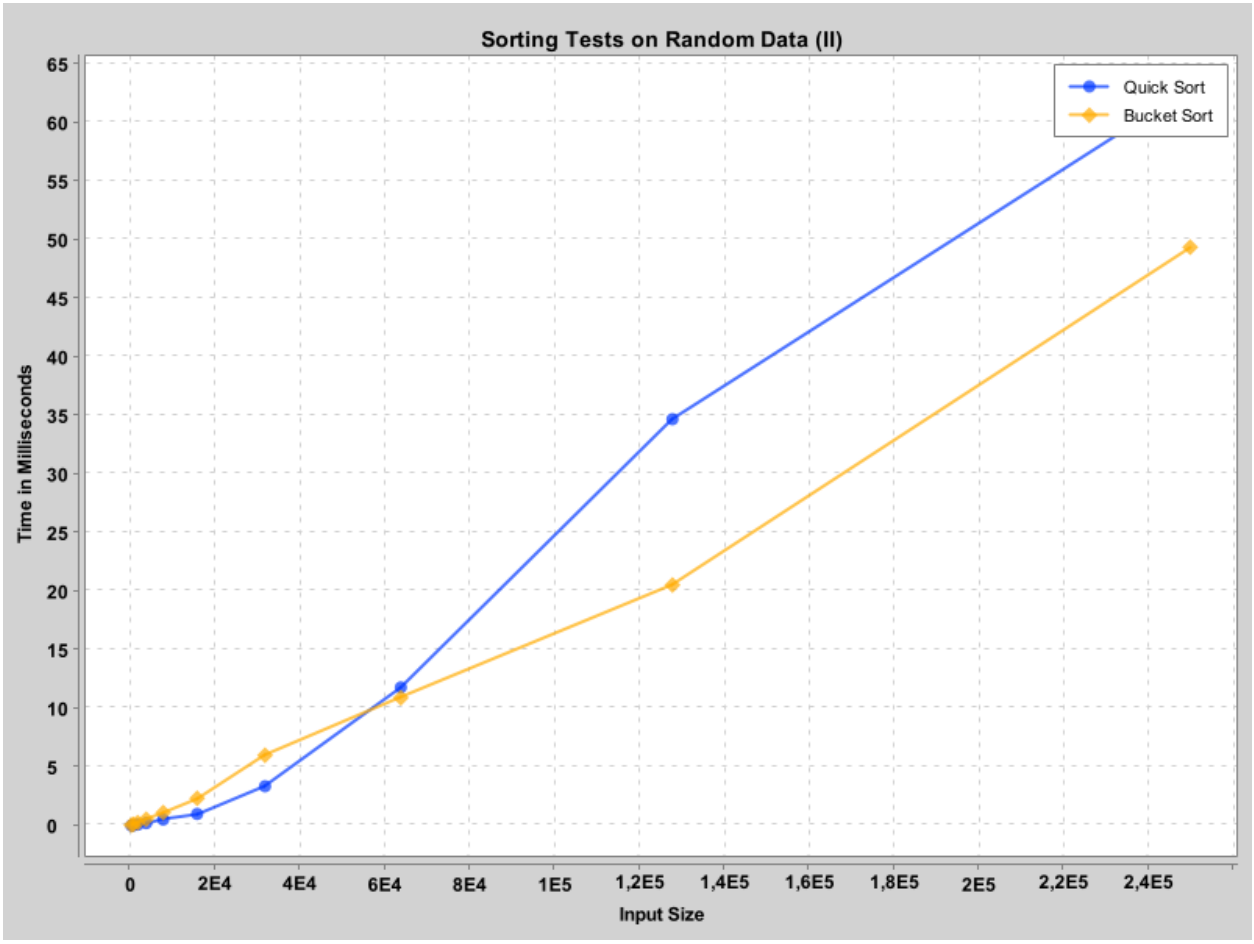

Figure 2: Comparison of Quick Sort and Bucket Sort on Random Data

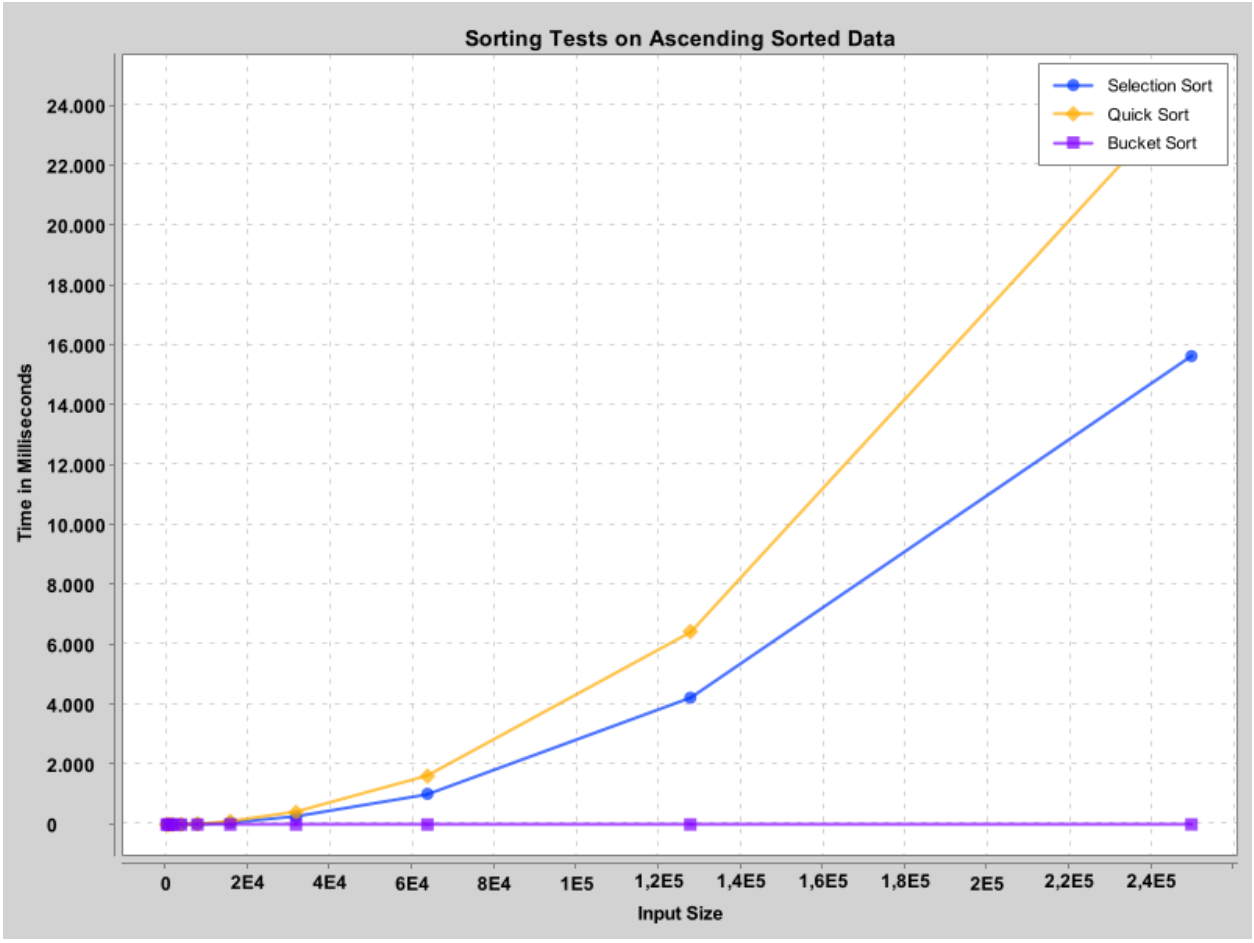

Figure 3: Sorting Tests on Ascending Sorted Data

Quick Sort is slower than Selection Sort due to the unbalanced partitioning of the Quick Sort algorithm in this data set. Hence leads to the worst case of Quick Sort.

