SORTING ALGORITHM

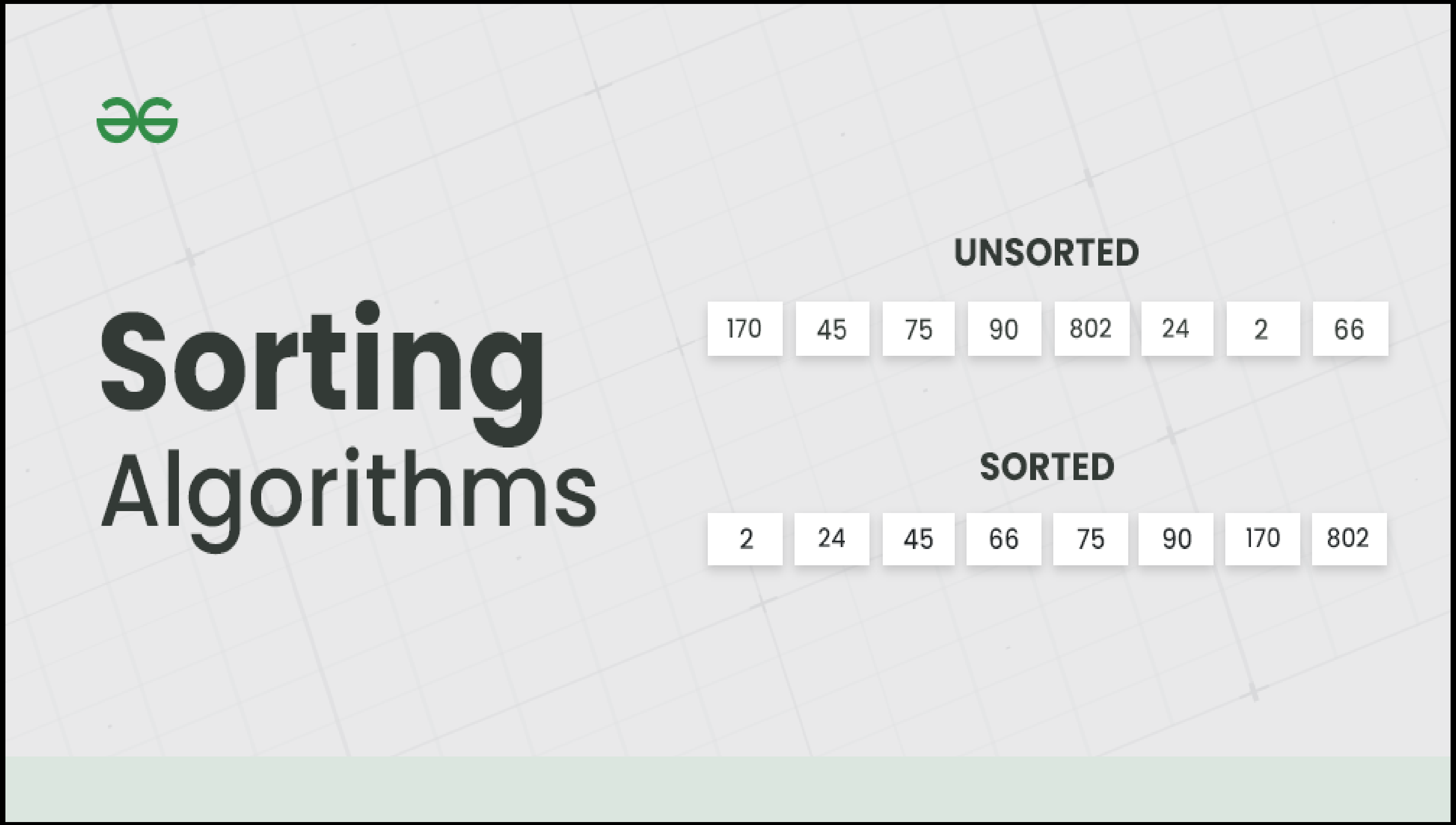
FACULTY OF ARTIFICIAL INTELLIGENCE

UNDER SUPERVISION :
DR: MOHAMED BEHIRY
ENG: Mohamed Towfik

Name:	Id:
Hesham Mohamed	225152
Youssef Emad	225241
Youssef Botros	225148
Mohamed Alshamy	225167
Omar Tokal	225238

