Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 29, 2023

Student name:
Tugrul ACAR

Student Number:
b2210356144

1 Problem Definition

In this project , we are expected to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities and we have to implement 3 sorting and 2 searching algorithms given as pseudocodes, and perform a set of experiments on the given datasets to show that the empirical data follows the corresponding asymptotic growth functions. These algorithms are Selection Sort ,Quick Sort, Bucket Sort, Linear Search and Binary Search.

2 Solution Implementation

2.1 Selection Sort Algorithm

```
1  public static void selectionSort(int arr[])
2      {
3          int n = arr.length;
4          for (int i = 0; i < n-1; i++)
5          {
6              int min_i = i;
7              for (int j = i+1; j < n; j++)
8                  if (arr[j] < arr[min_i])
9                      min_i = j;
10             int temp = arr[min_i];
11             arr[min_i] = arr[i];
12             arr[i] = temp;
13         }
14     }
```

And you can reference line **??** in the code like this.

2.2 Quick Sort Algorithm

```
15 public static int partition(int arr[], int low, int high)
16     {
17         int pivot = arr[high];
18         int i = low - 1;
19         for (int j = low; j < high; j++) {
20             if (arr[j] <= pivot) {
21                 i++;
22                 int temp = arr[i];
23                 arr[i] = arr[j];
24                 arr[j] = temp;
25             }
26         }
27         int temp = arr[i + 1];
28         arr[i + 1] = arr[high];
29         arr[high] = temp;
```

```
30          return i + 1;
31      }
32
33      public static void quickSort(int arr[], int l, int h)
34      {
35          int[] stack = new int[h - l + 1];
36          int top = -1;
37          stack[++top] = l;
38          stack[++top] = h;
39          while (top >= 0) {
40              h = stack[top--];
41              l = stack[top--];
42              int p = partition(arr, l, h);
43              if (p - 1 > l) {
44                  stack[++top] = l;
45                  stack[++top] = p - 1;
46              }
47              if (p + 1 < h) {
48                  stack[++top] = p + 1;
49                  stack[++top] = h;
50              }
51          }
52      }
```

And you can reference line **??** in the code like this.

2.3 Bucket Sort Algorithm

```
53  public static void bucketSort(int[] array) {
54          int numberOfBuckets = (int)Math.sqrt(array.length);
55          ArrayList<Integer>[] buckets = new ArrayList[numberOfBuckets];
56          for (int i = 0; i < numberOfBuckets; i++) {
57              buckets[i] = new ArrayList<>();
58          }
59
60          int max = array[0];
61          for (int i = 1; i < array.length; i++) {
62              if (array[i] > max) {
63                  max = array[i];
64              }
65          }
66
67          for (int i = 0; i < array.length; i++) {
68              int bucketIndex = hash(array[i], max, numberOfBuckets);
69              buckets[bucketIndex].add(array[i]);
70          }
71
72          for (int i = 0; i < numberOfBuckets; i++) {
```

```
73              Collections.sort(buckets[i]);
74          }
75
76          int index = 0;
77          for (int i = 0; i < numberOfBuckets; i++) {
78              for (int j = 0; j < buckets[i].size(); j++) {
79                  array[index] = buckets[i].get(j);
80                  index++;
81              }
82          }
83      }
84
85      public static int hash(int number, int max, int numberOfBuckets) {
86          return (int) Math.floor(number / (double) max * (numberOfBuckets - 1))
                ;
87      }
```

And you can reference line **??** in the code like this.

2.4 Linear Search Algorithm

Example how to add Java code:

```
88  public static int linearSearch(int arr[], int x)
89      {
90          int N = arr.length;
91          for (int i = 0; i < N; i++) {
92              if (arr[i] == x)
93                  return i;
94          }
95          return -1;
96      }
```

And you can reference line **??** in the code like this.

2.5 Binary Search Algorithm

Example how to add Java code:

```
97  public static int binarySearch(int v[], int find_item)
98      {
99          int lo = 0, hi = v.length - 1;
100         while (hi - lo > 1) {
101             int mid = (hi + lo) / 2;
102             if (v[mid] < find_item) {
103                 lo = mid + 1;
104             }
105             else {
106                 hi = mid;
```

```
            }
        }
        if (v[lo] == find_item) {
            return lo;
        }
        else if (v[hi] == find_item) {
            return hi;
        }
        else {
            return -1;
        }
    }
```

And you can reference line **??** in the code like this.