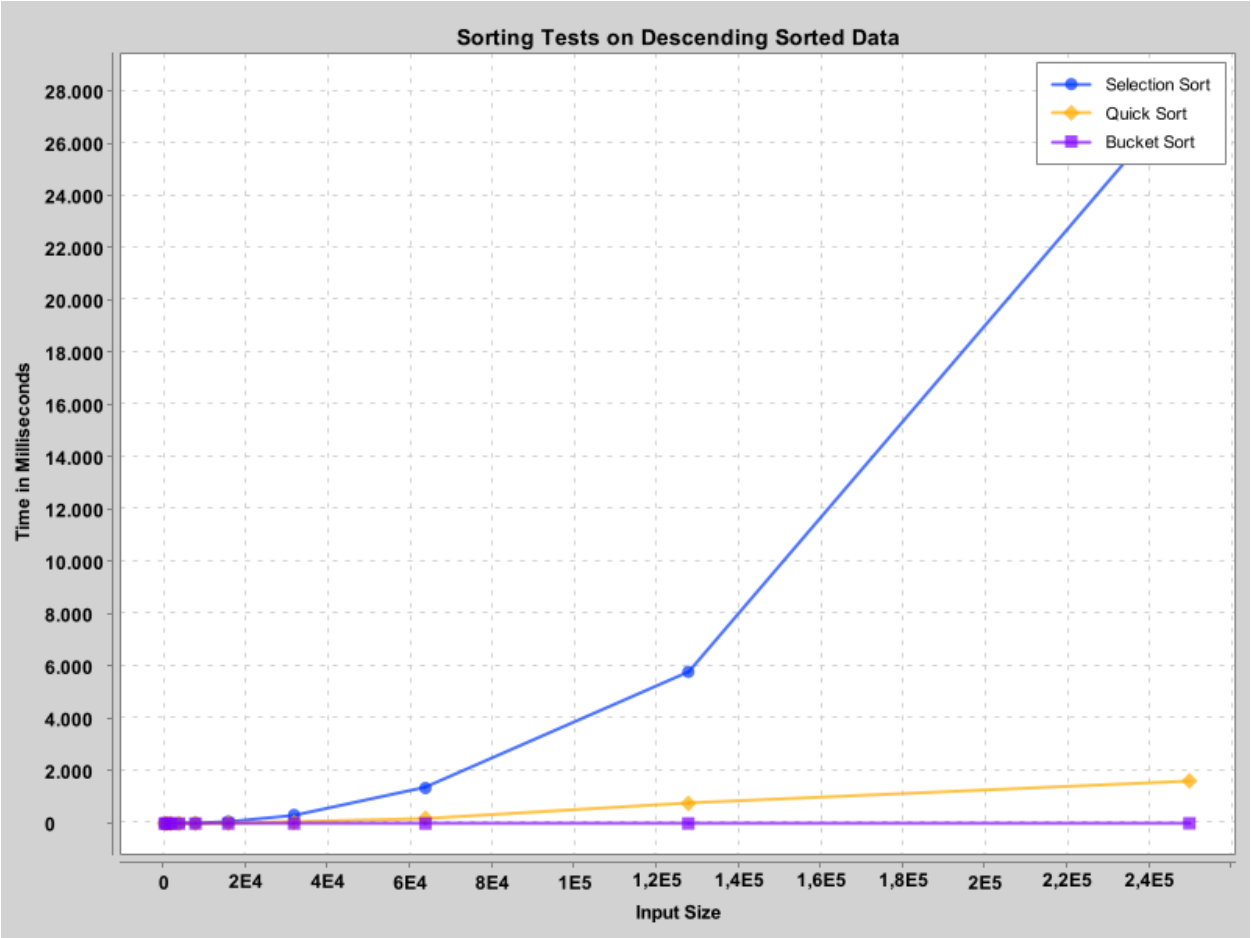

Figure 4: Sorting Tests on Descending Sorted Data

Although Quick Sort has unbalanced partitioning also in this data set, swap operations are less comparing to ascending sorted data due to the algorithm. Hence Quick Sort is faster than Selection Sort due to the hidden constant in worst case complexity.