Introduction

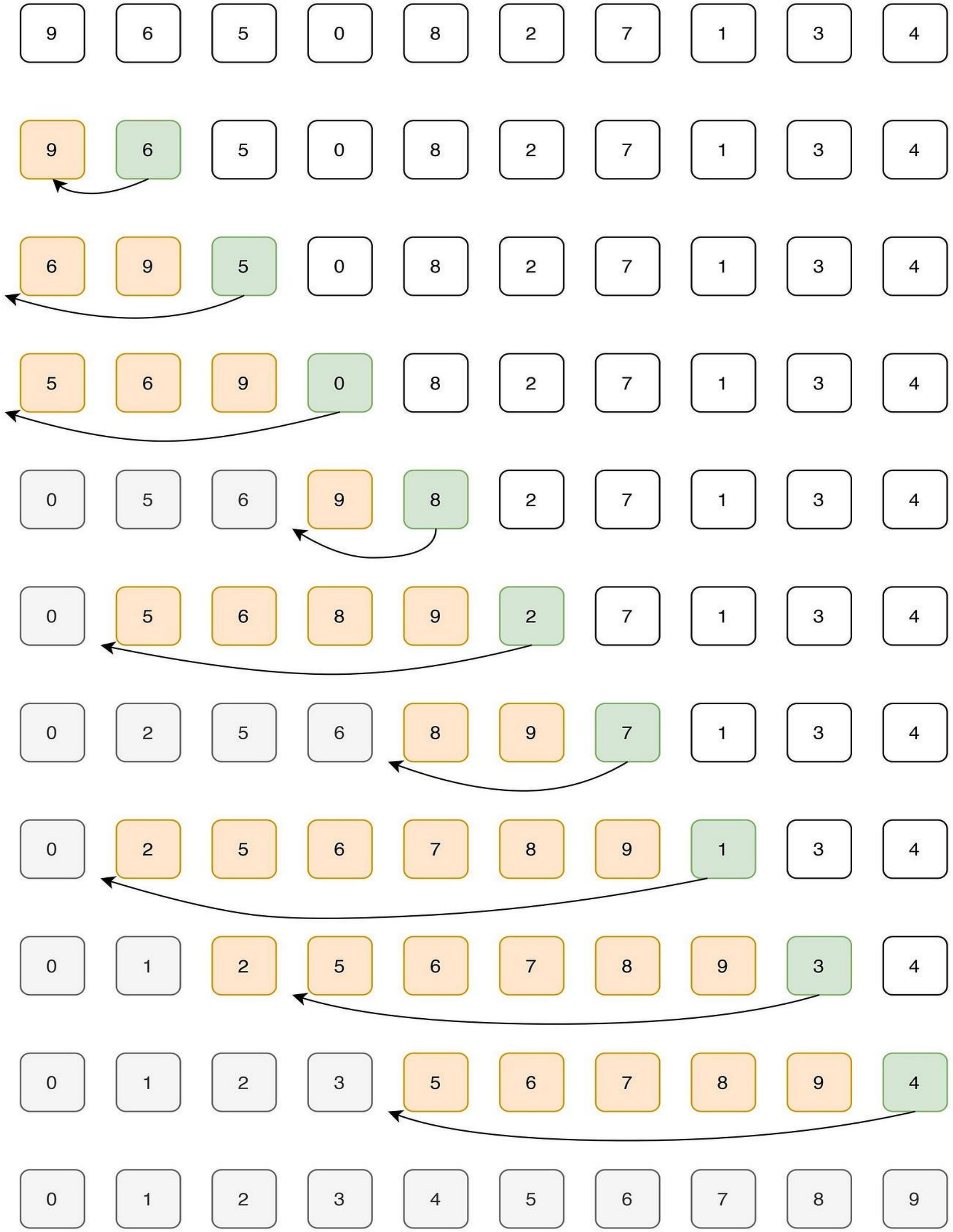
- Purpose: This document outlines the technical specifications for various sorting algorithms implemented in the system.
- Scope: It covers the algorithms' functionalities, inputs, outputs, time complexity, and space complexity.


Sorting
Algorithms
UNSORTED
170 45 75 90 802 24 2 66
SORTED
2 24 45 66 75 90 170 802

ALGORITHMS:

TYPE OF SORTING

- Insertion sorting
- Selection sorting
- Bubble sorting
- Heap sorting
- Quick sorting



INSERTION SORT

Description: Insertion Sort builds the final sorted array one item at a time by inserting each item into its correct position within the sorted portion of the array.

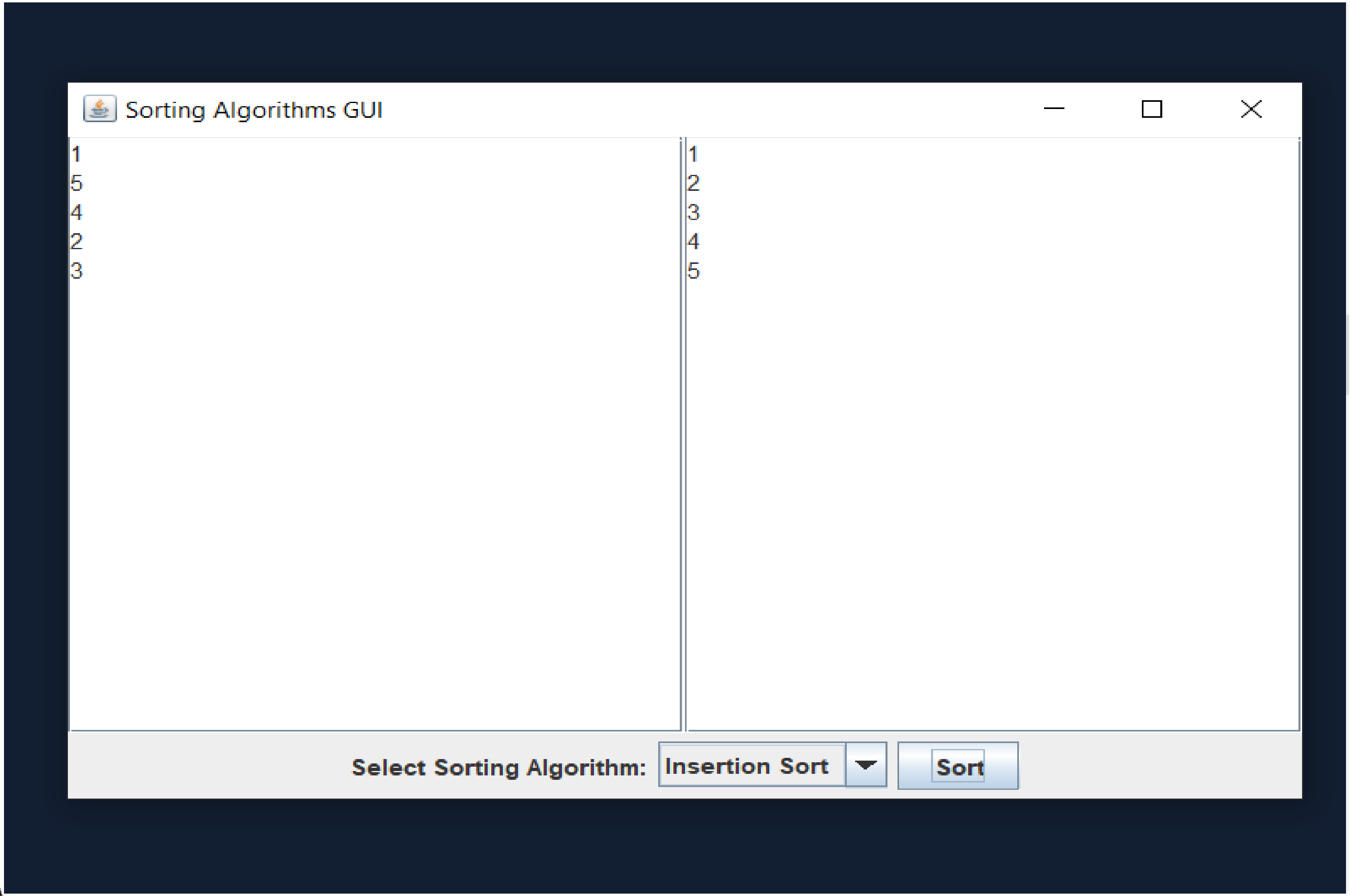
- **Time Complexity:** O(n^2)
- **Space Complexity:** O(1)

