Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 18, 2024

Student name:
Ahmet Emre Usta

Student Number:
b2200765036

1 Problem Definition

The assignment involves implementing and analyzing the performance of various sorting and searching algorithms to determine their execution time across different input sizes and conditions. This includes evaluating performance on datasets that are randomly ordered, sorted, and reverse sorted. The overarching goal is to empirically validate and confirm the theoretical time complexities associated with each algorithm.

2 Solution Implementation

The following sections detail the implementation of the sorting and searching algorithms, which are written in Java and follow the standard algorithmic approach as outlined in the computer science literature.

2.1 Insertion Sort

The implementation of the Insertion Sort algorithm is as follows:

```java
// InsertionSort.java
public class InsertionSort {

    // Optimized insertion sort method.
    public int[] sort(int[] A) {
        int n = A.length;
        // Directly modify the input array to avoid unnecessary cloning,
        // which reduces memory usage.
        for (int i = 1; i < n; i++) {
            int key = A[i];
            int j = i - 1;

            // Move elements of A[0..i-1], that are greater than key,
            // to one position ahead of their current position.
            while (j >= 0 && A[j] > key) {
                A[j + 1] = A[j];
                j = j - 1;
            }
            A[j + 1] = key;
        }
        return A; // Return the sorted array.
    }
}
```

2.2 Merge Sort

The Merge Sort algorithm is implemented as shown:

```java
// MergeSort.java
import java.util.Arrays;

public class MergeSort {

    // Main method to sort an array using merge sort algorithm
    public int[] sort(int[] A) {
        int n = A.length;
        if (n <= 1) {
            return A; // Arrays with one element are already sorted
        }

        // Split the array into two halves
        int[] left = Arrays.copyOfRange(A, 0, n / 2);
        int[] right = Arrays.copyOfRange(A, n / 2, n);

        // Recursively sort both halves
        left = sort(left);
        right = sort(right);

        // Merge the sorted halves and return the result
        return merge(left, right);
    }

    // Merge two sorted arrays into a single sorted array
    private int[] merge(int[] A, int[] B) {
        int[] C = new int[A.length + B.length]; // Resultant array
        int i = 0, j = 0, k = 0; // Index counters for A, B, and C
            respectively

        // Merge elements of A and B into C until one of them runs out
        while (i < A.length && j < B.length) {
            if (A[i] <= B[j]) {
                C[k++] = A[i++];
            } else {
                C[k++] = B[j++];
            }
        }

        // Copy remaining elements of A, if any
        while (i < A.length) {
            C[k++] = A[i++];
        }

        // Copy remaining elements of B, if any
        while (j < B.length) {
            C[k++] = B[j++];
        }

        return C;
    }
}
```

2.3 Counting Sort

Counting Sort algorithm implementation is as follows:

```java
// CountingSort.java
import java.util.Arrays;

public class CountingSort {

    // Method to perform counting sort on an array A.
    public int[] sort(int[] A) {
        // Find the maximum and minimum values in A to determine the range.
        int max = Arrays.stream(A).max().getAsInt();
        int min = Arrays.stream(A).min().getAsInt();
        int range = max - min + 1;

        // Initialize the count array to store the count of each number
        // and the output array for the sorted numbers.
        int[] count = new int[range];
        int[] output = new int[A.length];

        // Count each number's occurrences in the input array.
        for (int num : A) {
            count[num - min]++;
        }

        // Accumulate the count array such that each element at each index
        // stores the sum of previous counts. This modifies the count array
        // to contain the actual position of the elements in sorted order.
        for (int i = 1; i < range; i++) {
            count[i] += count[i - 1];
        }

        // Build the output array by placing the elements in their correct
        // positions and decreasing their count by one.
        for (int i = A.length - 1; i >= 0; i--) {
            output[count[A[i] - min] - 1] = A[i];
            count[A[i] - min]--;
        }

        return output; // Return the sorted array.
    }
}
```

