SORTING TECHNIQUES VISUALIZATION

RAVITEJA KONDAGORLA

DESCRIPTION

This Java project visualizes sorting techniques using a Swing-based graphical user interface. The application features buttons for Bubble, Insertion, Selection, and Quick sorts, allowing real-time visualization of sorting algorithms on a dynamically changing array of bars. Users can reset the array and control the speed of visualization, providing an interactive and educational experience for understanding sorting techniques.

```
import javax.swing.*;
import java.awt.*;
import java.util.Arrays;
import java.util.Random;

public class MySortingApp extends JFrame {
    private int[] array;
    private SortPanel sortPanel;
    private JComboBox<String> algorithmComboBox;
    private JRadioButton ascendingRadioButton;
    private JRadioButton descendingRadioButton;

    private MySortingApp(int[] array) {
        this.array = Arrays.copyOf(array, array.length);
        this.sortPanel = new SortPanel();
        initializeUI();
    }
    private void initializeUI() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        JPanel controlPanel = new JPanel();

        // Combo box for selecting the sorting algorithm
        algorithmComboBox = new JComboBox<>(new String[]{"Bubble Sort",
"Insertion Sort", "Selection Sort", "Quick Sort", "Merge Sort", "Heap
Sort"});
        algorithmComboBox.setSelectedIndex(0); // Default selection
        controlPanel.add(algorithmComboBox);

        // Radio buttons for selecting the order
        ascendingRadioButton = new JRadioButton("Ascending");
        descendingRadioButton = new JRadioButton("Descending");
        ButtonGroup orderGroup = new ButtonGroup();
        orderGroup.add(ascendingRadioButton);
        orderGroup.add(descendingRadioButton);
        ascendingRadioButton.setSelected(true); // Default selection
        controlPanel.add(ascendingRadioButton);
        controlPanel.add(descendingRadioButton);

        JButton sortButton = new JButton("Sort");
        JButton resetButton = new JButton("Reset");

        sortButton.addActionListener(e -> performSort());
        resetButton.addActionListener(e -> resetArray());
```

```
        controlPanel.add(sortButton);
        controlPanel.add(resetButton);

        add(controlPanel, BorderLayout.NORTH);
        add(sortPanel, BorderLayout.CENTER);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
    private void performSort() {
        String selectedAlgorithm = (String)
algorithmComboBox.getSelectedItem();
        boolean isAscending = ascendingRadioButton.isSelected();

        switch (selectedAlgorithm) {
            case "Bubble Sort":
                performBubbleSort(isAscending);
                break;
            case "Insertion Sort":
                performInsertionSort(isAscending);
                break;
            case "Selection Sort":
                performSelectionSort(isAscending);
                break;
            case "Quick Sort":
                performQuickSort(isAscending);
                break;
            case "Merge Sort":
                performMergeSort(isAscending);
                break;
            case "Heap Sort":
                performHeapSort(isAscending);
                break;
        }
    }
    private void performBubbleSort(boolean isAscending) {
        performSort(() -> {
            for (int i = 0; i < array.length - 1; i++) {
                for (int j = 0; j < array.length - 1 - i; j++) {
                    if ((isAscending && array[j] > array[j + 1]) ||
(!isAscending && array[j] < array[j + 1])) {
                        swap(j, j + 1);
                    }
                }
            }
        });
```

```
    }
    private void performInsertionSort(boolean isAscending) {
        performSort(() -> {
            for (int i = 1; i < array.length; i++) {
                int current = array[i];
                int j = i;

                while (j > 0 && ((isAscending && array[j - 1] >
current) || (!isAscending && array[j - 1] < current))) {
                    array[j] = array[j - 1];
                    j--;
                }

                array[j] = current;
            }
        });
    }
    private void performSelectionSort(boolean isAscending) {
        performSort(() -> {
            for (int i = 0; i < array.length - 1; i++) {
                int minIndex = i;
                for (int j = i + 1; j < array.length; j++) {
                    if ((isAscending && array[j] < array[minIndex]) ||
(!isAscending && array[j] > array[minIndex])) {
                        minIndex = j;
                    }
                }
                if (minIndex != i) {
                    swap(i, minIndex);
                }
            }
        });
    }
    private void performQuickSort(boolean isAscending) {
        performSort(() -> quickSort(0, array.length - 1, isAscending));
    }
    private void quickSort(int low, int high, boolean isAscending) {
        if (low < high) {
            int pivotIndex = partition(low, high, isAscending);
            quickSort(low, pivotIndex - 1, isAscending);
            quickSort(pivotIndex + 1, high, isAscending);
        }
    }
    private int partition(int low, int high, boolean isAscending) {
        int pivot = array[high];
        int i = low - 1;
```