CODE:</>

```
private void insertionSort() {
    String[] inputLines = inputArea.getText().split(regex: "\\s+");
    int[] array = new int[inputLines.length];
    for (int i = 0; i < inputLines.length; i++) {
        array[i] = Integer.parseInt(inputLines[i]);
    }
    insertionSort(array);
}

private void insertionSort(int[] array) {
    int n = array.length;
    for (int i = 1; i < n; i++) {
        int key = array[i];
        int j = i - 1;
        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = key;
    }
    displaySortedArray(array);
}
```

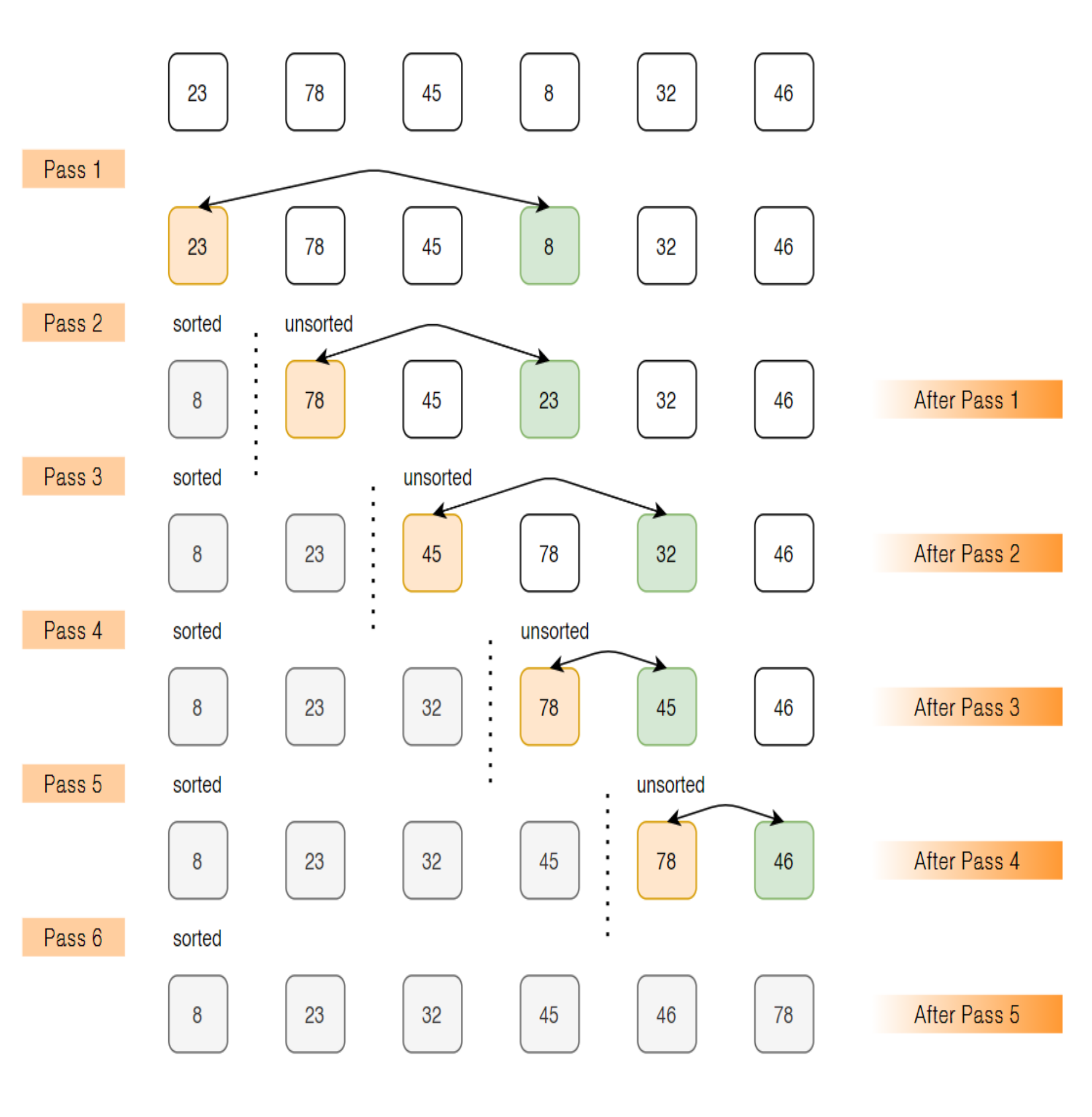
RUN THE CODE:


Sorting Algorithms GUI
1
5
4
2
3
1
2
3
4
5
Select Sorting Algorithm:
Insertion Sort
Sort

SELECTION SORT

Description: Selection Sort divides the input array into two parts: sorted and unsorted. It repeatedly selects the minimum element from the unsorted portion and swaps it with the first unsorted element.