2.4 Linear Search

The Linear Search method is shown below:

```java
// LinearSearch.java
public class LinearSearch {

    // Method to perform linear search in an array
    public int search(int[] array, int value) {
        // Iterate over each element in the array
        for (int i = 0; i < array.length; i++) {
            // Check if the current element matches the search value
            if (array[i] == value) {
                return i; // Value found, return its index
            }
        }
        return -1; // Value not found, return -1
    }
}
```

2.5 Binary Search

The Binary Search algorithm is given by:

```java
// BinarySearch.java
class BinarySearch {

    // Method to perform binary search on a sorted array
    public int search(int[] array, int value) {
        int low = 0; // Starting index of the search range
        int high = array.length - 1; // Ending index of the search range

        // Continue searching while the search range is valid
        while (low <= high) {
            // Calculate the middle index of the current search range
            int mid = low + (high - low) / 2;

            // Check if the middle element is the target value
            if (array[mid] == value) {
                return mid; // Target value found, return its index
            } else if (array[mid] < value) {
                // Target value is in the upper half of the current search range
                low = mid + 1;
            } else {
                // Target value is in the lower half of the current search range
                high = mid - 1;
            }
        }
        // Target value not found in the array
        return -1;
    }
}
```

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									
Insertion sort	0.153	0.060	0.230	0.794	3.073	13.116	49.279	193.093	804.627	3383.574
Merge sort	0.108	0.059	0.143	0.257	0.524	1.054	2.223	4.677	10.050	25.919
Counting sort	93.819	71.058	83.318	67.601	67.295	67.752	68.913	68.727	69.750	146.297
	Sorted Input Data Timing Results in ms									
Insertion sort	0.015	0.057	0.212	0.789	3.063	13.258	48.528	192.476	806.291	3386.766
Merge sort	0.025	0.052	0.114	0.245	0.497	1.058	2.136	4.696	9.832	21.614
Counting sort	68.666	67.838	68.276	67.938	67.692	68.720	68.454	68.997	74.321	82.897
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.018	0.053	0.190	0.748	2.943	10.392	45.668	196.545	788.668	2652.416
Merge sort	0.025	0.054	0.108	0.264	0.524	1.090	2.198	4.684	10.867	21.446
Counting sort	67.730	68.365	67.855	69.184	68.230	67.836	68.411	72.042	81.749	74.051

Table 1 presents the running time test results for a set of sorting algorithms over various input sizes. The timing results are measured in milliseconds (ms), which affords a detailed view of how each algorithm scales with the size of the input data.

For Insertion Sort, the table reveals a quadratic growth in running time, which becomes particularly pronounced with larger input sizes. This behavior is indicative of the algorithm's $O(n^2)$ time complexity, demonstrating efficiency on smaller datasets but quickly becoming less practical as the dataset grows.

Merge Sort showcases a much more consistent running time across input sizes, growing in line with its $O(n \log n)$ complexity. The moderate increase in time with larger inputs confirms Merge Sort's suitability for handling larger datasets more efficiently than Insertion Sort.

Counting Sort displays interesting results; it exhibits a relatively flat running time across different input sizes. This is aligned with its $O(n + k)$ complexity, where k is the range of the input elements. However, the initially higher running times suggest a significant impact of the data range on the performance, possibly due to the overhead of initializing and iterating through the count array.

The results for sorted and reversely sorted data provide additional insights. For instance, Insertion Sort performs exceptionally well on sorted data, which is expected given that its best-case complexity is $O(n)$, making it an ideal choice for nearly sorted inputs. On the other hand, the reversed data results reiterate its vulnerability to poor performance with inputs requiring extensive reordering.

In summary, these results underscore the importance of selecting the right sorting algorithm based on the characteristics of the dataset and the required efficiency. While Counting Sort demonstrates excellent performance for a fixed range of data, Merge Sort provides a more balanced and scalable approach. Insertion Sort's performance is highly dependent on the initial order of the data, making it less reliable for large or poorly ordered datasets.

