Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 27, 2023

Student name:
Melike Akkaya

Student Number:
b2210356124

1 Problem Definition

Analysis of algorithms is the area of computer science that provides tools to analyze the efficiency of different methods of solutions.
Efficiency of an algorithm depends on these parameters;
i) how much time,
ii) memory space,
iii) disk space it requires.
Analysis of algorithms is mainly used to predict performance and compare algorithms that are developed for the same task. Also it provides guarantees for performance and helps to understand theoretical basis.
In this assignment, the implementations of different search and sort algorithms on inputs of different sizes were evaluated.

2 Solution Implementation

Certain steps were followed in order to perform the algorithm analysis correctly:
• Algorithms implemented correctly.
• The time required for each basic operation were determined using nanoTime() function. The result in nanoseconds has been converted to milliseconds for easier evaluation.
• Unknown quantities that can be used to describe the frequency of execution of the basic operations had been identified.
• A realistic model for the input to the program had been developed.
• The unknown quantities analized, assuming the modeled input.
• The total running time by multiplying the time by the frequency for each operation calculated, then adding all the products.

2.1 Structure

Six java files were created.
One of them is Main (file reading is done here, a function has been created to automate the graphing process to be used in the analysis of both search and sorting algorithms).
Search class with a value of which all search operations are made.
The remaining four were created to perform sorting operations. Since there is a lot of work to be done for sorting algorithms and it is thought that they will stand more regularly in this way, too many classes have been included. While 3 of them (BucketSort, QuickSort, SelectionSort) include the implementation of the algorithms requested in the assignment, ExperimentsWithSortingAlgorithms includes the operations to be done with these three algorithms.

2.2 Selection Sort

```java
// this function finds and returns the index of the smallest member of the array
public static int findMin (int[] numbers, int startIndex) {
    int minIndex = startIndex;

    for (int i = startIndex; i < numbers.length; i++) {
        if (numbers[i] < numbers[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// this function uses the findMin function and detects the smallest element in the list.
// then places the smallest element in the right place.
public static void sort(int[] numbers, int size) {
    for (int i = 0; i < size; i++) {
        int minIndex = findMin(numbers, i);

        int temp = numbers[minIndex];
        numbers[minIndex] = numbers[i];
        numbers[i] = temp;
    }
}
```

2.3 Quick Sort

** IMPORTANT: While making this implementation, the pseudocode given in the assignment pdf was greatly utilized.

```java
public static void swap (int[] numbers, int firstElement, int secondElement) {
    int temp = numbers[firstElement];
    numbers[firstElement] = numbers [secondElement];
    numbers[secondElement] = temp;
}

// used to divide the array into two parts
public static int partition (int[] numbers, int low, int high) {
    int pivot = numbers[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (numbers[j] <= pivot) {
            i++;
            swap (numbers, i, j);
        }
    }
    swap (numbers, i+1, high);
    return i+1;
}
public static void sort (int[] numbers, int high, int low) {
    int stackSize = high - low + 1;
    int[] stack = new int[stackSize];

    int top = -1;

    stack[++top] = low;
    stack[++top] = high;

    while (top >= 0) {
        high = stack[top--];
        low = stack[top--];
        int pivot = partition(numbers, low, high);

        if (pivot - 1 > low) {
            stack[++top] = low;
            stack[++top] = pivot - 1;
        }
        if (pivot + 1 < high) {
            stack[++top] = pivot + 1;
            stack[++top] = high;
        }
    }
}
```

2.4 Bucket Sort

** IMPORTANT: While making this implementation, the pseudocode given in the assignment pdf was greatly utilized.

```java
public static void sort(int[] numbers) {
    int numberOfBuckets = (int) Math.ceil(Math.sqrt(numbers.length));

    List<Integer>[] buckets = new List[numberOfBuckets];

    for (int i = 0; i < numberOfBuckets; i++) {
        buckets[i] = new ArrayList<>();
    }

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

    for (int k : numbers) {
        int bucketIndex = hash(k, max, numberOfBuckets);
        buckets[bucketIndex].add(k);
    }

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

    int index = 0;
    for (int i = 0; i < numberOfBuckets; i++) {
        for (int j = 0; j < buckets[i].size(); j++) {
            numbers[index++] = buckets[i].get(j);
        }
    }
}

public static int hash(int i, int max, int numberOfBuckets) {
    return (int) Math.floor(i * 1.0 / max * (numberOfBuckets - 1));
}
```

2.5 Linear Search

```java
public static int linearSearch(ArrayList<Integer> numbers, int value, int size
    ) {
     for (int i = 0; i < size; i++) {
         if (value == numbers.get(i)) {
             return i;
         }
     }
     return -1; //not found
}
```

2.6 Binary Search

Since the search algorithms work faster than the sorting algorithms, it was thought that it would not be a problem to implement the binary search algorithm as recursive. However, in this implementation, an input with a maximum size of 2000 could be examined without a stack overflow error. For this reason, the implementation was done iteratively.