3 Results, Analysis, Discussion

Your explanations, results, plots go in this section...
Selection sort:
Best Case: Sorted data
Average case: Random data.
Worst case: Sorted in reverse order
Quick sort:
Best Case: when the partitioning process always divides the list into two equal halves
Average case: when the partitioning process divides the list into two sub-lists of roughly equal size
Worst case: when the partitioning process always picks the largest or smallest element as the pivot
Bucket sort:
Best Case: when the elements in the array are uniformly distributed among the buckets.
Average case: Random data.
Worst case: when all the elements in the array fall into the same bucket.
Linear search:
Best Case: When the target element is found at the first position of the array
Average case: when the target element is found somewhere in the middle of the array
Worst case: when the target element is located at the last position of the array.
Binary search:
Best Case: when the target element is found at the middle of the array
Average case: when the target element is found somewhere in the array.
Worst case: when the target element is not in the array or it is located at the edge of the array.

Obtained results (running times of your algorithm implementations) matches their theoretical asymptotic complexities. Running time test results are given in Table 1 2

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	0	3	14	55	215	866	3432	12981
Quick sort	0	0	0	0	0	1	2	10	30	53
Bucket sort	0	0	0	1	1	4	7	9	19	42
	Sorted Input Data Timing Results in ms									
Selection sort	0	0	0	3	13	52	210	851	3381	12997
Quick sort	0	0	1	5	22	85	345	1398	5728	21417
Bucket sort	0	0	0	0	0	0	0	1	2	4
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0	0	0	3	13	52	209	851	3385	13029
Quick sort	0	0	1	5	21	84	341	1356	5436	21550
Bucket sort	0	0	0	0	0	0	0	0	2	5

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)	2992	3950	709	1112	2193	3859	7514	12794	22975	46886
Linear search (sorted data)	206	307	550	941	2092	3869	7630	13534	22784	43824
Binary search (sorted data)	304	137	155	165	116	118	124	126	133	155

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**
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	$O(logn)$
Bucket Sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

Selection Sort doesn't need any additional space. So auxiliary space complexity is O(1)

The iterative implementation of **quick sort** uses a stack to store information about sub-arrays that need to be sorted. Specifically, it stores the start and end indices of each sub-array. During each iteration, the top of the stack is popped and its corresponding sub-array is partitioned. The resulting sub-arrays are then pushed onto the stack if they are not yet sorted. The depth of the stack is proportional to the depth of the recursion tree in the recursive implementation of quick sort. Since the recursion tree has a depth of O(log n) for average case.

The size of the auxiliary array of **buckets** is proportional to the number of elements in the input array, which is O(n). Additionally, the number of buckets used by bucket sort is typically less than or equal to the range of the input elements, which is represented by the parameter k. Therefore, the total auxiliary space complexity of bucket sort is O(n + k).