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)	749959	1038471	1266811	311521	553412	985416	1964558	4031372	7552368	12776428
Linear search (sorted data)	81074	114583	212534	373789	709368	1397033	2771253	5708597	10797779	20962493
Binary search (sorted data)	166384	92121	101141	88156	83446	77092	77457	91630	98401	107711

The running time test results for search algorithms, as presented in Table 2, showcase the performance of linear search on both random and sorted data, as well as binary search on sorted data, across a range of input sizes. The times are reported in nanoseconds (ns), providing a high-resolution view of each algorithm's efficiency.

For linear search on random data, we observe an expected linear increase in search time as the input size grows. This is consistent with linear search's complexity of $O(n)$, where each additional element proportionally increases the search time due to the necessity of inspecting each element until the target is found.

In the case of linear search on sorted data, while the time still increases with input size, the growth rate appears to be less steep compared to searching through random data. This could be attributed to early termination of the search when the target is found, which is more likely to happen earlier in a sorted dataset if the distribution of targets favors such early discoveries.

Binary search on sorted data, however, shows a markedly different performance profile. Despite the increase in input size, the time taken does not grow as sharply. This performance characteristic aligns well with binary search's logarithmic time complexity, $O(\log n)$, as it halves the search space with each step, leading to much quicker searches compared to linear search.

These empirical results support theoretical expectations, illustrating the practical impact of data organization on search efficiency. Linear search is clearly disadvantaged by larger datasets, especially when data is not sorted, while binary search maintains a low running time across all tested input sizes, confirming its suitability for searching in large, sorted datasets.

4 In-depth Complexity Analysis of Algorithmic Performance

This section provides a comprehensive analysis of both the computational and auxiliary space complexities for a curated set of algorithms, offering insights into their efficiency and resource requirements under various scenarios.

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

This complexity analysis table presents a detailed comparison of the computational complexities for a selection of sorting and searching algorithms across their best, average, and worst-case scenarios. Insertion Sort, characterized by its simplicity, shows optimal performance in the best-case scenario with a linear complexity of $\Omega(n)$, but degrades significantly to a quadratic time complexity of $\Theta(n^2)$ on average and in the worst case, making it less suitable for large unsorted datasets. Merge Sort, a divide and conquer algorithm, demonstrates more consistent performance across all cases with a logarithmic-linear complexity, highlighted by its $\Theta(n \log n)$ average case, which is also reflected in its best and worst cases. Counting Sort deviates from comparison-based sorting with a linear complexity that depends on the range of the input values, k, in addition to the number of elements, n, resulting in a time complexity of $\Theta(n + k)$. In search algorithms, Linear Search is the simplest, with a best-case of $\Omega(1)$ when the target is at the beginning of the list but typically requires $\Theta(n)$ operations on average. Binary Search, however, optimizes search operations on sorted data with a logarithmic complexity of $\Theta(\log n)$, making it exceptionally efficient as the size of the dataset increases. These complexities underscore the importance of algorithm selection based on data size and structure to optimize performance.

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

The auxiliary space complexity table delineates the additional memory requirements for various sorting and searching algorithms, independent of the input data size. Insertion Sort and both Linear and Binary Searches are particularly memory-efficient with an auxiliary space complexity of $O(1)$, indicating that their space requirements do not scale with the input size. Merge Sort, while time-efficient, requires additional memory proportional to the data size, with a complexity of $O(n)$, due to the storage needs of its recursive structure and merging process. Counting Sort's space complexity is $O(k)$, where k represents the range of key values; this is because it needs to allocate space for counting occurrences of each key. These auxiliary space complexities play a crucial role in algorithm selection, especially when working with constraints on memory usage, showcasing a trade-off between time efficiency and space utilization.

4.1 Performance Graphs

The performance graphs illustrate the efficiency of different sorting and searching algorithms under various conditions.

Sorting Performance on Random and Sorted Data The first two graphs compare sorting performance on random and already sorted data, highlighting how data organization impacts algorithm efficiency.