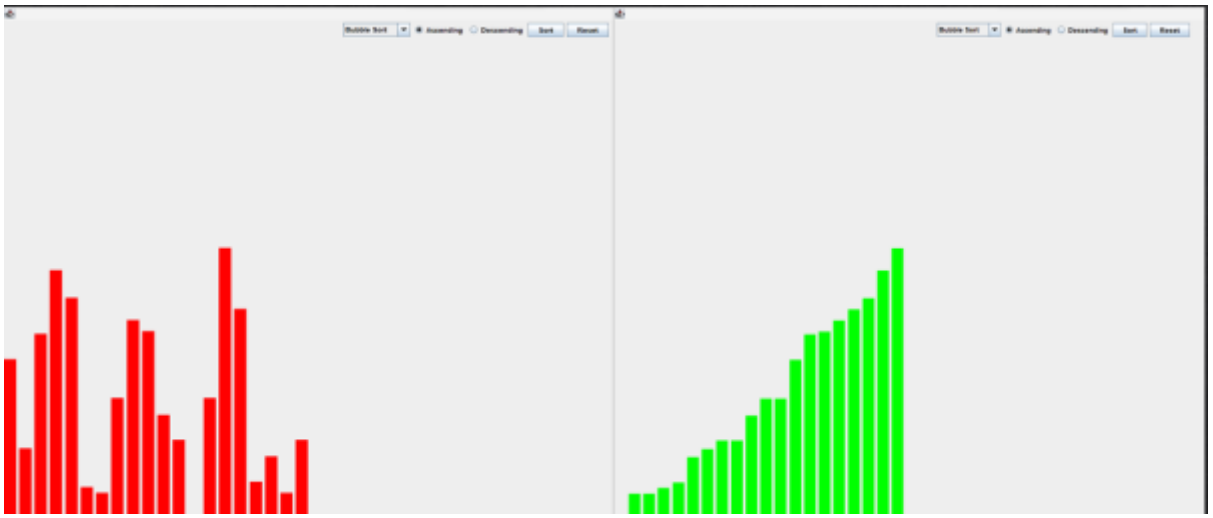
```
        for (int j = low; j < high; j++) {
            if ((isAscending && array[j] <= pivot) || (!isAscending &&
array[j] >= pivot)) {
                i++;
                swap(i, j);
            }
        }
        swap(i + 1, high);
        return i + 1;
    }
    private void performMergeSort(boolean isAscending) {
        performSort(() -> mergeSort(0, array.length - 1, isAscending));
    }
    private void mergeSort(int low, int high, boolean isAscending) {
        if (low < high) {
            int mid = (low + high) / 2;
            mergeSort(low, mid, isAscending);
            mergeSort(mid + 1, high, isAscending);
            merge(low, mid, high, isAscending);
        }
    }
    private void merge(int low, int mid, int high, boolean isAscending)
{
        int[] temp = Arrays.copyOf(array, array.length);
        Color[] tempColors = Arrays.copyOf(sortPanel.barColors,
sortPanel.barColors.length);
        int i = low;
        int j = mid + 1;
        int k = low;
        while (i <= mid && j <= high) {
            if ((isAscending && temp[i] <= temp[j]) || (!isAscending &&
temp[i] >= temp[j])) {
                array[k] = temp[i];
                sortPanel.barColors[k] = tempColors[i];
                k++;
                i++;
            } else {
                array[k] = temp[j];
                sortPanel.barColors[k] = tempColors[j];
                k++;
                j++;
            }
        }
        while (i <= mid) {
            array[k] = temp[i];
            sortPanel.barColors[k] = tempColors[i];
```