Recursive implementation:

```java
public static int binarySearch(ArrayList<Integer> numbers, int value, int
    startIndex, int endIndex) {
     if (startIndex > endIndex)
         return -1; //not found

     int middle = (startIndex + endIndex) / 2;

     if (numbers.get(middle) == value)
         return middle;

     else if (numbers.get(middle) < value)
         return binarySearch(numbers, value, startIndex, middle);

     else
         return binarySearch(numbers, value, middle + 1, endIndex);
}
```

Iterative implementation:

```
public static int binarySearch(int[] numbers, int value) {
    int low = 0;
    int high = numbers.length - 1;

    while (high - low > 0) {
        int mid = (high + low) / 2;

        if (numbers[mid] == value)
            return mid;

        else if (numbers[mid] < value)
            low = mid + 1;

        else
            high = mid - 1;
    }
    return - 1; //not found
}
```

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	5.8081	6.3578	17.5443	56.3135	165.0664	711.9575	2489.0079	9913.9483	39209.3839	150325.5839
Quick sort	1.7688	0.9488	1.7171	2.9139	5.0493	10.2465	33.5961	120.5374	356.7313	637.2471
Bucket sort	7.1318	4.4098	4.9323	8.3540	17.7854	37.5361	49.2697	85.4101	192.8901	402.048
	Sorted Input Data Timing Results in ms									
Selection sort	0.6992	2.505	9.8444	39.0987	156.4625	628.3865	2654.3276	10971.3018	39220.7147	148658.1369
Quick sort	1.1266	4.3047	16.9648	67.3738	270.7145	1074.3731	4374.6587	18556.0779	84160.7573	329834.2146
Bucket sort	1.0891	1.1709	1.9678	3.8819	6.5671	5.2028	6.8344	16.3601	32.0814	67.4966
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	6.1178	11.0728	44.1491	176.0982	714.1666	2771.6797	11021.7889	44564.6956	177494.7704	673819.6634
Quick sort	303.9925	7.5918	36.3178	60.1538	195.7556	778.5892	2996.2359	11639.8575	46667.7646	16783.4020
Bucket sort	327.0643	11.3316	8.2724	13.0614	22.5284	21.9219	33.6723	34.9058	75.9896	143.4405

It can be easily seen that bucket sort is the fastest among sorting algorithms.

7.

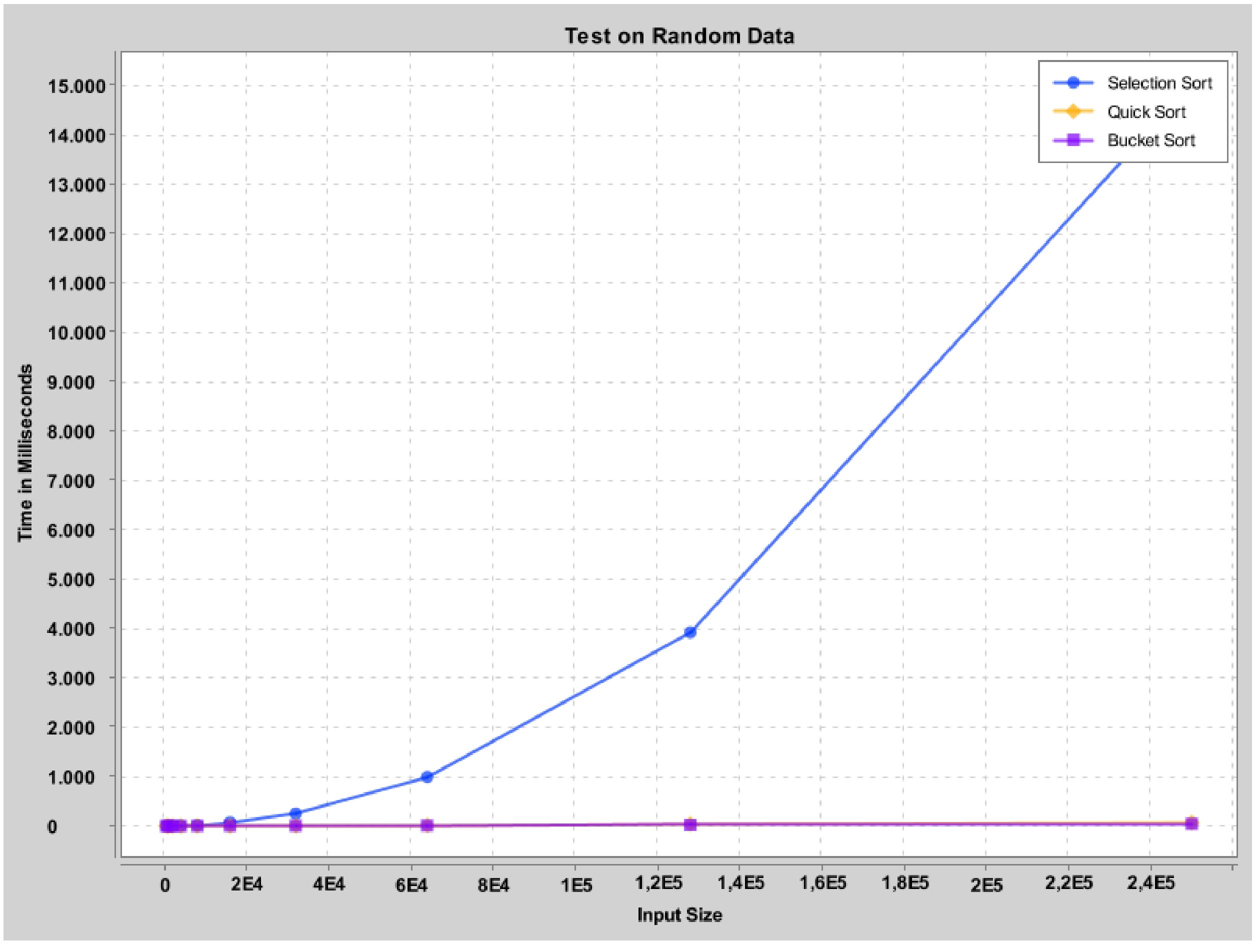

Figure 1: these values were obtained by running the sorting algorithms ten times on different input sizes with randomly sorted data and averaging.