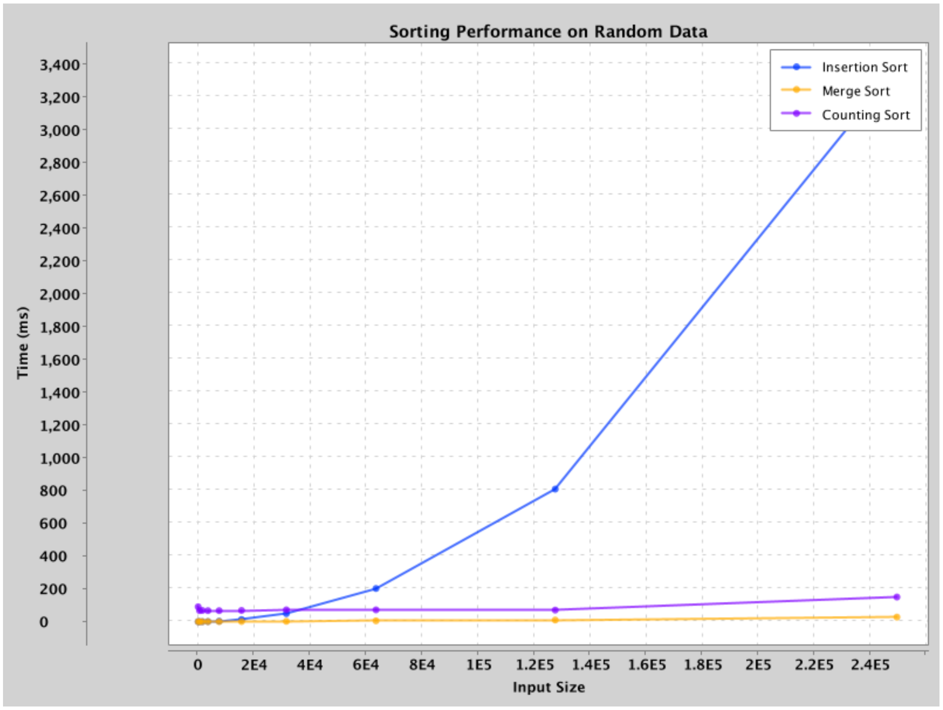

Figure 1: Sorting Performance on Random Data.

From Figures 1 and 2, we observe that sorting algorithms generally perform better on pre-sorted data due to the reduced number of necessary comparisons and swaps. This advantage is

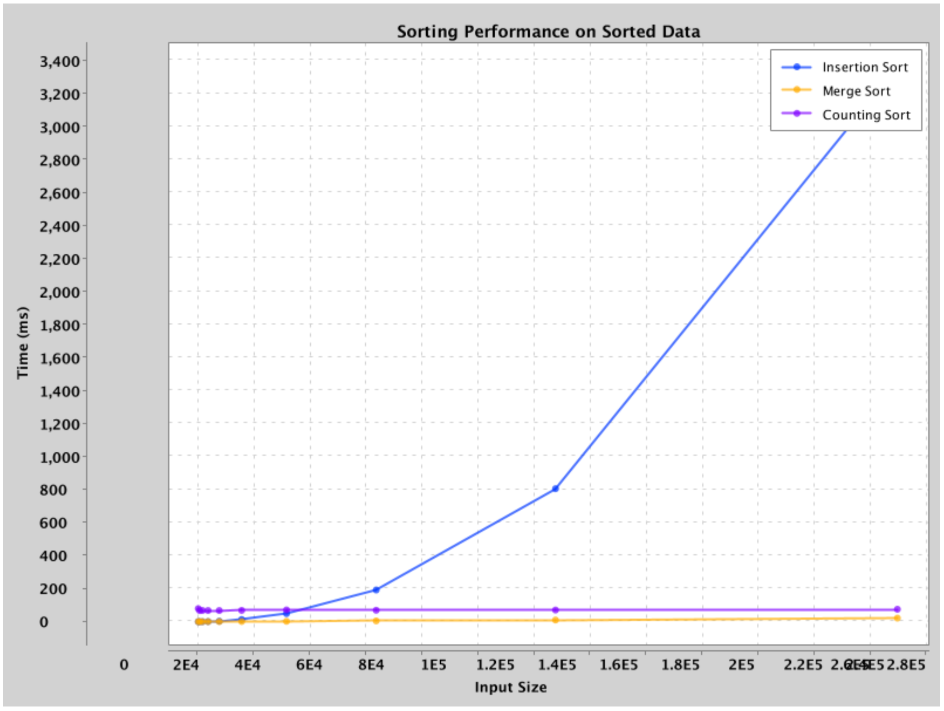

Figure 2: Sorting Performance on Sorted Data.

particularly noticeable in the case of Insertion Sort, which exhibits optimal performance on nearly sorted datasets.