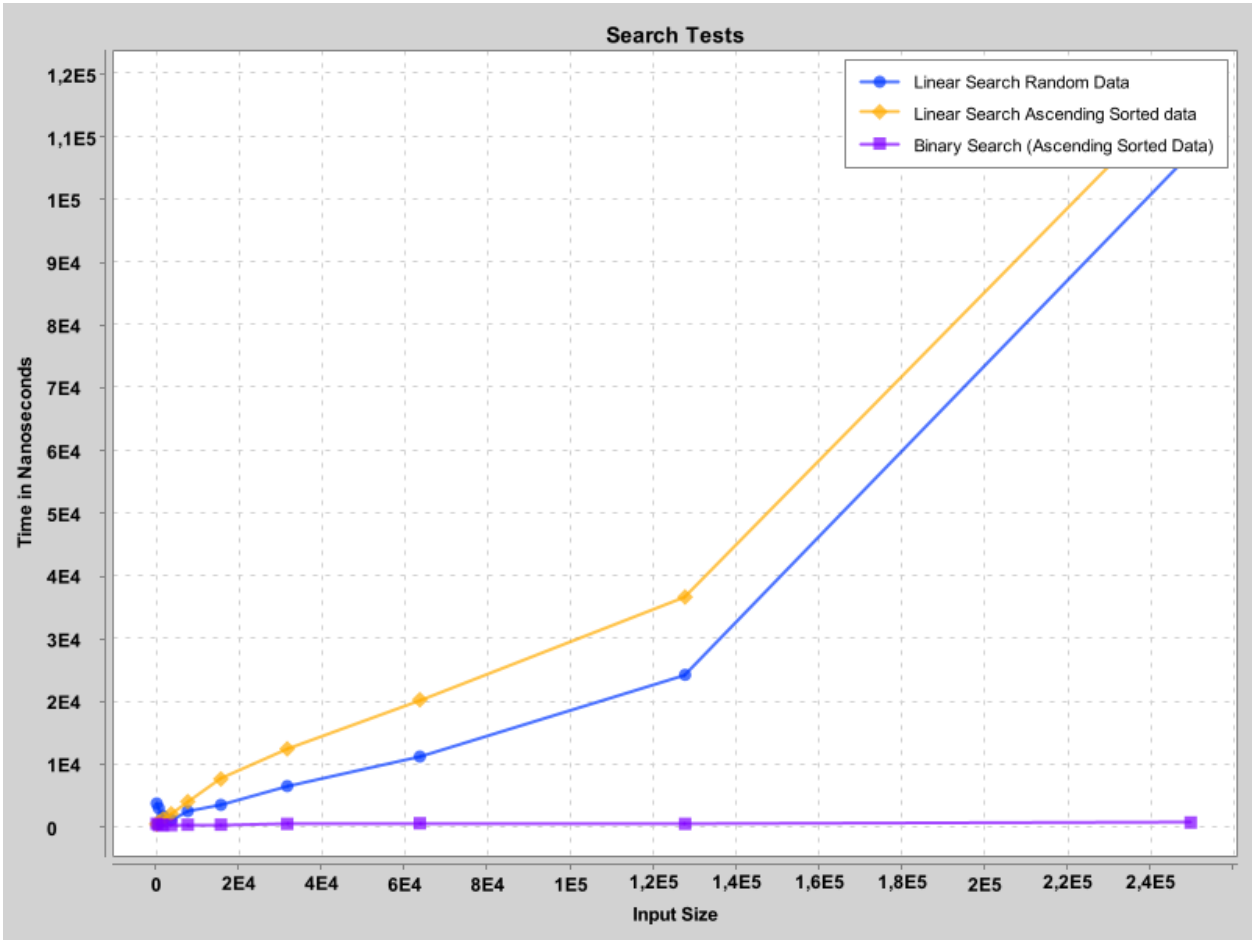

Figure 5: Search Tests

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

  According to the measurements and the plots: (Only best and worst cases are described as average case determination did not seem possible.)

  Selection Sort: Descending sorted is the worst case, ascending sorted is the best case.

  Quick Sort: Ascending sorted is the worst case, random data is the best case.

  Bucket Sort: Ascending sorted is the best case, random data is the worst case.

  Linear Search: Sorted is the best case, however it is observed that on different measurements plots yield different best cases, as it is not significant whether the list is sorted for linear search.

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

  Selection Sort: As seen and justified in the plots, matches to the theory.

  Quick Sort: As seen and justified in the plots, matches to the theory.

  Bucket Sort: Due to the reason given at notes, it is not applicable to compare the results with the theory. But as seen in plots, it is faster than Quick Sort and Selection Sort hence matches with the comparison theory between algorithms.

  Linear Search,Binary Search: It is seen from 5 that Binary Search is better than Linear Search, Linear time complexity is observed in Linear Search, matches with the theory. Constant time complexity is observed in Binary Search, does not match with the theory.

4 Notes

Bucket Sort's time complexity is mainly determined by the input data's distribution into buckets.

In the conducted experiments, data sets in 3 types of inputs (random,ascending sorted,descending sorted) is the same set in each size. Since the data set is the same, measurements reflect the only cause of difference that is sorting algorithm's behaviour in sorting the buckets.
To examine the theory for Bucket Sort, experiments should be conducted with different data sets for each size.

References

- Bucket Sort

- Linear Search
- Binary Search