- While the speed of quick sort and bucket sort is almost the same in random data, selection sort is noticeably slower as the input size grows.
- It is known that the theoretical asymptotic complexity for selection sort is $O(n^2)$. It can be interpreted that the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy via looking at the plot.
- Because the complexity of Selection Sort is huge compared to Bucket Sort and Quick Sort, a plot with all three comes out like Bucket Sort and Quick Sort constant. However, both are known to have no constants in complexity, even in the best case. In order to make a more accurate analysis, graphs of both were drawn.
7.

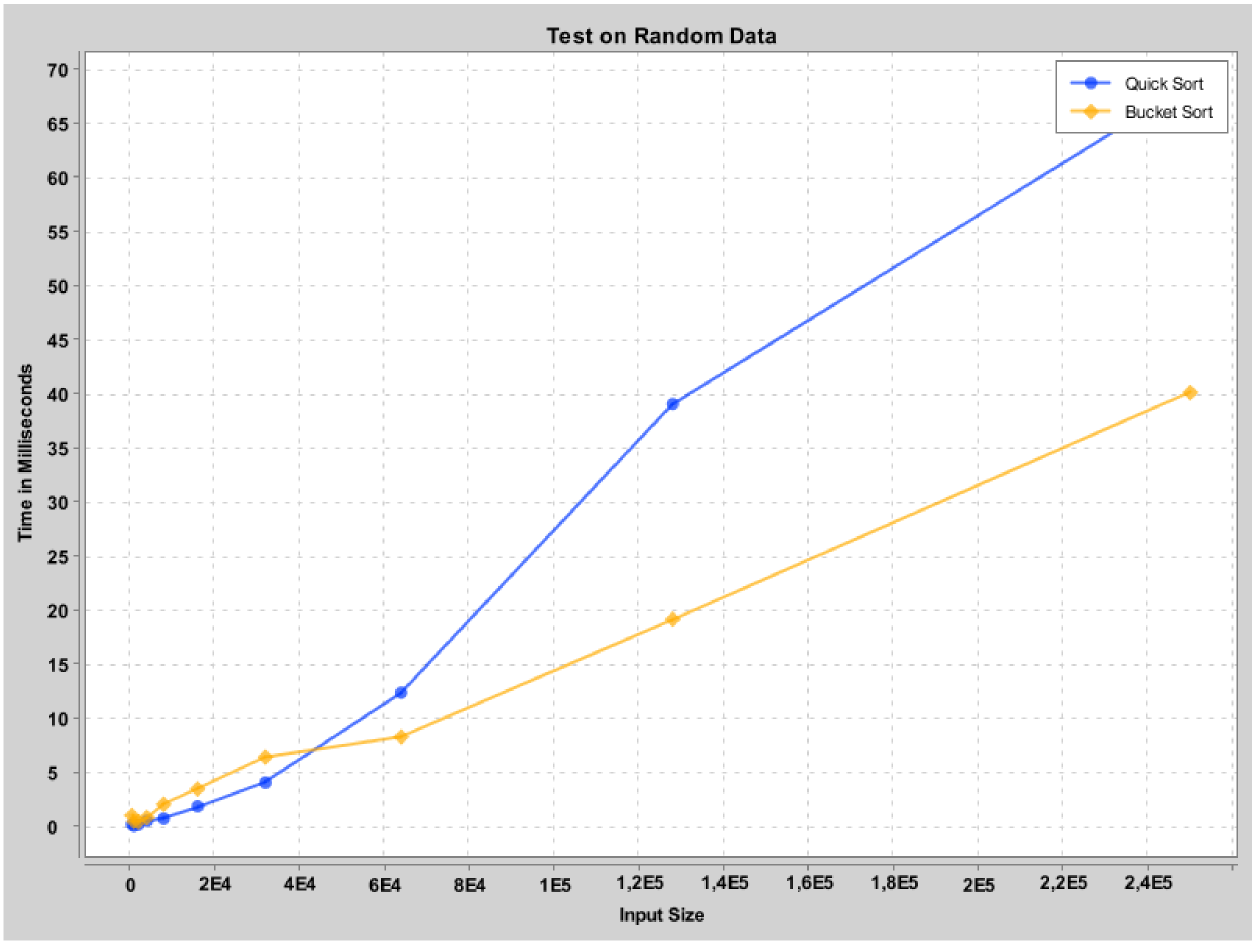

Figure 2: these values were obtained by running the search algorithms a thousand times on different input sizes and averaging them.

- It is known that Quick Sort's best case occurs when the partitions are as evenly balanced as possible. The complexity for quick sort -in the random data case- can be said to be $O(nlogn)$, as the best case almost provides. Considering that the complexity of Bucket Sort is also nearly (it is something between $O(n)$ and $O(n^2)$) $O(n)$ (hash function prevents worst case from happening), it can be said that the running times of the implementations of Bucket Sort and Quick Sort are also consistent with the theoretical asymptotic complexitiy in random data.
NOTE: The graph did not come out straight because the complexity of Quick Sort is not exactly $O(nlogn)$ and the complexity of Bucket Sort is not exactly $O(n)$. Even in small inputs, quick sort can be faster than bucket sort.

7.