- **Time Complexity:** O(n^2)
- **Space Complexity:** O(1)



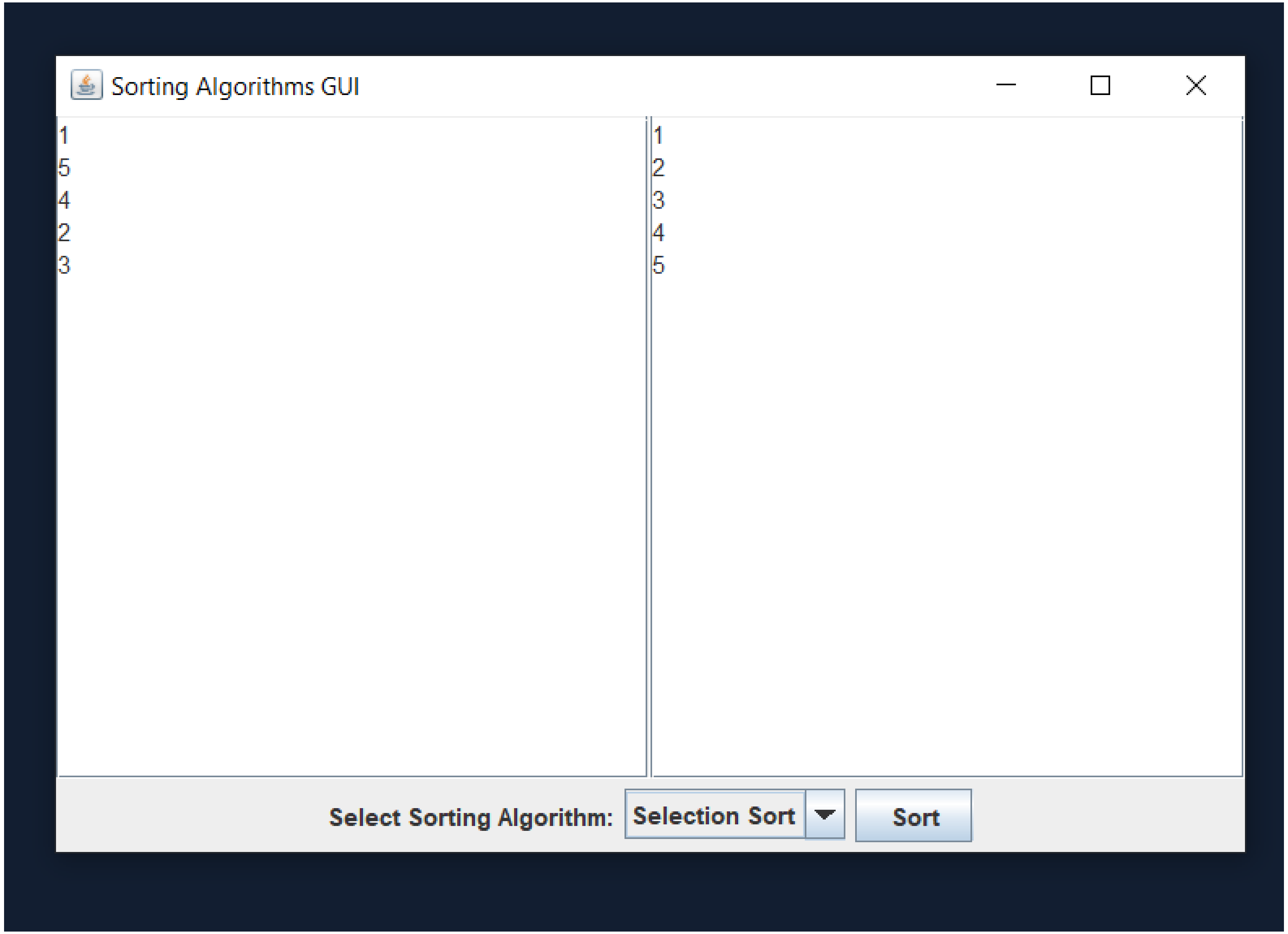
23 78 45 8 32 46
Pass 1
23 78 45 8 32 46
Pass 2
sorted
unsorted
8 78 45 23 32 46
After Pass 1
Pass 3
sorted
unsorted
8 23 45 78 32 46
After Pass 2
Pass 4
sorted
unsorted
8 23 32 78 45 46
After Pass 3
Pass 5
sorted
unsorted
8 23 32 45 78 46
After Pass 4
Pass 6
sorted
8 23 32 45 46 78
After Pass 5

CODE:</>

```java
private void selectionSort() {
    String[] inputLines = inputArea.getText().split(regex: "\\s+");
    int[] array = new int[inputLines.length];
    for (int i = 0; i < inputLines.length; i++) {
        array[i] = Integer.parseInt(inputLines[i]);
    }
    selectionSort(array);
}

private void selectionSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }
        int temp = array[minIndex];
        array[minIndex] = array[i];
        array[i] = temp;
    }
    displaySortedArray(array);
}
```

RUN THE CODE:



Sorting Algorithms GUI
1
5
4
2
3
1
2
3
4
5
Select Sorting Algorithm:
Selection Sort
Sort

BUBBLE SORT

Description: Merge Sort is a divide and conquer algorithm that divides the input array into smaller sub-arrays, recursively sorts them, and then merges them back together.