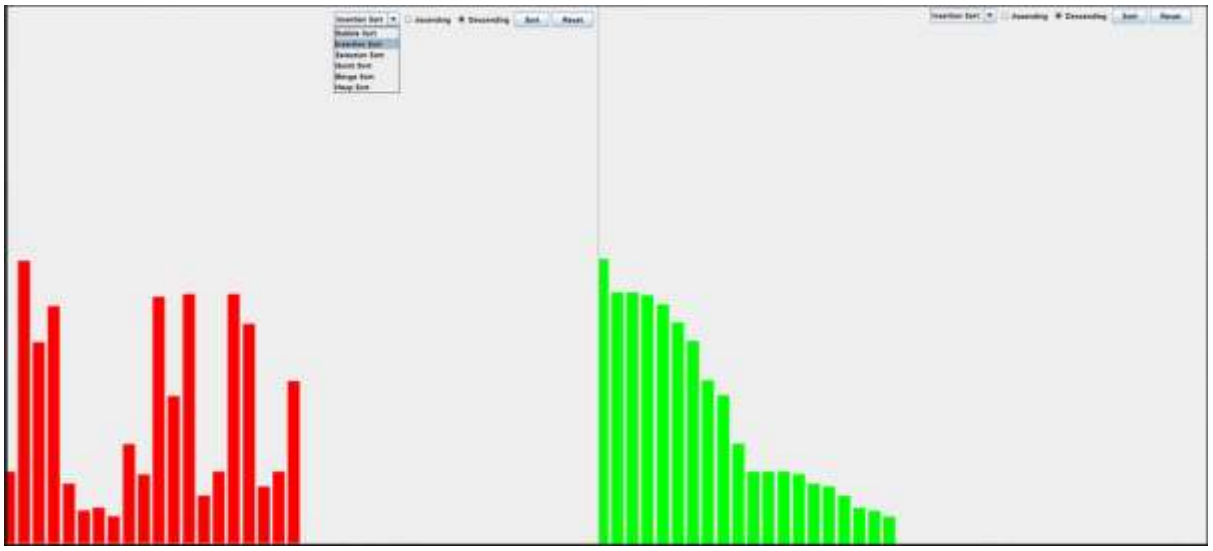
```
            k++;
            i++;
        }
        sortPanel.repaint();
    }
    private void performHeapSort(boolean isAscending) {
        performSort(() -> {
            int n = array.length;
            for (int i = n / 2 - 1; i >= 0; i--) {
                heapify(n, i, isAscending);
            }
            for (int i = n - 1; i > 0; i--) {
                swap(0, i);
                heapify(i, 0, isAscending);
            }
        });
    }
    private void heapify(int n, int i, boolean isAscending) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && ((isAscending && array[left] > array[largest])
|| (!isAscending && array[left] < array[largest]))) {
            largest = left;
        }
        if (right < n && ((isAscending && array[right] >
array[largest]) || (!isAscending && array[right] < array[largest]))) {
            largest = right;
        }
        if (largest != i) {
            swap(i, largest);
            heapify(n, largest, isAscending);
        }
    }
    private void performSort(Runnable sortingAlgorithm) {
        new Thread(() -> {
            sortingAlgorithm.run();
            Color[] sortedColors = new Color[array.length];
            Arrays.fill(sortedColors, Color.GREEN); // Sorted color
            sortPanel.updateArray(array, sortedColors);
        }).start();
    }
    private void resetArray() {
        array = generateStartingArray(array.length, 0, 100);
        Color[] resetColors = new Color[array.length];
        Arrays.fill(resetColors, Color.RED); // Reset color
```

