Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 28, 2023

Student name:
Betül Sema MANAV

Student Number:
b2200356019

1 Problem Definition

Classifying selection, quick, bucket sort and linear, binary search algorithms based on computational (time) complexity and auxiliary memory (space) complexity.

2 Solution Implementation

See my implementations of the five algorithms below. I implemented all of them according to the given pseudocodes.

2.1 Selection Sort

```
1  public int[] selectionSort(int[] arrToSort, int arrLength) {
2      int[] sortedArr = arrToSort;
3      arrLen = arrLength;
4      long startTime = System.nanoTime(); // nanoTime for extra precision (will
           be converted later)
5
6      for (int i = 0; i < arrLen; i++) {
7          int minIndex = i;
8          for (int j = i + 1; j < arrLen; j++) {
9              if (sortedArr[j] < sortedArr[minIndex]) {
10                 minIndex = j;
11             }
12         }
13         if (minIndex != i) {
14             int temp = sortedArr[i];
15             sortedArr[i] = sortedArr[minIndex];
16             sortedArr[minIndex] = temp;
17         }
18     }
19
20     long duration = System.nanoTime() - startTime;
21     runningTimes.add(duration);
22
23     return sortedArr;
24 }
```

2.2 Quick Sort

```
25 public int[] quickSort(int[] arrToSort, int arrSize) {
26     sortedArr = arrToSort;
27     arrLen = arrSize;
28     long startTime = System.nanoTime(); // nanoTime for extra precision (will
           be converted later)
```

```
29
30      int high = arrSize - 1;
31      int low = 0;
32      ArrayList<Integer> stack = new ArrayList<>();
33      int topIndex = -1;
34      stack.add(++topIndex, low);
35      stack.add(++topIndex, high);
36
37      while (topIndex >= 0) {
38          high = stack.get(topIndex);
39          stack.remove(topIndex);
40          topIndex--;
41
42          low = stack.get(topIndex);
43          stack.remove(topIndex);
44          topIndex--;
45
46          int pivot = partition(low, high);
47          if (pivot - 1 > low) {
48              stack.add(++topIndex, low);
49              stack.add(++topIndex, pivot - 1);
50          }
51          if (pivot + 1 < high) {
52              stack.add(++topIndex, pivot + 1);
53              stack.add(++topIndex, high);
54          }
55      }
56
57      long duration = System.nanoTime() - startTime;
58      runningTimes.add(duration);
59
60      return sortedArr;
61  }
62
63  private int partition(int low, int high) {
64      int pivot = sortedArr[high];
65      int i = low - 1;
66      for (int j = low; j < high; j++) {
67          if (sortedArr[j] <= pivot) {
68              i++;
69              int temp = sortedArr[i];
70              sortedArr[i] = sortedArr[j];
71              sortedArr[j] = temp;
72          }
73      }
74      int temp = sortedArr[i + 1];
75      sortedArr[i + 1] = sortedArr[high];
76      sortedArr[high] = temp;
```

```
77
78      return i + 1;
79  }
```

2.3 Bucket Sort

```
80  public int[] bucketSort(int[] arrToSort, int arrSize) {
81      int[] sortedArr = arrToSort;
82      arrLen = arrSize;
83      long startTime = System.nanoTime(); // nanoTime for extra precision (will
            be converted later)
84
85      double arrSizeDouble = arrSize;
86      int numberOfBuckets = (int) Math.ceil(Math.sqrt(arrSizeDouble));
87      ArrayList<ArrayList<Integer>> buckets = new ArrayList<>();
88
89      // initializing the buckets with empty buckets
90      for (int i = 0; i < numberOfBuckets; i++) {
91          ArrayList<Integer> placeHolderList = new ArrayList<>();
92          buckets.add(placeHolderList);
93      }
94
95      int maxVal = Arrays.stream(arrToSort).max().getAsInt();
96
97      // distributing values among buckets
98      for (int val : sortedArr) {
99          ArrayList<Integer> tempBucket = buckets.get(hash(val, maxVal,
                numberOfBuckets));
100         tempBucket.add(val);
101         buckets.set(hash(val, maxVal, numberOfBuckets), tempBucket);
102     }
103
104     // sorting each bucket individually
105     for (int index = 0; index < numberOfBuckets; index++) {
106         ArrayList<Integer> tempBucket = buckets.get(index);
107         Collections.sort(tempBucket);
108         buckets.set(index, tempBucket);
109     }
110
111     int index = 0;
112     for (ArrayList<Integer> arrayList : buckets) {
113         for (int val : arrayList) {
114             sortedArr[index] = val;
115             index++;
116         }
117     }
118
```

```
119     long duration = System.nanoTime() - startTime;
120     runningTimes.add(duration);
121 
122     return sortedArr;
123 }
124 
125 private int hash(double val, double max, double numberOfBuckets) {
126     return (int) Math.floor((val / max) * (numberOfBuckets - 1));
127 }
```