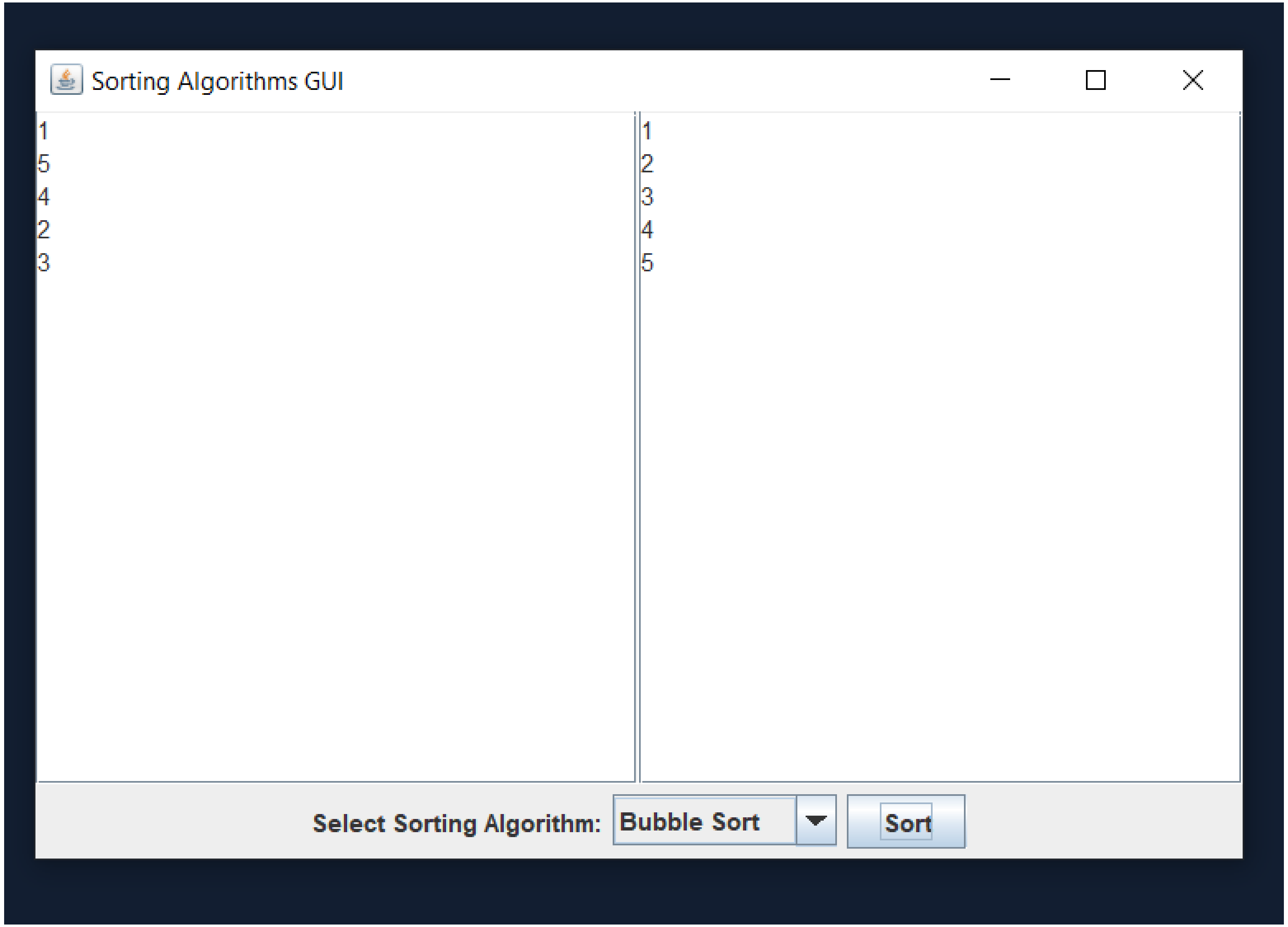
- **Time Complexity:** O(n log n)
- **Space Complexity:** O(n)

CODE:</>

```
private void bubbleSort() {
    String[] inputLines = inputArea.getText().split(regex: "\\s+");
    int[] array = new int[inputLines.length];
    for (int i = 0; i < inputLines.length; i++) {
        array[i] = Integer.parseInt(inputLines[i]);
    }
    bubbleSort(array);
}

private void bubbleSort(int[] array) {
    int n = array.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    displaySortedArray(array);
}
```

RUN CODE:



Sorting Algorithms GUI
1
5
4
2
3
1
2
3
4
5
Select Sorting Algorithm:
Bubble Sort
Sort

HEAP SORT

Description: Heap Sort uses a binary heap data structure to repeatedly remove the maximum element from the heap and place it at the end of the sorted array.

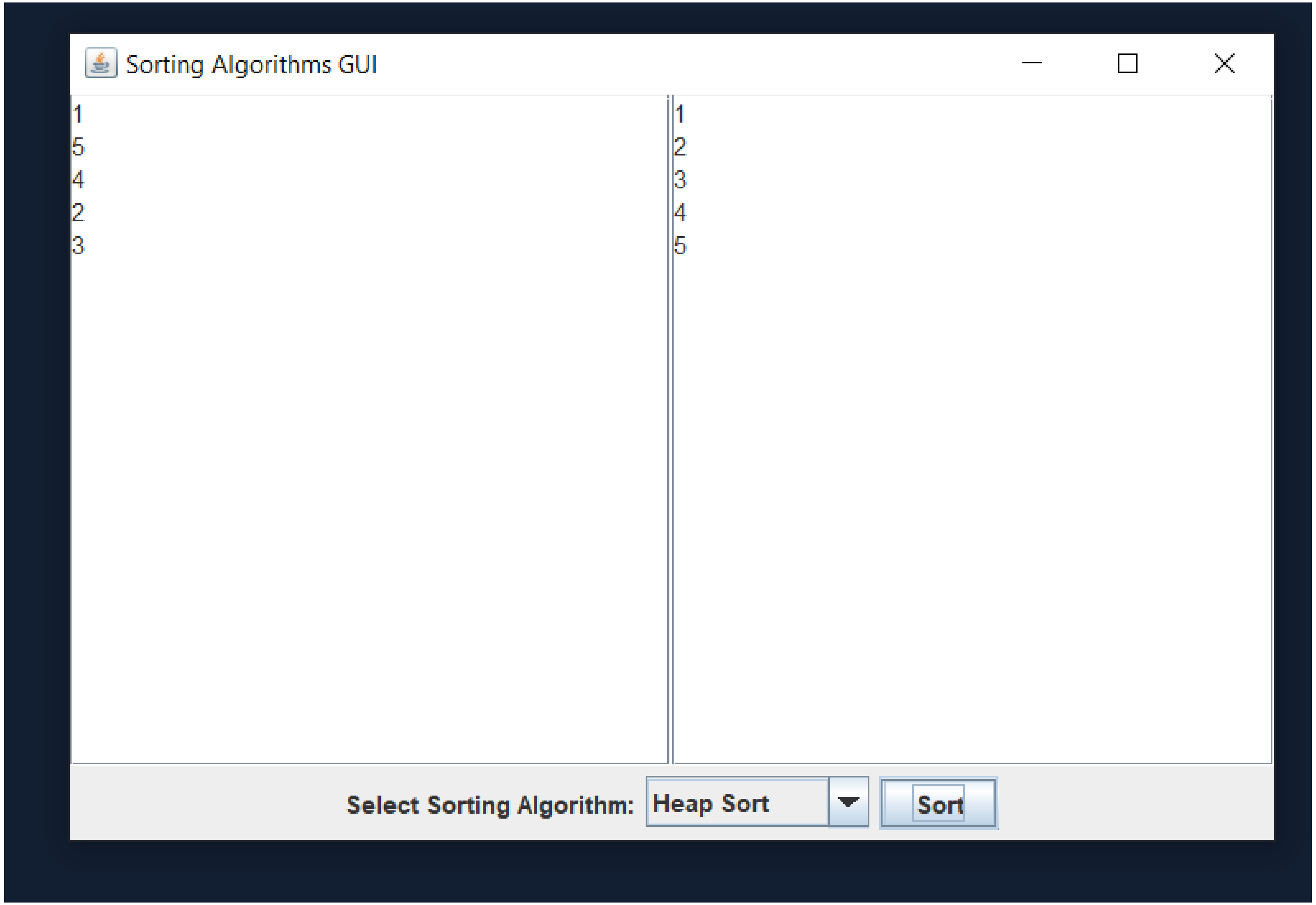
- **Time Complexity:** O(n log n)
- **Space Complexity:** O(1)