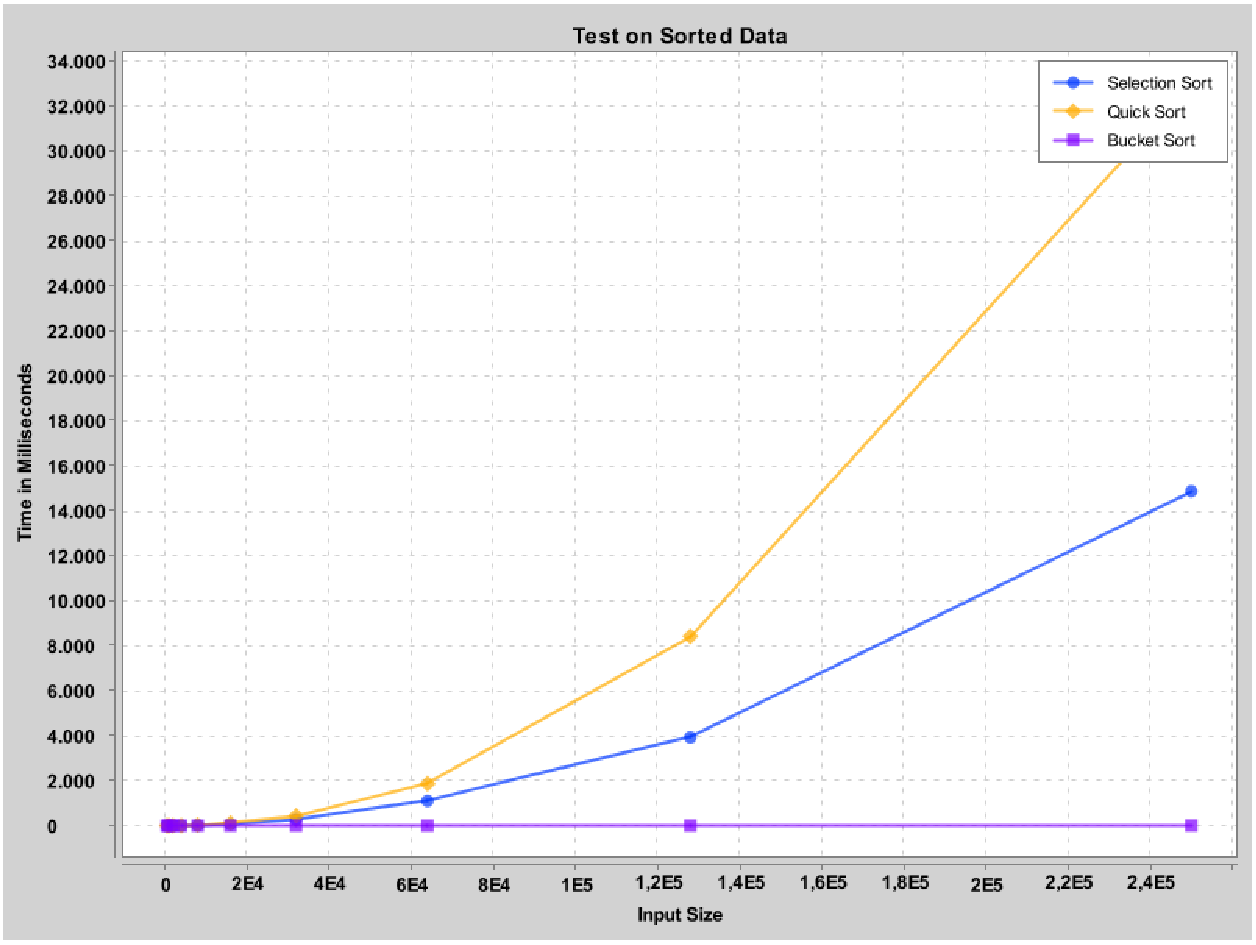

Figure 3: these values were obtained by running the sorting algorithms ten times on different input sizes with sorted data and averaging.

- In sorted data, the speed of selection sort and quick sort are close to each other and they are increasing gradually as the input size grows, while bucket sort remains very fast compared to them.
- It is known that the theoretical asymptotic complexity for selection sort is $O(n^2)$. It can be interpreted that the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy via looking at the plot.
- The fact that the input data is already sorted prevents the partitions in Quick Sort from being balanced.By looking at the plot that came out of the experiment in random data, it can be seen that this affects the complexity of Quick Sort. It's pretty close to the Selection Sort, which is $O(n^2)$ compared to the previous case. So the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy.
- Since hash function prevents worst case from happening, Bucket Sort has $O(n)$ complexity which

is pretty smaller than the complexities of other sorting algorthims. This explains why it looks almost constant on the plot. So the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy.
- In order to make a clearer interpretation, a single plot of the bucket sort has been drawn:
 7.