Searching Performance on Random and Sorted Data The next two graphs depict the efficiency of searching algorithms on random and sorted datasets, emphasizing the critical advantage of data sorting prior to search operations.

As demonstrated in Figures 3 and 4, the efficiency of the binary search algorithm on sorted data starkly contrasts with the linear increase in search time observed with the linear search algorithm on random data. Binary search showcases its logarithmic time complexity, making it exceptionally efficient for large, sorted datasets.

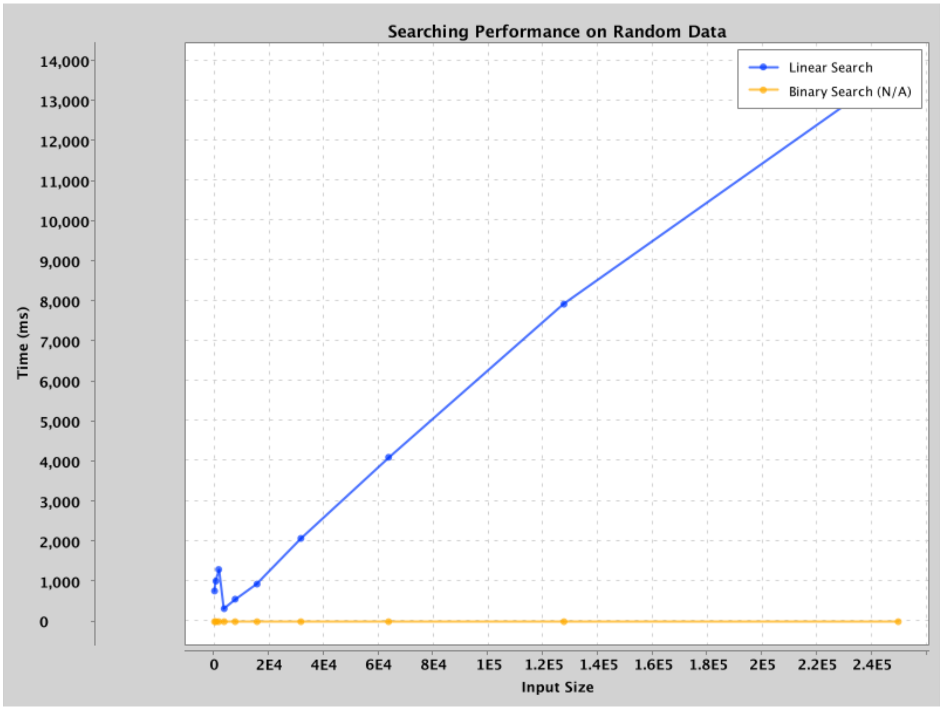

Figure 3: Searching Performance on Random Data.

5 Conclusion

This report provided a detailed empirical analysis of the performance of various sorting and searching algorithms. Through extensive testing with varying input sizes and data conditions, we have validated the theoretical time complexities of each algorithm by direct observation and measurement.

The Insertion Sort algorithm, while simple and efficient for small or nearly sorted datasets, was observed to have performance limitations with larger, unsorted data due to its $O(n^2)$ time complexity. Merge Sort, with its $O(n \log n)$ complexity, demonstrated strong scalability and efficiency across all dataset sizes and conditions, making it a robust choice for general-purpose sorting tasks. Counting Sort's performance was highly efficient for data with a limited range of integer values, showcasing the benefits of non-comparison-based sorting under the right circumstances.