CODE:</>

```
private void heapSort() {
    String[] inputLines = inputArea.getText().split(regex: "\\s+");
    int[] array = new int[inputLines.length];
    for (int i = 0; i < inputLines.length; i++) {
        array[i] = Integer.parseInt(inputLines[i]);
    }
    heapSort(array);
}

private void heapSort(int[] array) {
    int n = array.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(array, n, i);
    }
    for (int i = n - 1; i > 0; i--) {
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;
        heapify(array, n: i, i: 0);
    }
    displaySortedArray(array);
}
```

RUN CODE:



Sorting Algorithms GUI
1
5
4
2
3
1
2
3
4
5
Select Sorting Algorithm:
Heap Sort
Sort

QUICK SORT

Description: Quick Sort selects a pivot element and partitions the array into two sub-arrays, recursively sorting each sub-array. It is known for its efficiency and is widely used in practice.

- **Time Complexity:** O(n log n) average case, O(n^2) worst case
- **Space Complexity:** O(log n) average case, O(n) worst case

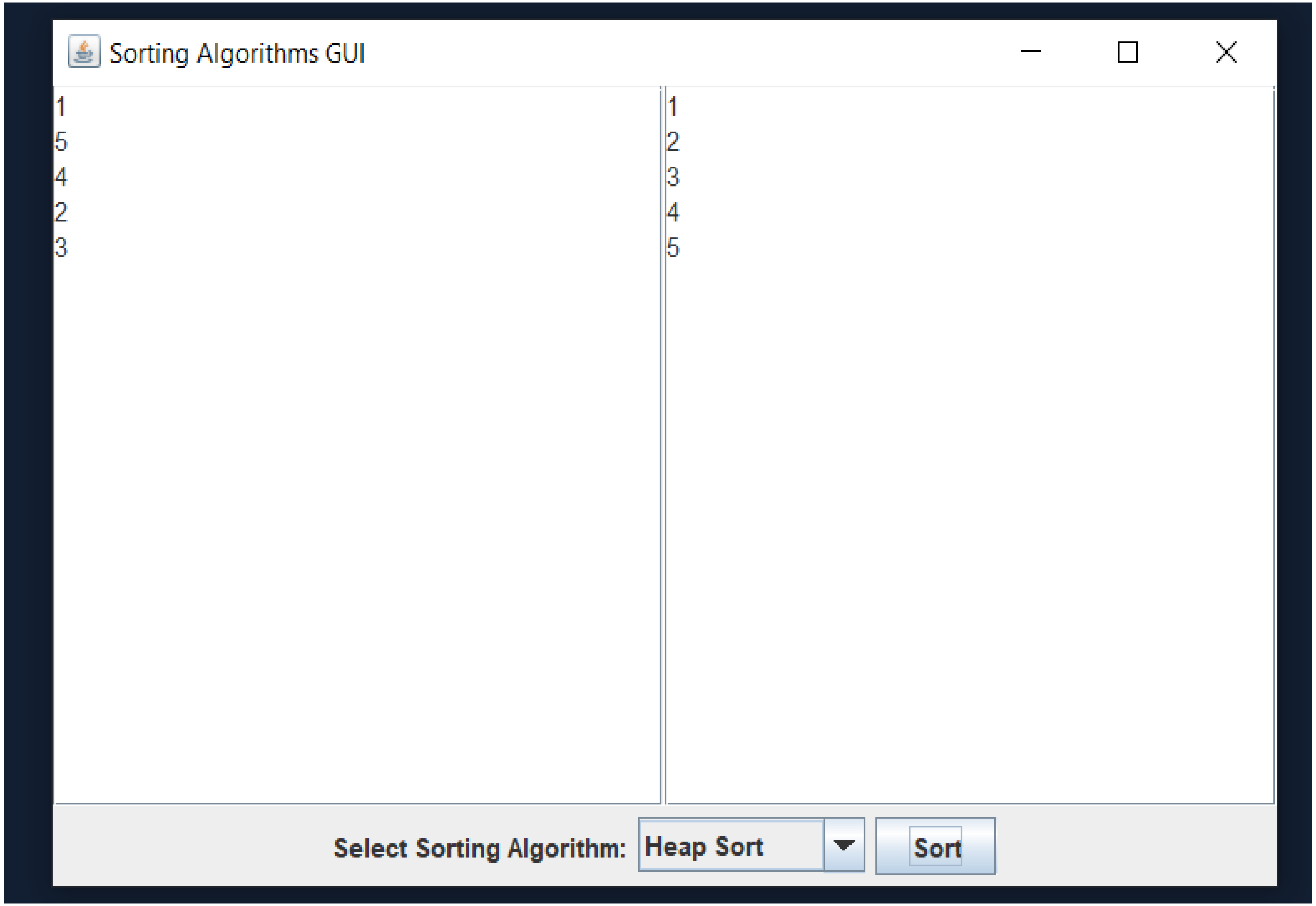
CODE:</>

```java
private void quickSort() {
    String[] inputLines = inputArea.getText().split(regex: "\\s+");
    int[] array = new int[inputLines.length];
    for (int i = 0; i < inputLines.length; i++) {
        array[i] = Integer.parseInt(inputLines[i]);
    }
    quickSort(array, low: 0, array.length - 1);
}

private void quickSort(int[] array, int low, int high) {
    if (low < high) {
        int pi = partition(array, low, high);
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
    displaySortedArray(array);
}

private int partition(int[] array, int low, int high) {
    int pivot = array[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;
```