2.4 Linear Search

```
128 public int linearSearch(int[] arrToSearch, int valToFind, int arrSize) {
129     arrLen = arrSize;
130     long startTime = System.nanoTime();
131 
132     for (int index = 0; index < arrSize; index++) {
133         if (arrToSearch[index] == valToFind) {
134             long duration = System.nanoTime() - startTime;
135             runningTimes.add(duration);
136             return index;
137         }
138     }
139 
140     long duration = System.nanoTime() - startTime;
141     runningTimes.add(duration);
142     return -1;
143 }
```

2.5 Binary Search

```
144 public int binarySearch(int[] arrToSearch, int valToFind, int arrSize) {
145     arrLen = arrSize;
146     long startTime = System.nanoTime();
147 
148     int low = 0;
149     int high = arrSize - 1;
150     while (high - low > 1) {
151         int mid = (high + low) / 2;
152         if (arrToSearch[mid] < valToFind) {
153             low = mid + 1;
154         } else {
155             high = mid;
156         }
157     }
```

```
158     if (arrToSearch[low] == valToFind) {
159         return low;
160     } else if (arrToSearch[high] == high) {
161         return high;
162     }
163 
164     long duration = System.nanoTime() - startTime;
165     runningTimes.add(duration);
166     return -1;
167 }
```

3 Results, Analysis, Discussion

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	0.66	1.34	3.04	9.48	33.31	125.80	503.45	2656.61	11431.91	43015.55
Quick sort	0.26	0.68	1.45	1.64	1.18	2.14	7.58	35.39	123.58	219.12
Bucket sort	0.74	2.02	2.27	1.59	2.88	3.55	6.17	15.04	45.77	101.82
	Sorted Input Data Timing Results in ms									
Selection sort	0.31	1.61	7.94	38.01	59.51	229.17	682.70	2869.57	12009.90	44884.80
Quick sort	0.54	3.94	22.20	70.71	117.49	499.61	1587.57	6949.70	25460.22	97588.81
Bucket sort	0.51	0.72	1.36	2.13	0.44	1.03	1.38	2.88	7.17	15.42
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.16	0.66	2.23	8.29	33.88	128.41	708.11	2895.85	12801.98	46216.35
Quick sort	0.35	1.32	4.84	18.83	76.66	298.12	1568.60	6231.60	26364.24	102479.45
Bucket sort	0.03	0.08	0.10	0.16	0.32	0.56	1.42	2.77	8.09	17.70

For the randomised data the theoretical average time complexities match as expected.

When it comes to sorted and reversely sorted data quick sorts worst case occurs and it runs on $O(n^2)$ complexity.

In the implementation of bucket sort I have used Collections.sort() to sort the individual buckets and that's what probably caused the running time difference between randomised and sorted data cases. Thus bucket sort seems to have run on best/average cases in all situations.

Bucket sorts worst case would be if during the distribution process all the data was collected in a single or too few buckets. This case seems to have not occurred in this experiment with this set of data.

Selection sort seems to have $O(n^2)$ complexity in all cases which is what's expected as its best, average and worst cases are the same.

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)	2679.68	2008.75	6932.00	6345.09	1552.19	2428.28	4957.89	9395.26	18942.35	38593.06
Linear search (sorted data)	73.56	217.64	256.05	807.49	1121.21	2228.29	5553.68	12042.75	24347.73	50101.76
Binary search (sorted data)	54.19	330.05	160.75	144.43	48.07	27.04	39.62	63.03	91.45	129.67

The best case for linear search would be if the element sought after was in the first index and the worst case would be if the value was in the last index. When a lot of random experiments occur, as we've done in this experiment, this irregularity smooths out, forming a linear graph as can be seen in the plot.

The best case for binary is also similar, if the searched value is the middle element then the complexity would be $\Omega(1)$. The worst case would be if the target element is the smallest or the largest element in the sorted list.

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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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	$O(1)$

As also can be seen in the Table 3 bucket sorts best and average cases work with $\Omega(n+k)$ where n is the length of the list and k is the number of buckets. If the data to be sorted has a tendency to be repetitive a lot, I would not recommend using bucket sort considering that its complexity would then come close to quadratic time.

Regarding the space complexities of these algorithms, selection sort does not require space because it utilizes swapping while sorting. On the other hand quick sort needs $O(n)$ space although it does not create any other containers than the initial array. In light of all this information we can say that selection sort is the most efficient in terms of auxiliary space complexity since bucket sort needs space for the buckets, on top of the space the initial array requires.

Searching operations only require space for the target element and thus have a space complexity of $O(1)$.

The lines where extra space is allocated for the algorithms would be initial array creation for quick sort and bucket sort, also creating the bucket list in the bucket sort algorithm.