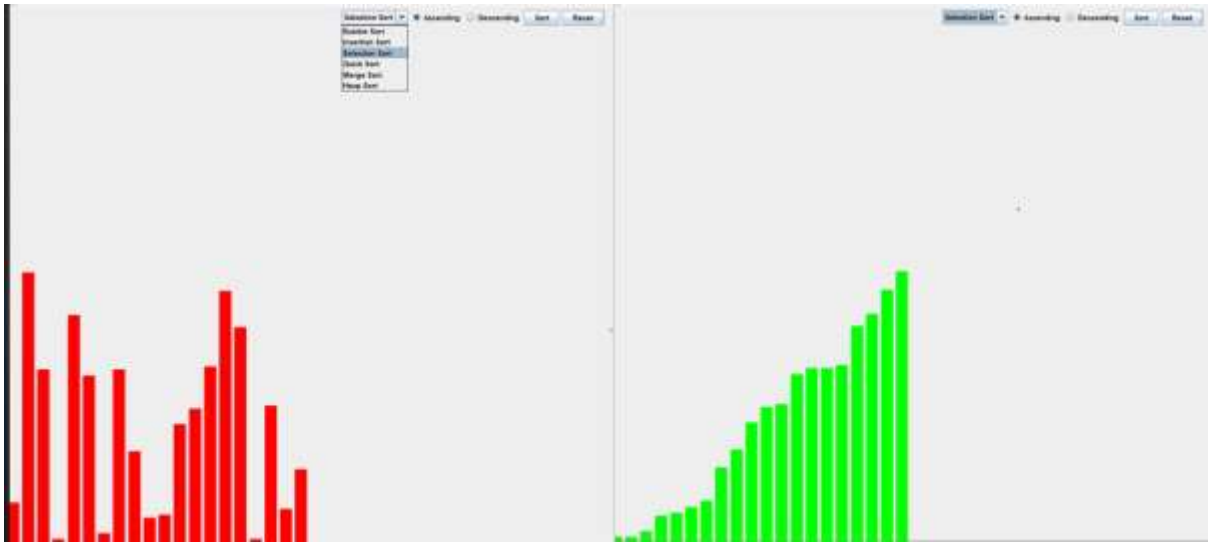
```
        sortPanel.updateArray(array, resetColors);
    }
    private void swap(int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;

        Color tempColor = sortPanel.barColors[i];
        sortPanel.barColors[i] = sortPanel.barColors[j];
        sortPanel.barColors[j] = tempColor;
        sortPanel.repaint();
        try {
            Thread.sleep(100);  // Adjust the sleep duration for
visualization speed
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    private class SortPanel extends JPanel {
        private static final int BAR_WIDTH = 20;
        private static final int BAR_SPACING = 5;
        private Color[] barColors;
        private SortPanel() {
            this.barColors = new Color[array.length];
            Arrays.fill(barColors, Color.RED); // Initial color
        }
        private void updateArray(int[] newArray, Color[] newColors) {
            array = Arrays.copyOf(newArray, newArray.length);
            barColors = Arrays.copyOf(newColors, newColors.length);
            repaint();
        }
        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            Graphics2D g2d = (Graphics2D) g;

            int x = 0;
            for (int i = 0; i < array.length; i++) {
                int barHeight = array[i] * 5;  // Adjust for
visualization
                g2d.setColor(barColors[i]);
                g2d.fillRect(x, getHeight() - barHeight, BAR_WIDTH,
barHeight);
                x += BAR_WIDTH + BAR_SPACING;
            }
        }
```

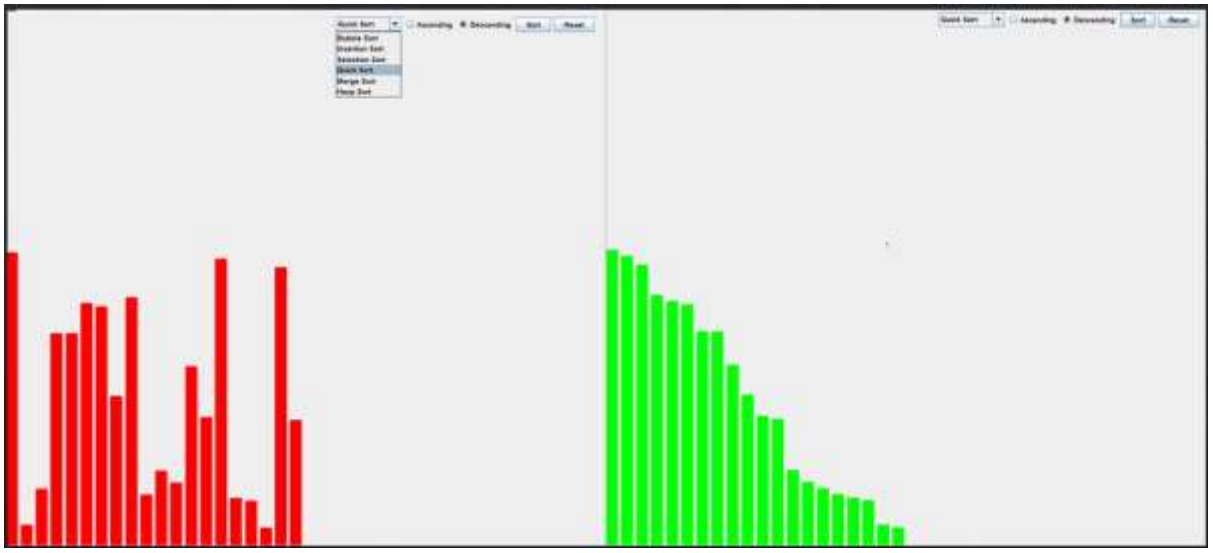
```
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            int n = 20;
            int minVal = 0;
            int maxVal = 100;
            int[] array = generateStartingArray(n, minVal, maxVal);

            new MySortingApp(array);
        });
    }
    private static int[] generateStartingArray(int n, int minVal, int
maxVal) {
        int[] array = new int[n];
        Random random = new Random();

        for (int i = 0; i < n; i++) {
            array[i] = random.nextInt(maxVal - minVal + 1) + minVal;
        }
        return array;
    }
}
```