RUN CODE:


Sorting Algorithms GUI
1
5
4
2
3
1
2
3
4
5
Select Sorting Algorithm:
Heap Sort
Sort

TYPES OF SEARCH

- LINER SEARCH
- BINARY SEARCH

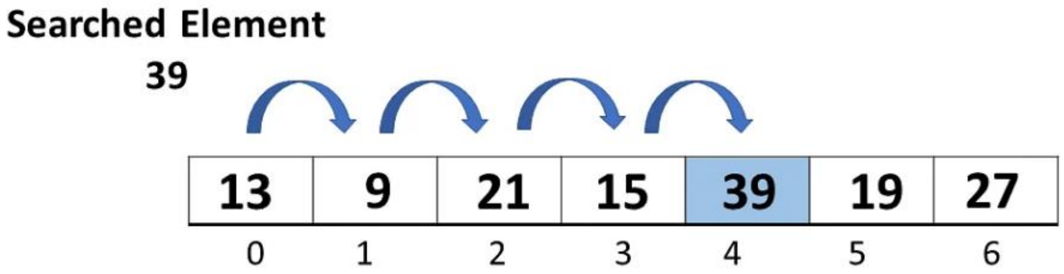
LINER SEARCH

Description: Linear search is the simplest search algorithm. It checks every element in the list sequentially until the desired element is found or the list ends.

•**Time Complexity**: O(n), where n is the number of elements in the list.

•**Space Complexity**: O(1).

Use Cases: Linear search is used in small or unsorted datasets where other search algorithms are not applicable.

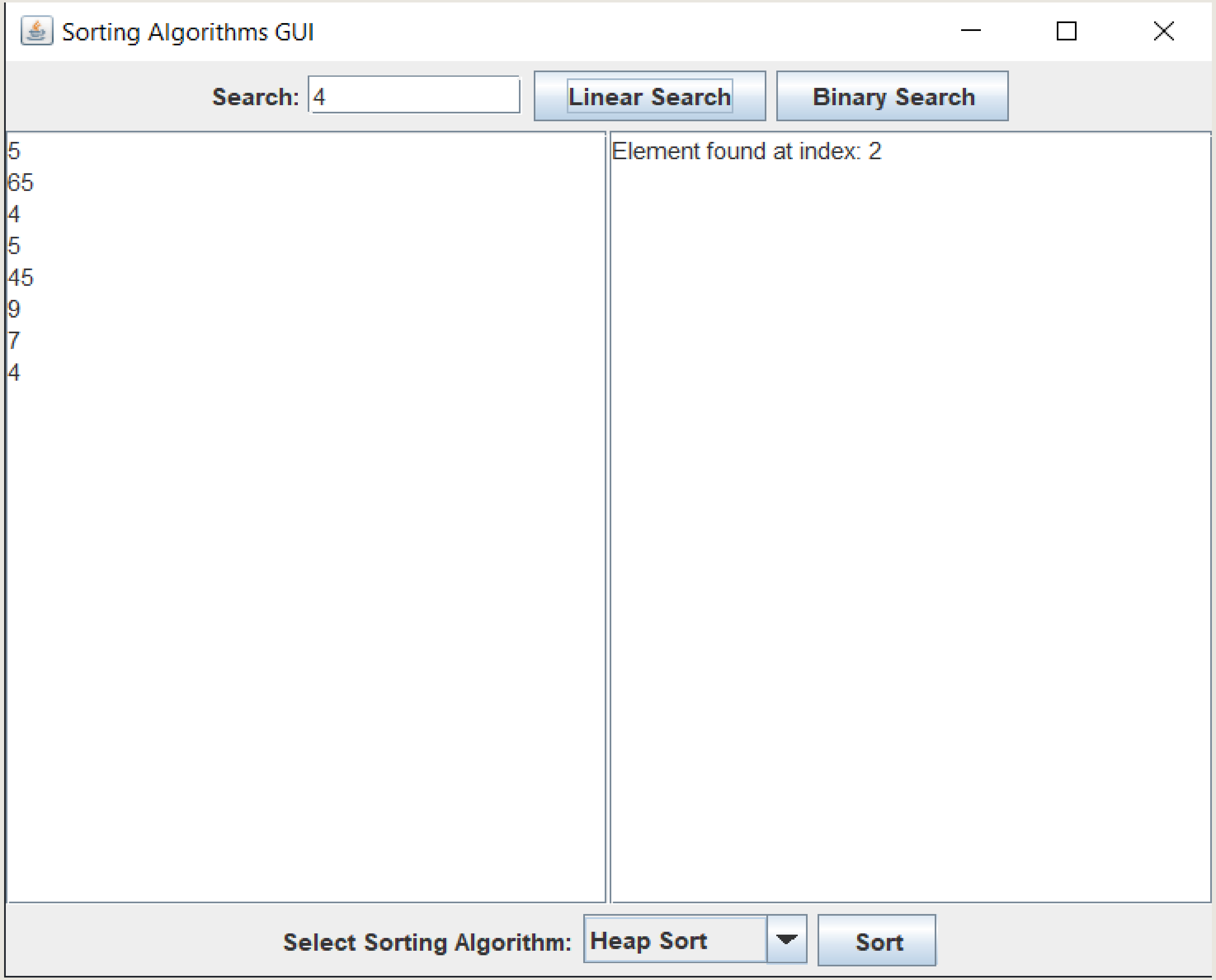


CODE:</>

```java
private void linearSearch() {
    int[] array = getInputArray();
    if (array != null) {
        try {
            int key = Integer.parseInt(s: searchField.getText());
            int index = linearSearch(array, key);
            outputArea.setText(index == -1 ? "Element not found" : "Element found at index: " + index);
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid search key. Please enter a number.", title: "Error",
        }
    }
}

private int linearSearch(int[] array, int key) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == key) {
            return i;
        }
    }
    return -1;
}
```