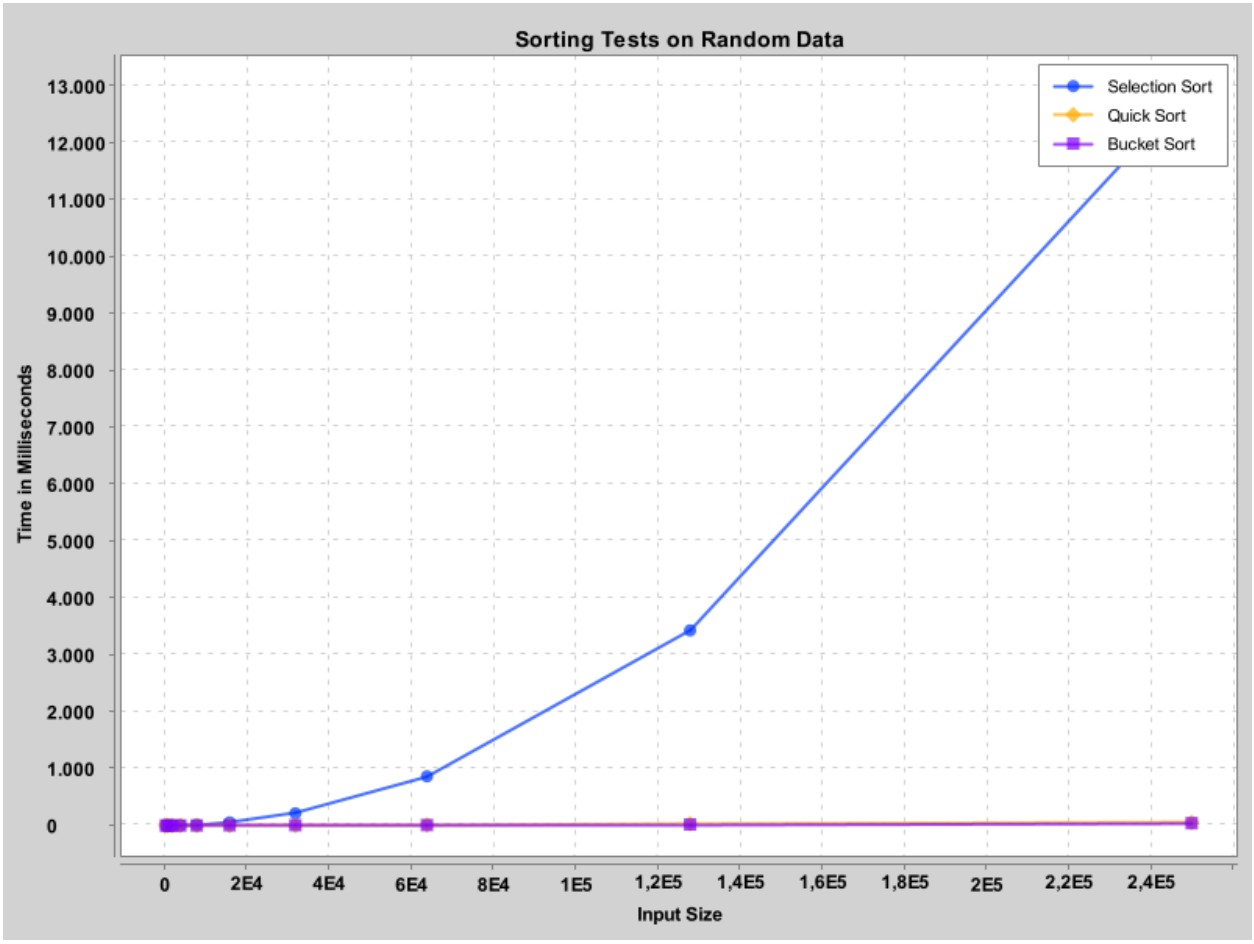

Figure 1: Tests on Random Data

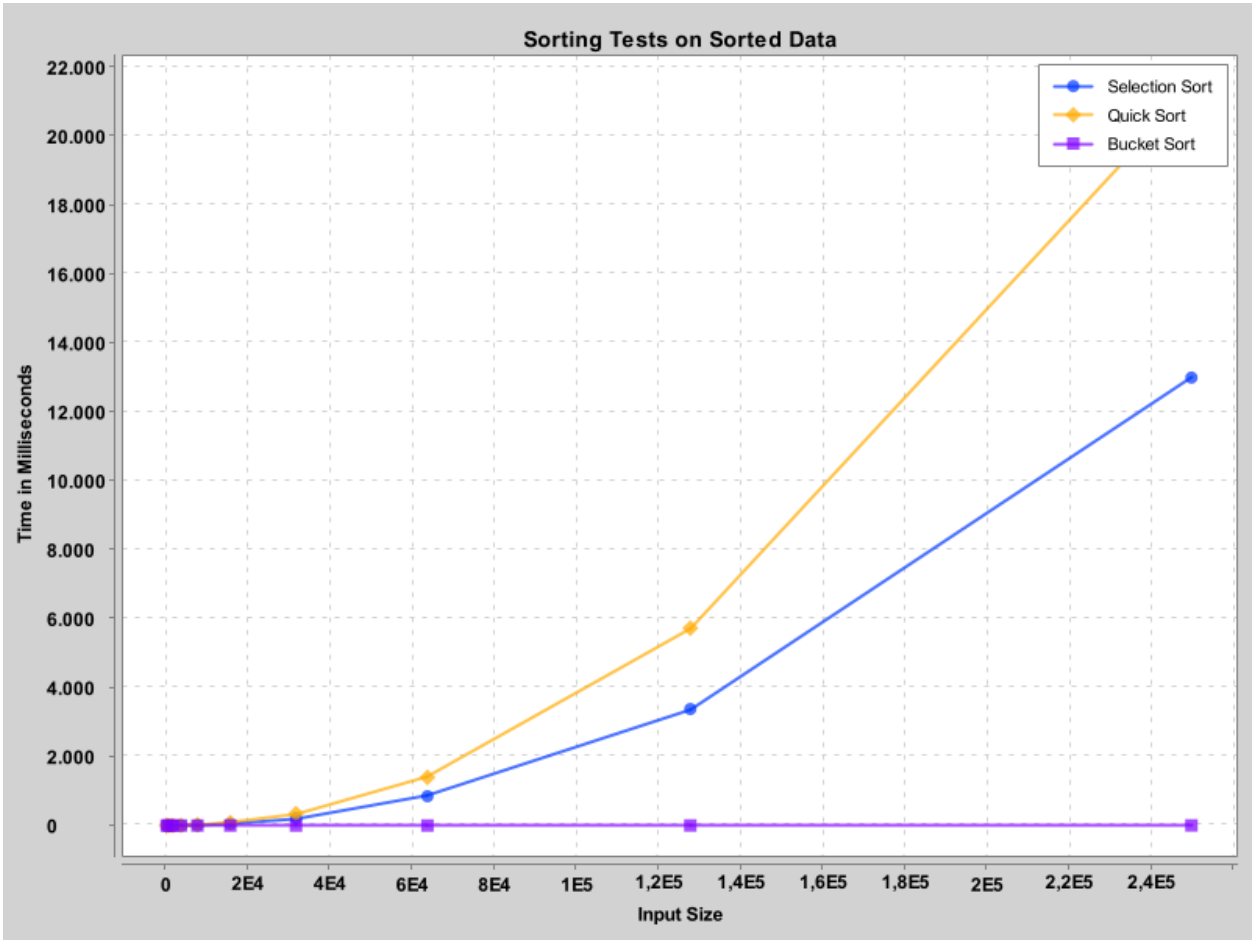

Figure 2: Tests on Sorted Data

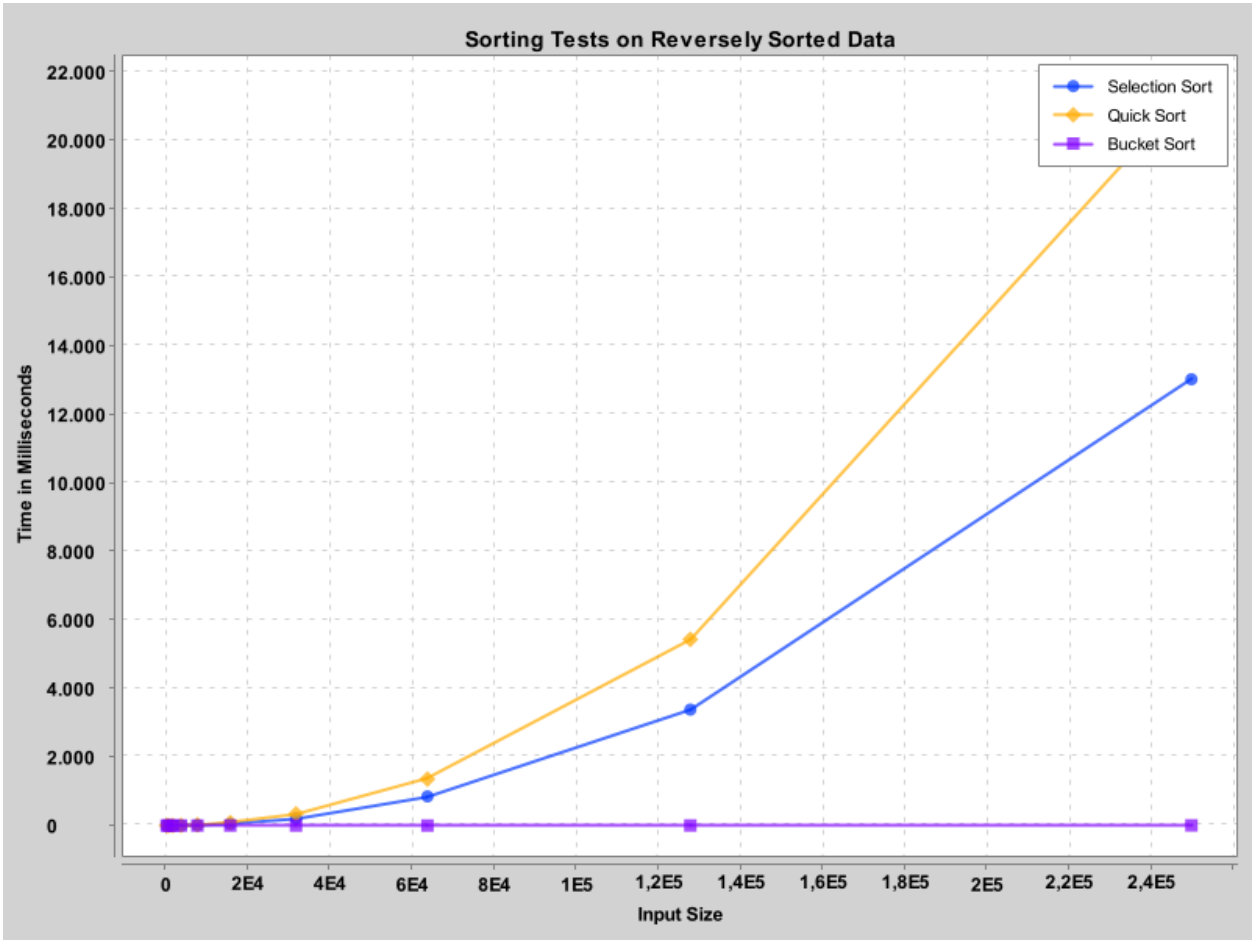