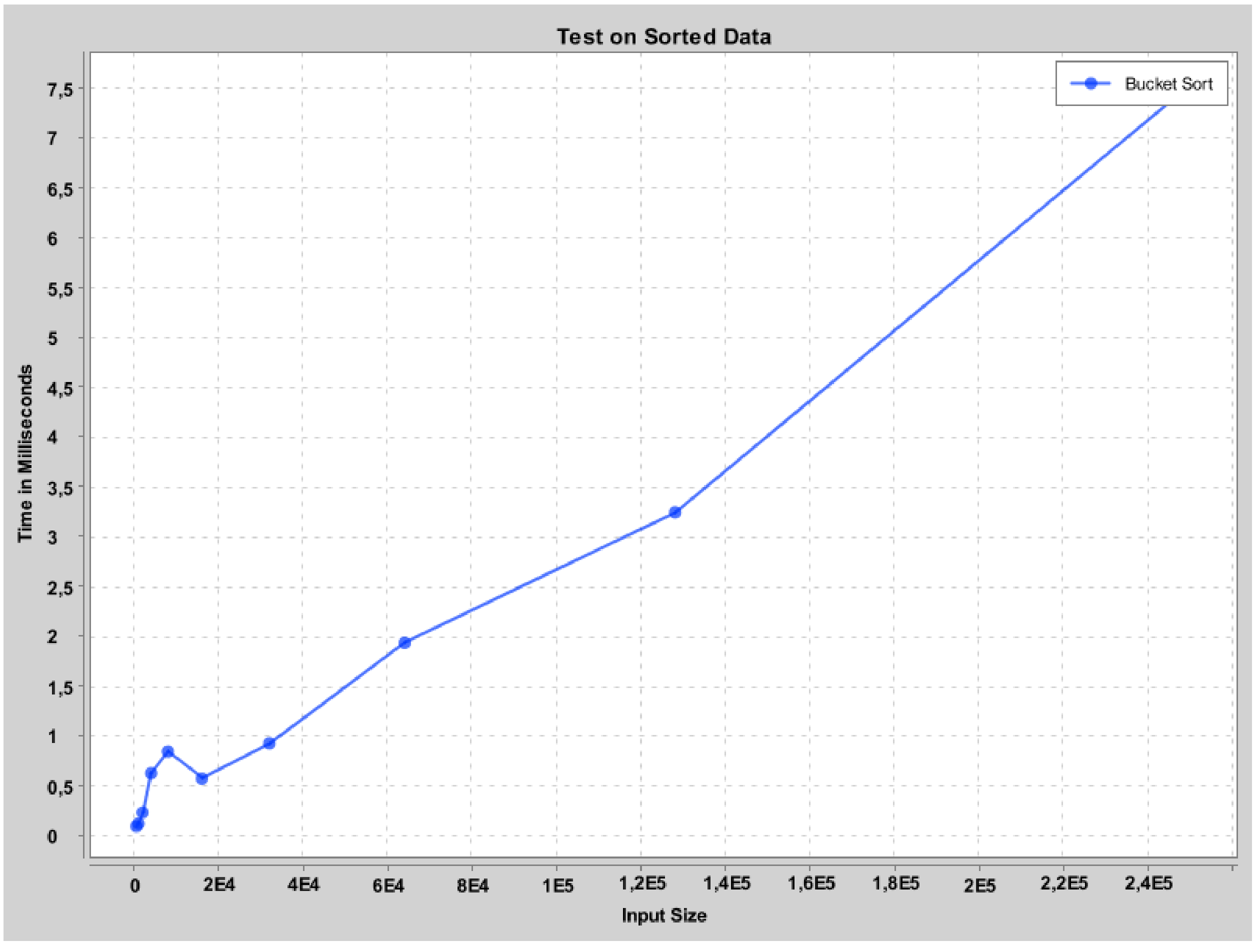

Figure 4: these values were obtained by running the sorting algorithms ten times on different input sizes with sorted data and averaging.

7.