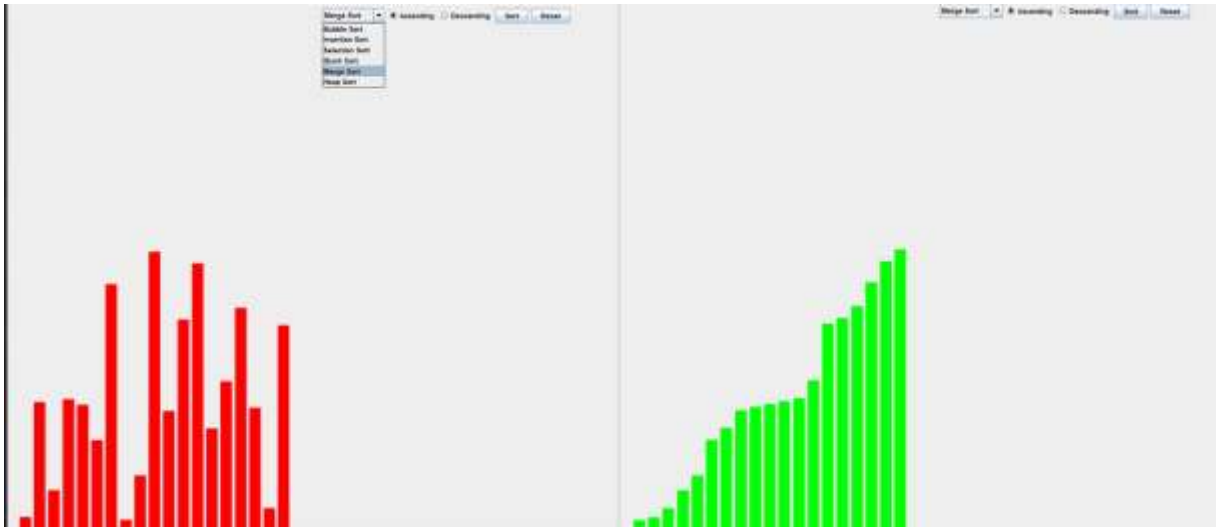
OUTPUT

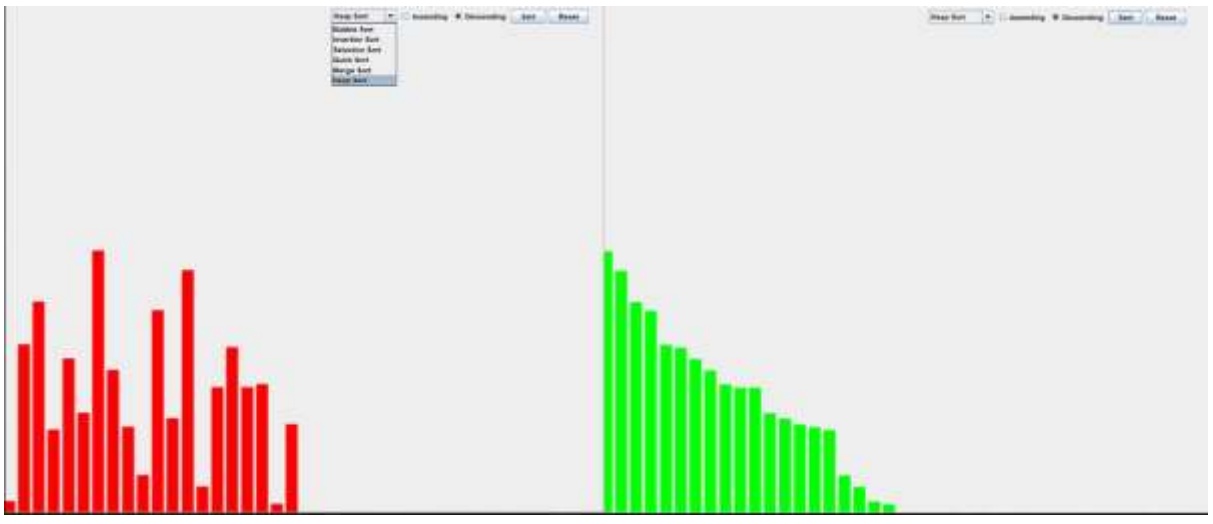












EXPLANATION

This Java code defines a Swing-based GUI application for visualizing various sorting algorithms. Swing is a set of GUI (Graphical User Interface) components for Java. It is part of the Java Foundation Classes (JFC) and provides a rich set of libraries for creating desktop applications with graphical user interfaces. Swing is built on top of the Abstract Window Toolkit (AWT) but offers more advanced and flexible components compared to AWT

Java Swing Libraries:

1.javax.swing.JFrame:

- import javax.swing.*;
- The JFrame class is part of the Swing library and is used to create a windowed GUI application.

2.java.awt.*:

- import java.awt.*;
- The java.awt package provides classes for creating GUI components such as panels and layouts.

Java Utility Libraries:

1.java.util.Arrays:

- import java.util.Arrays;
- The Arrays class provides utility methods for manipulating arrays, such as copying arrays.

2.java.util.Random:

- import java.util.Random;

- The Random class is used to generate random numbers. In this code, it is used for initializing the initial array with random values.

Swing Components:

1.JComboBox<E>:

- private JComboBox<String> algorithmComboBox;
- A Swing component that provides a dropdown menu. In this code, it is used for selecting sorting algorithms.

2.JRadioButton:

- private JRadioButton ascendingRadioButton;
- private JRadioButton descendingRadioButton;
- Swing components representing radio buttons. They are used for selecting the sorting order (ascending or descending).

3.ButtonGroup:

- ButtonGroup orderGroup = new ButtonGroup();
- A Swing component that groups radio buttons, ensuring that only one radio button in the group can be selected at a time.