RUN CODE:


Sorting Algorithms GUI
Search: 4
Linear Search
Binary Search
5
65
4
5
45
9
7
4
Element found at index: 2
Select Sorting Algorithm:
Heap Sort
Sort

BINARY SEARCH

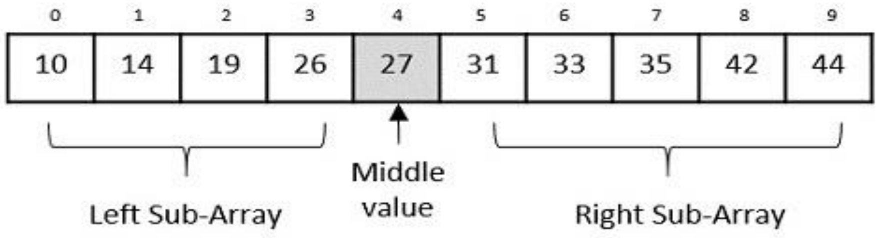
Description: Binary search is an efficient algorithm for finding an item in a sorted list by repeatedly dividing the search interval in half.

Characteristics:

• **Time Complexity**: O(log n), where n is the number of elements in the list.

• **Space Complexity**: O(1).

Use Cases: Binary search is used in large, sorted datasets where quick lookup times are necessary, such as searching in databases and dictionaries.

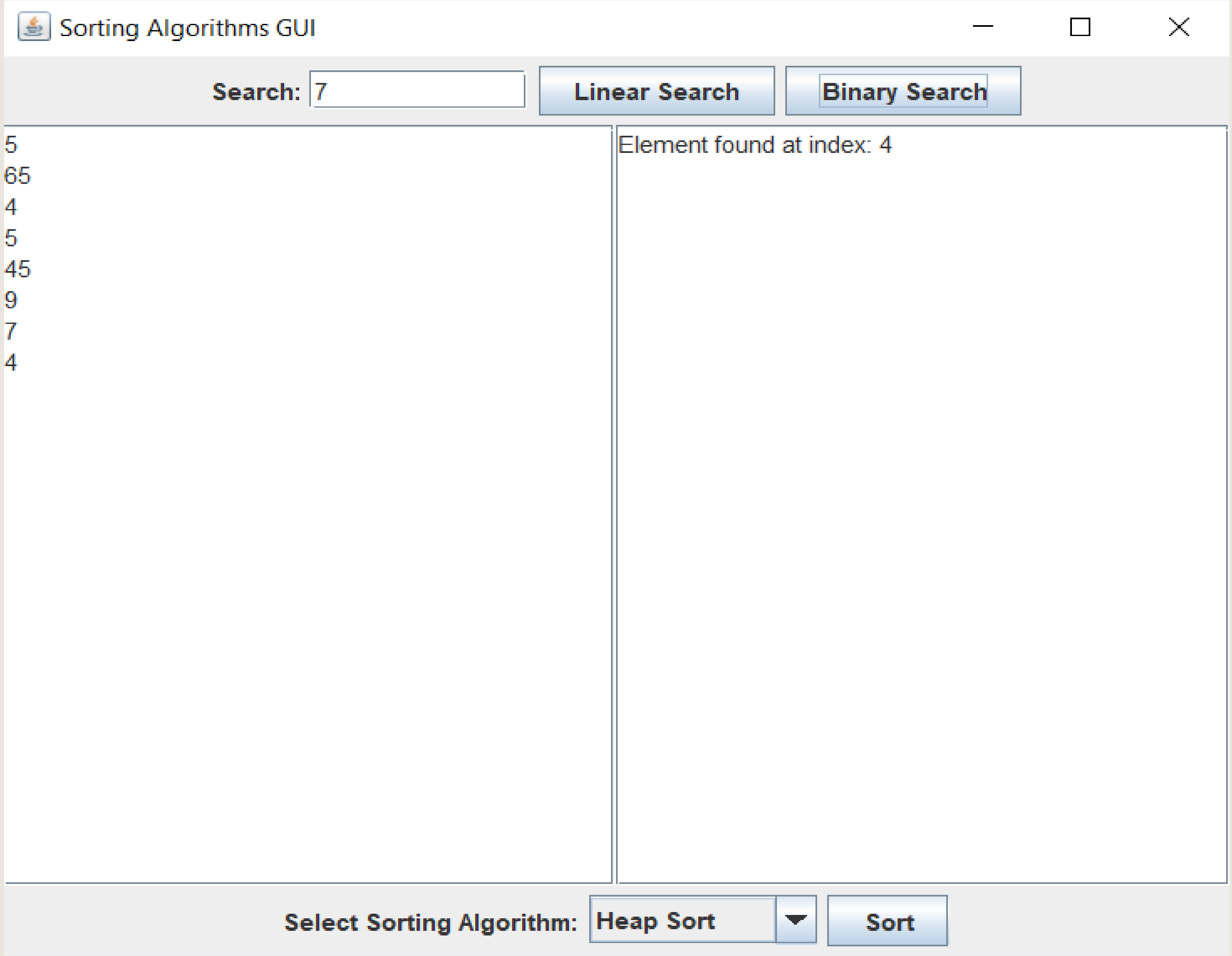


CODE:</>

```java
private void binarySearch() {
    int[] array = getInputArray();
    if (array != null) {
        try {
            int key = Integer.parseInt(s: searchField.getText());
            quickSort(array, low: 0, array.length - 1); // Ensure the array is sorted before binary search
            int index = binarySearch(array, key);
            outputArea.setText(index == -1 ? "Element not found" : "Element found at index: " + index);
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid search key. Please enter a number.", title: "Error",
        }
    }
}

private int binarySearch(int[] array, int key) {
    int left = 0, right = array.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (array[mid] == key) {
            return mid;
        }
        if (array[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

RUN CODE:



Sorting Algorithms GUI
Search: 7
Linear Search
Binary Search
5
65
4
5
45
9
7
4
Element found at index: 4
Select Sorting Algorithm:
Heap Sort
Sort

THANK YOU