Figure 3: Tests on Reverse Sorted Data

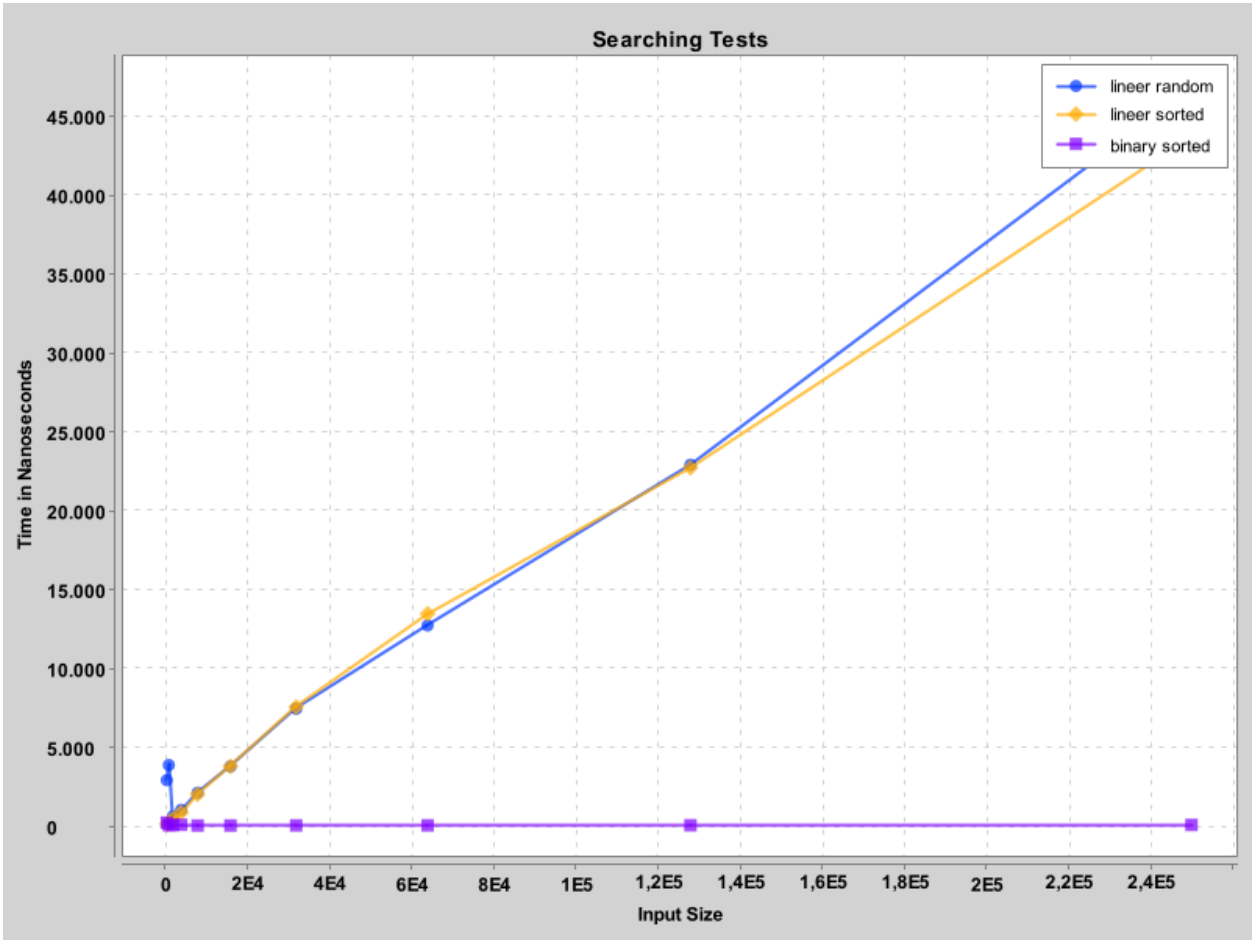

Figure 4: Tests on Search.

References

- https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/
- https://www.geeksforgeeks.org/insertion-sort/
- https://www.geeksforgeeks.org/quick-sort/
- https://www.geeksforgeeks.org/bucket-sort-2/
- https://www.geeksforgeeks.org/linear-search/
- https://www.geeksforgeeks.org/binary-search/