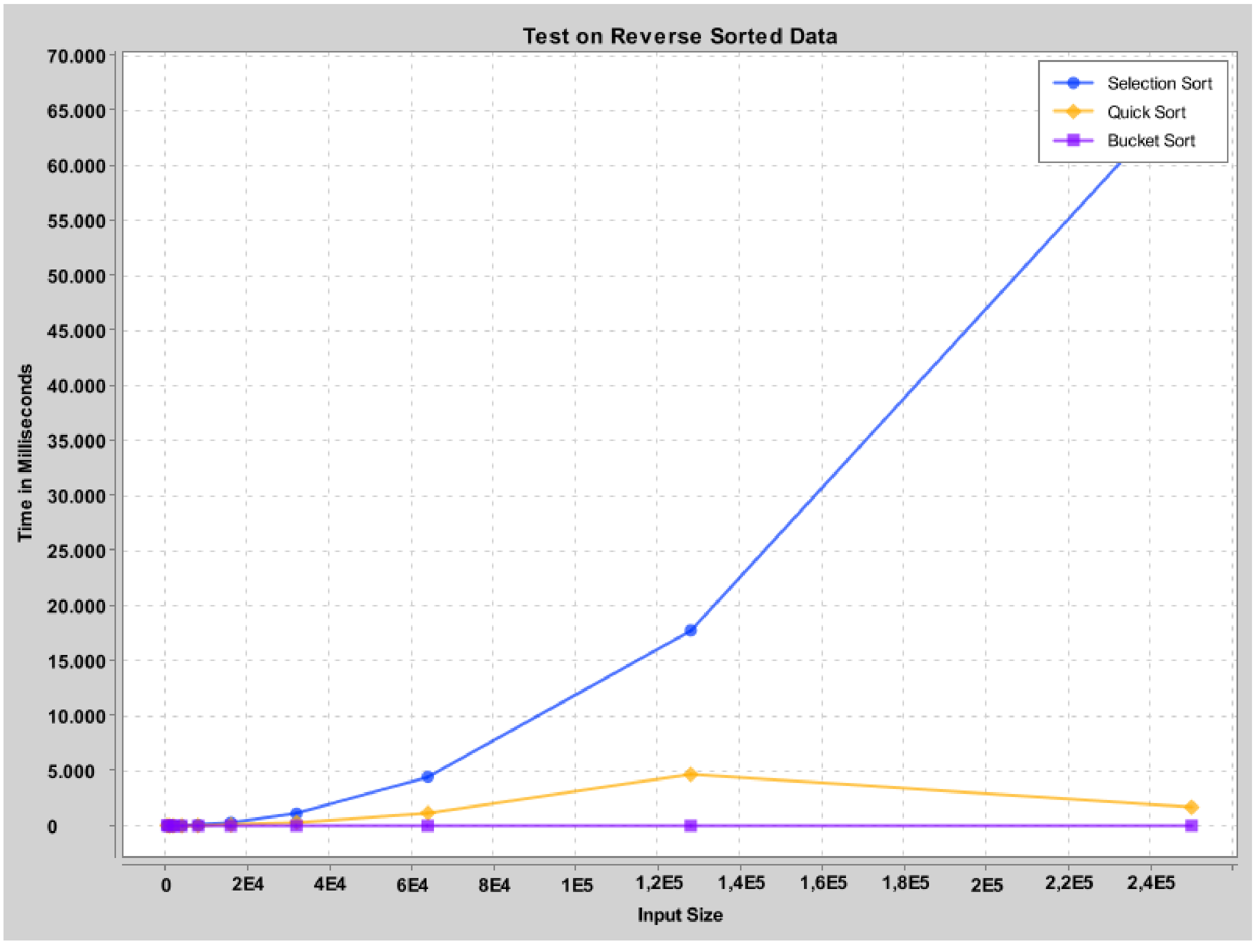

Figure 5: these values were obtained by running the sorting algorithms ten times on different input sizes with reverse sorted data and averaging.

- In reverse sorted data, quick sort performs close to bucket sort while selection sort is slow.
- It is known that the theoretical asymptotic complexity for selection sort is $O(n^2)$. It can be interpreted that the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy via looking at the plot.
- Like the situation described for comparing sorted data and random data; Since the partition cannot be balanced in reverse sorted, it is closer to selection sort ($O(n^2)$).
NOTE: However, I couldn't understand why quick sort is faster in reverse sorted data than sorted data. We can say that the input sorted creates the worst case for quick sort. I don't understand why it's in ascending or descending order affects performance. I think that this result is inconsistent with the theoretical knowledge. If there is a logical explanation for this situation, I would be very grateful if you could get back to me.

- Since hash function prevents worst case from happening, Bucket Sort has $O(n)$ complexity which is pretty smaller than the complexities of other sorting algorthims. This explains why it looks almost constant on the plot. So the running time of the algorithm implementation is consistent with the theoretical asymptotic complexitiy.
- In order to make a clearer interpretation, a single plot of the bucket sort has been drawn:
7.