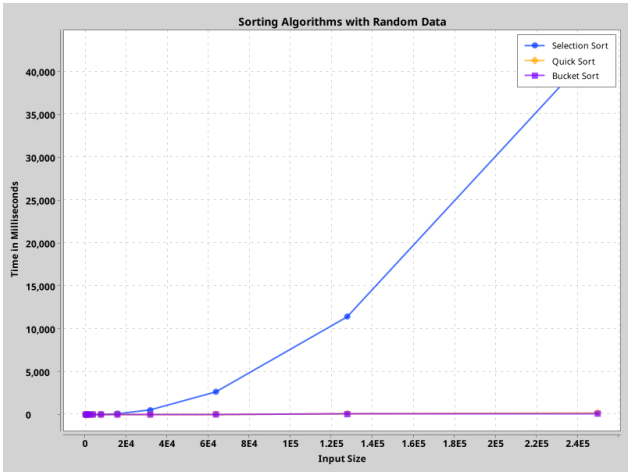

Figure 1: Sorting Algorithms with Random Data

Quick sort and bucket sort seems to overlap since their run times are very minor compared to selection sorts immense running time due to its quadratic time complexity. We can observe its quadratic growth in the plot.

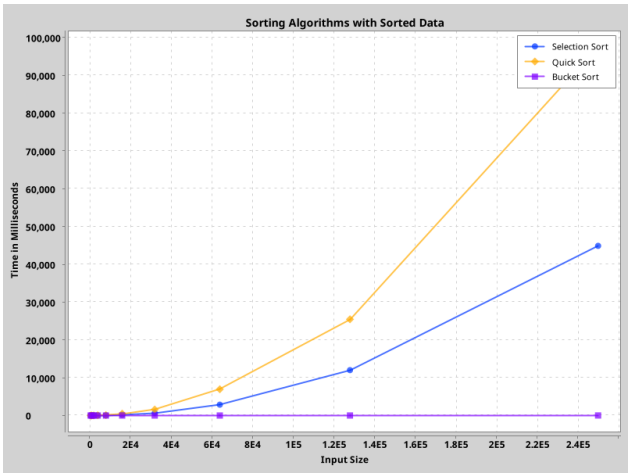

Figure 2: Sorting Algorithms with Sorted Data

When we perform the experiment with sorted data, quick sorts run time jumps up significantly due to its worst case and we can observe its quadratic growth on the graph along with selection sort. Bucket sort seems to follow a linear trend.

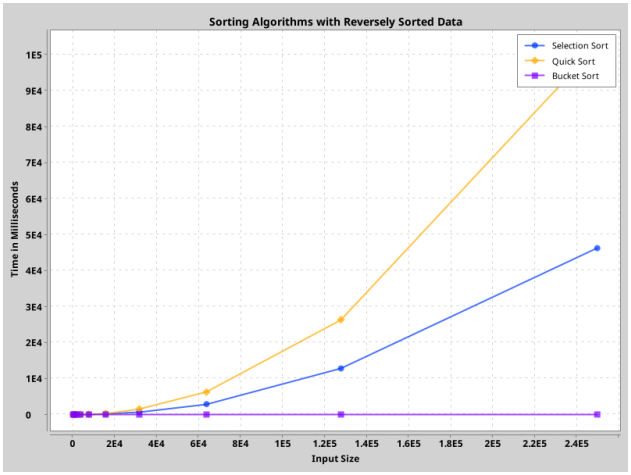

Figure 3: Sorting Algorithms with Reversely Sorted Data

The plot of the reversely sorted data seems to resemble the previous plot which was on sorted data.

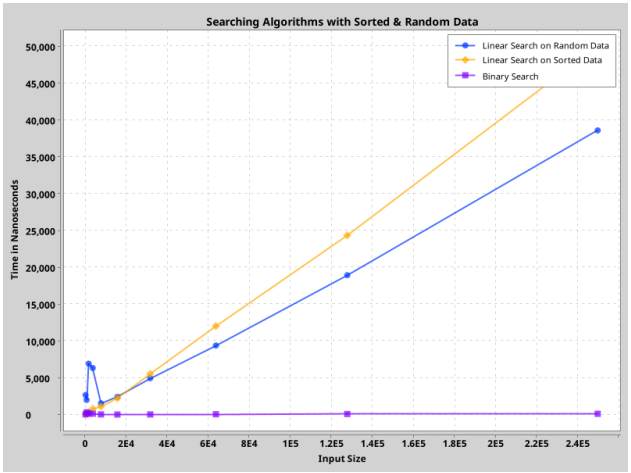

Figure 4: Searching Algorithms with Sorted And Random Data

In this plot we notice a small spike on the run time of linear search on random data. I could not ascertain as to why that happens every time I've run the program but it may have to do with the caching behavior of the CPU and the memory allocation and management by the operating system. Other than that the plot seems to be within our expectations based on our theoretical knowledge.

4 Notes

- I considered keeping a copy of the sorted arrays for later use but decided that it would be better to sort the array initially with Arrays.sort() in order to increase the readability.
- You can compile it with *javac -cp *.jar *.java -d .* and run it with *java -cp .:* Main TrafficFlowDataset.csv*. The data file name needs to be given through the command line as the first argument.
- I've written the program with version *openjdk 19.0.2 2023-01-17* and in linux.

References

- https://www.baeldung.com/java-generating-random-numbers-in-range
- https://www.baeldung.com/java-slicing-arrays
- https://copyprogramming.com/howto/conversion-of-nanoseconds-to-milliseconds-and-nanoseconds-999999-in-java
- https://www.javatpoint.com/how-to-sort-an-array-in-java