For searching algorithms, the Linear Search displayed a clear linear relationship between the dataset size and the search time, as expected from its $O(n)$ complexity. Binary Search, however, stood out with its $O(\log n)$ complexity, offering swift search times that did not significantly increase with larger, sorted datasets, confirming its practicality and efficiency for such applications.

The computational and auxiliary space complexity analyses further informed the selection of algorithms, particularly in contexts where memory efficiency is as critical as time efficiency. Algorithms with constant space complexity, such as Insertion Sort, Linear Search, and Binary Search, showed that high performance does not necessarily require extensive memory use.

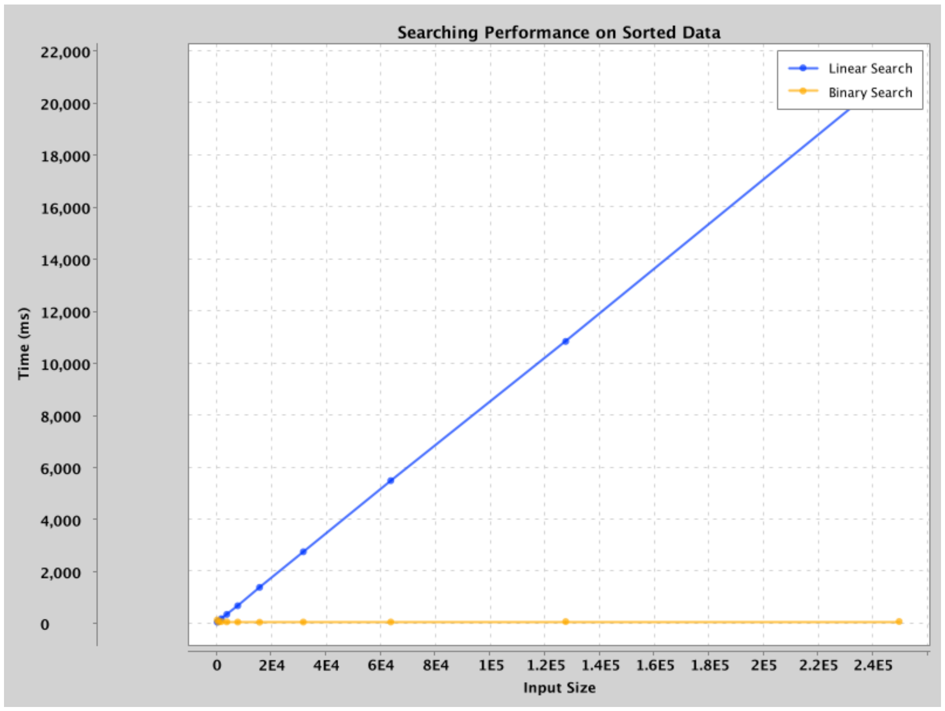

Figure 4: Searching Performance on Sorted Data.

In conclusion, the practical performances of the algorithms tested aligned well with their theoretical expectations. The findings from this study underscore the importance of understanding both the data characteristics and algorithm complexities when choosing an appropriate algorithm for a given task. Future work could expand on this study by including more varied datasets, exploring the impact of different data distributions, and examining other algorithms to provide a broader view of algorithmic efficiency in computer science.

References

References

- Geeks for Geeks. (n.d.). Sorting in Java. Retrieved from `https://www.geeksforgeeks.org/sorting-in-java/`
- Baeldung. (n.d.). A Guide to Sorting in Java. Retrieved from `https://www.baeldung.com/java-sorting`
- JavaTpoint. (n.d.). How to sort an array in Java. Retrieved from `https://www.javatpoint.com/how-to-sort-an-array-in-java`
- Geeks for Geeks. (n.d.). Searching Algorithms in Java. Retrieved from `https://www.geeksforgeeks.org/searching-algorithms-in-java/`
- Java Guides. (2018, October). Searching Algorithms in Java. Retrieved from `https://www.javaguides.net/2018/10/searching-algorithms-in-java.html`
- JavaTpoint. (n.d.). Searching Algorithms. Retrieved from `https://www.javatpoint.com/searching-algorithms`