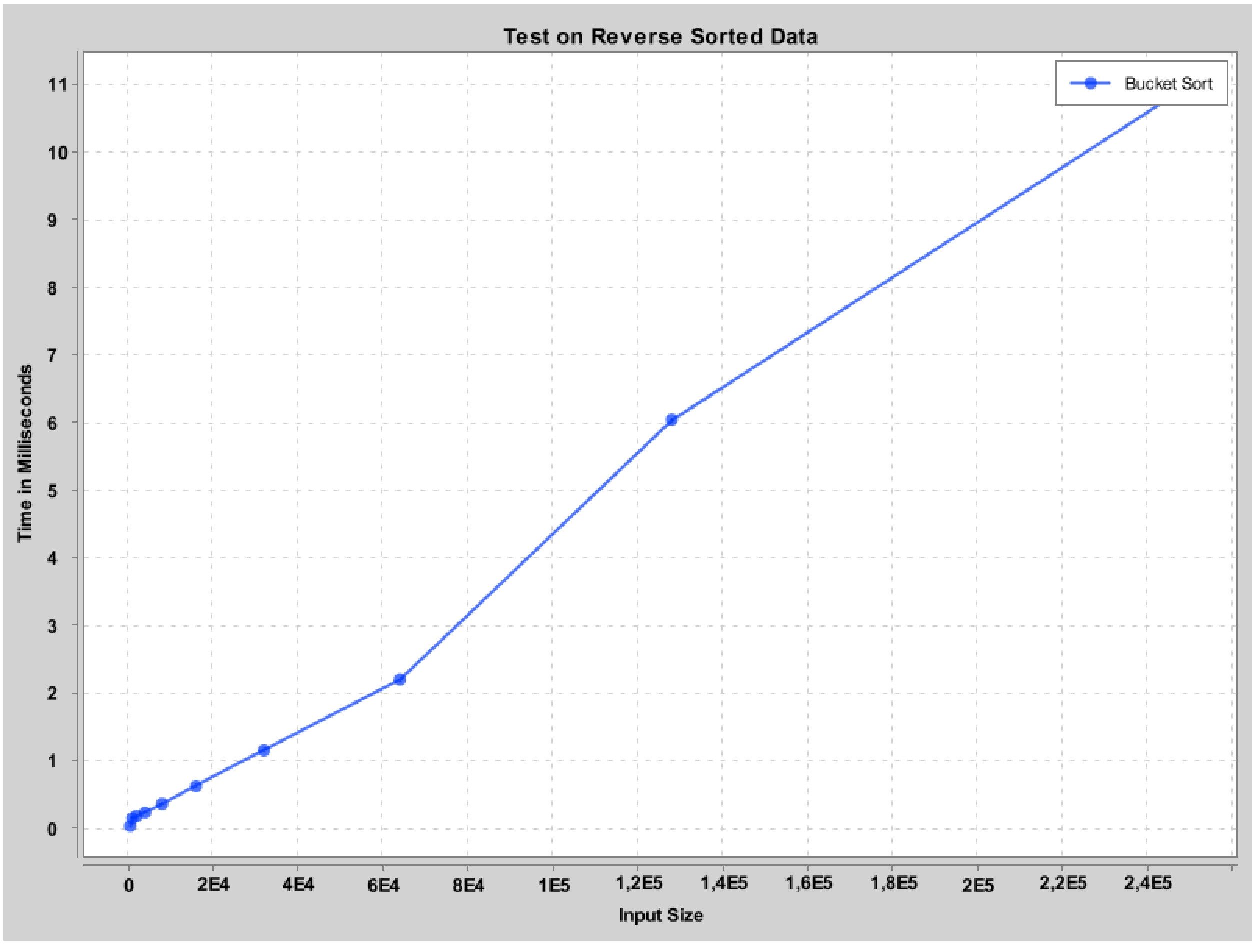

Figure 6: these values were obtained by running the sorting algorithms ten times on different input sizes with sorted data and averaging.

By examining all plots, the following conclusions can be reached:

i) Although selection sort has the same time complexity in all three cases, it works slower on reverse sorted data than others.

i) Quick sort works fastest on random data and the slowest on sorted data.

i) In these experiments, the contents of the inputs do not change. Only the way they are lined up changes. The important thing that changes complexity in Bucket Sort is the number of collisions. It would be wrong to say that the order has a huge impact on the speed of bucket sort, since the input does not change as content. However; if we had to make a comment, we can say that Bucket Sort works slowest on random data, the fastest on sorted data (with a negligible difference compared to reverse sorted data) by examining the plots.

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)	1169500	2287800	213800	346600	610500	1079400	2123200	4239600	1.8093000	1.43175E7
Linear search (sorted data)	51900	96000	131200	306100	476400	1081400	2716800	5901500	1.33014E7	2.69452E7
Binary search (sorted data)	380400	200200	165200	204400	320500	210900	218900	272900	400300	611500

7.

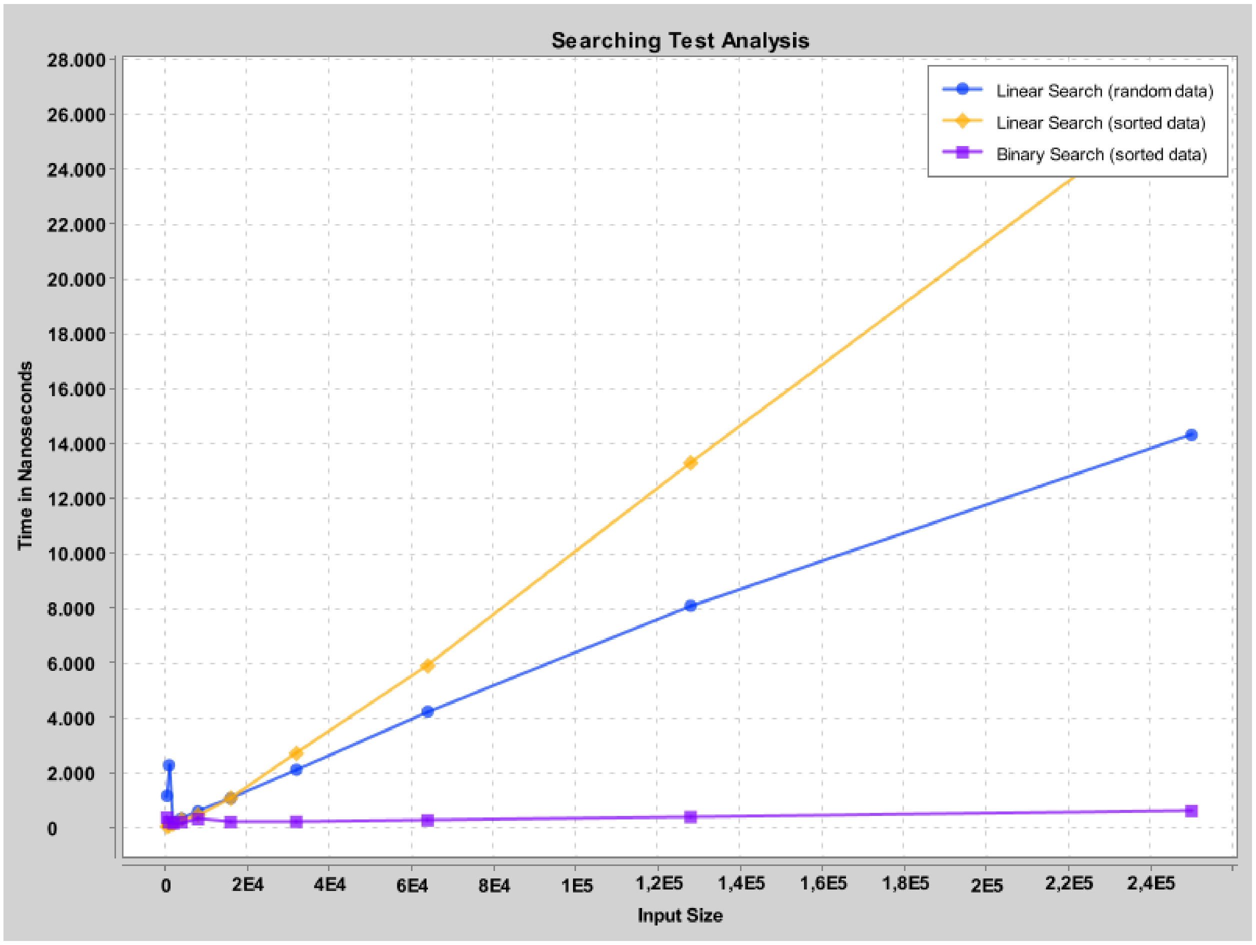

Figure 7: these values were obtained by running the search algorithms a thousand times on different input sizes and averaging them.