Swing UI Components:

1.JButton:

- JButton sortButton = new JButton("Sort");

- JButton resetButton = new JButton("Reset");
- Swing components representing buttons. They are used to trigger sorting and resetting actions.

Swing Event Handling:

1.ActionListener:

- sortButton.addActionListener(e -> performSort());
- resetButton.addActionListener(e -> resetArray());
- The ActionListener interface is used to handle button click events. In this code, lambda expressions are used to define the actions when the buttons are clicked.

Threading:

1.Thread:

- new Thread(() -> { /* sorting logic */ }).start();
- The Thread class is used to execute sorting algorithms in a separate thread, allowing for real-time visualization without freezing the UI.

ava Graphics:

1.java.awt.Graphics:

- protected void paintComponent(Graphics g) { /* drawing logic */ }
- The Graphics class is part of the AWT library and is used for rendering graphics. The paintComponent method is overridden to draw the bars representing array elements.

2.java.awt.Graphics2D:

- Graphics2D g2d = (Graphics2D) g;
- The Graphics2D class extends Graphics and provides additional capabilities for drawing shapes and text.

Color:

1.java.awt.Color:

- Arrays.fill(resetColors, Color.RED);
- Arrays.fill(sortedColors, Color.GREEN);
- The Color class is used to set colors for visualization. In this code, red is used for the initial state, and green is used for the sorted state.

SwingUtilities:

1.SwingUtilities.invokeLater:

- SwingUtilities.invokeLater(() -> { /* code to run on the EDT */ });

The invokeLater method is used to ensure that the GUI is created and modified on the Event Dispatch Thread (EDT), which


THANK YOU!!