It can be easily interpreted that binary search is noticeably faster than linear search in searching algorithms.
- Although searching in random data in linear search starts slower than sorted data, the search performed in random data has a better performance in large input sizes.
- Since Binary Search's time complexity is $O(1)$ in the best case and $O(logn)$ in the worst case, we should expect the time to come out in this range. Likewise, since Linear Search's time complexity is $O(1)$ in the best case and $O(n)$ in the worst case, the duration should be within the given range. We can say that the plot provides for these intervals. For this reason, the result of the experiment and the theoretical information for the search algorithms were consistent.

Complexity analysis tables (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	$O(n)$	$O(n+k)$	$O(n^2 logn)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(logn)$	$O(logn)$

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	$O(n + size)$
Linear Search	$O(1)$
Binary Search	$O(1)$ (iterative implementation)

4 Notes About Execution

Assignment worked fine on my computer (IntelliJ IDEA). But no matter how hard I tried, I couldn't run the .jar file in terminal. As I understood from the conversations in Piazza that this would not cause a problem, I decided to load the assignment in this way. I did not take the input file as a command line argument because as I mentioned begore I did not run it in the terminal.

5 Notes About Implementation

1- ArrayList<Integer> used because it was tempting to easily insert and dynamically resize while reading the data. However, the code had been changed because it is realized that the performance was almost 10 times better if int[] preferred.

2- The quick sort implementation was originally written using stack as the data structure. Afterwards, an array with a stack-like function was tried for performance reasons. It was noticed that it had a slight effect on performance.

3- Although the quick sort algorithm can be written and read more easily as recursive, implementation was written iteratively to avoid "stack overflow error" in large data sizes.

6 Notes About Computational Complexity

1- Selection Sort:
Time complexity equals $O(n^2)$ in best, average, and worst case. That means it is independent of distribution of data.

2- Quick Search:
i) Best case: When the partitions are as evenly balanced as possible.
ii) Worst case: When quicksort always has the most unbalanced partitions possible.

3- Bucket Sort:
i) Best case: The best-case scenario would be when each element in the input array falls into a different bucket. In this case, the time complexity of the algorithm would be O(n) since there would be no need for sorting the individual buckets, and the elements could be directly appended to the output array in the order of the buckets.
ii) Average case: The average-case time complexity of the bucket sort algorithm is O(n + k), where k is the number of buckets used. However, this analysis assumes that the hash function evenly distributes the elements among the buckets. If the hash function does not distribute the elements evenly, then the performance of the algorithm can deteriorate.
iii) Worst case: The worst case occurs when all the elements in the input array fall into the same bucket. The worst-case time complexity of the Collections.sort() method is O(n log n). Therefore, the worst-case time complexity of the bucket sort algorithm would be O(n^2 log n).

4- Linear Search:
i) Best case: The best case will take place if the element to be search is on the first index.
ii) Worst case: The worst case will take place if the element to be search is in the last index or the element to be search is not present in the list.

5- Binary Search:

i) Best case: The best case of Binary Search occurs when the element to be search is in the middle of the list.

ii) Worst case: The worst case of Binary Search occurs when the element is to search is in the first index or last index.

7 Notes About Auxiliary Space Complexity

1- Selection Sort

The implementation on the second page does not use any extra space for data structures such as arrays, lists, or stacks. Instead, it performs the sorting in place by swapping elements of the input array. Therefore, the auxiliary space complexity of this implementation of Selection Sort is O(1), constant space complexity.

2- Quick Sort

In the implementation on the third page, line 45 (int stackSize = high - low + 1;) and line 46 (int[] stack = new int[stackSize];) shows additional memory space. This lines initializes an array called stack with a size equal to the difference between the highest index high and the lowest index low in the input array, plus one. This array is used as a stack to keep track of the indices that still need to be sorted, and to do this, the algorithm pushes the indices onto the stack and pops them off as they get sorted. Since the size of the stack is proportional to the size of the input array, the auxiliary space complexity of this implementation of QuickSort is O(n).

3- Bucket Sort:

In the implementation on the fourth page, line 69 (List¡Integer¿[] buckets = new List[size]) shows additional memory space. This line declares an array of size ArrayLists, which is used as the data structure to store the numbers during the sorting process. The space complexity of this data structure is proportional to the size of the input array, which makes the auxiliary space complexity of the algorithm O(n).

4- Linear Search:

In linear search, we are creating an integer variable to store the index of the data we are looking for (if exist, if not store -1). In the implementation on the fifth page, the ”int i” variable on line 102 shows additional memory space.

5- Binary Search:

In an iterative implementation of binary search, the space complexity will be O(1). This is because we need two variable to keep track of the range of elements that are to be checked. No other data is needed. In the implementation on the sixth page, the variable ”int low” on line 125 and the variable ”int high” on line 126 show additional memory space.

In a recursive implementation of Binary Search, the space complexity will be O(logN). This is because in the worst case, there will be logN recursive calls and all these recursive calls will be stacked in memory.

8 References

- The assignment pdf you shared for the assignment from Piazza.
- https://iq.opengenus.org/
- https://www.geeksforgeeks.org/
- https://www.youtube.com/@maxodidily
- https://www.youtube.com/@MichaelSambol
- https://www.youtube.com/@CodingWithJohn
- https://stackoverflow.com/
- https://www.khanacademy.org