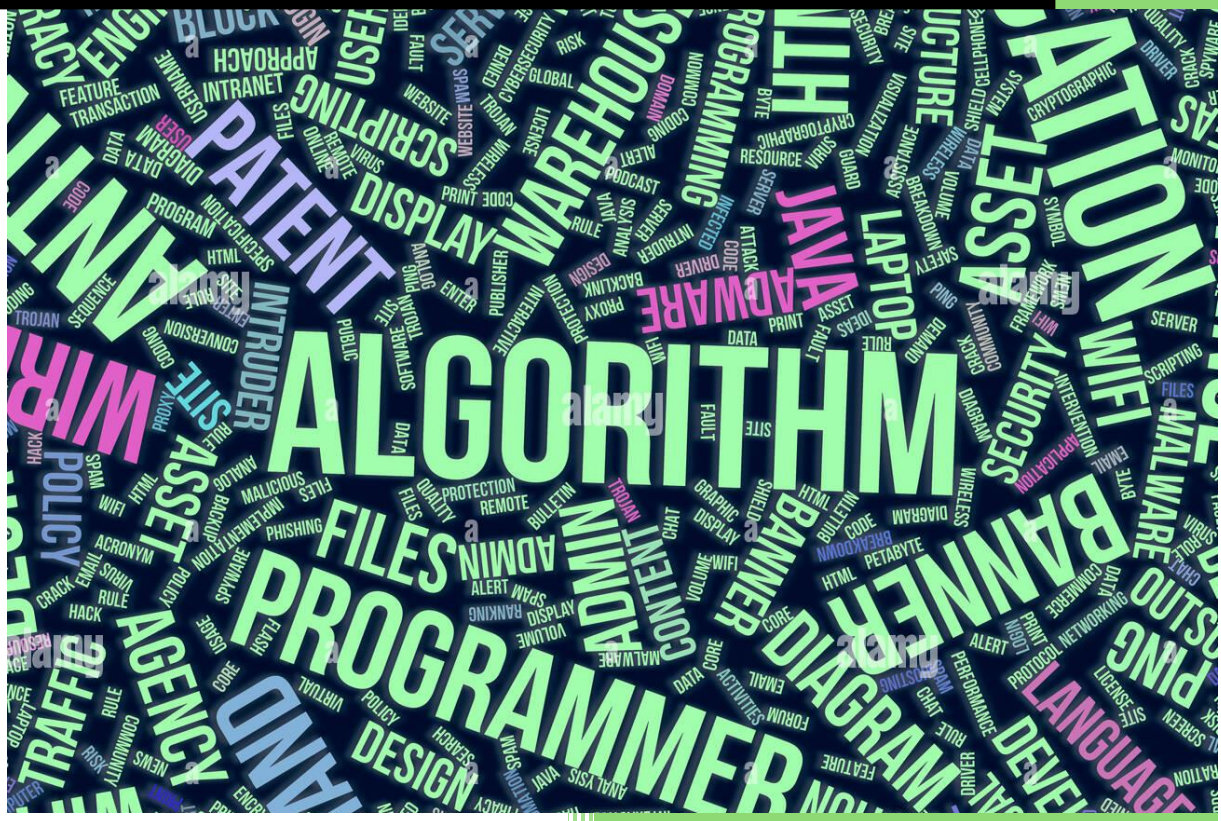
2024

Algorithms Project



Algorithms Master

ERU

5/22/2024

Team:

- Mahmoud Maher 225200
- Arwa Hassan 225176
- Youmna Alaa 225023
- Akram Mohamed 225186

Introduction:

Algorithms are fundamental tools in computer science that allow for the efficient processing of data. This college project aims to explore and implement various sorting and searching algorithms using an array of numbers. By examining the performance and characteristics of different sorting and searching techniques, we gain valuable insights into their practical applications and efficiency.

Sorting Algorithms:

Sorting is a critical operation in many applications, ranging from organizing data to optimizing search operations. This project includes the implementation and comparison of six different sorting algorithms:

1. **Selection Sort**: A simple comparison-based algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion and places it at the beginning.
2. **Bubble Sort**: Another comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
3. **Merge Sort**: A divide-and-conquer algorithm that divides the array into halves, sorts each half, and then merges the sorted halves to produce a sorted array.
4. **Heap Sort**: A comparison-based sorting technique that uses a binary heap data structure. It first converts the array into a heap and then repeatedly extracts the maximum (or minimum) element to build the sorted array.
5. **Insertion Sort**: A simple and intuitive sorting algorithm that builds the final sorted array one item at a time, inserting each new element into its correct position.
6. **Quick Sort**: Another divide-and-conquer algorithm that selects a 'pivot' element and partitions the array into sub-arrays of elements less than and greater than the pivot, recursively sorting the sub-arrays.

Searching Algorithms:

Searching is another fundamental operation, essential for retrieving data efficiently. This project also includes the implementation and comparison of two search algorithms:

1. **Linear Search**: A straightforward method that checks each element in the array one by one until the desired element is found or the end of the array is reached.
2. **Binary Search**: An efficient algorithm for finding an element in a sorted array by repeatedly dividing the search interval in half, discarding the half in which the element cannot lie.

Objectives:

The primary objectives of this project are:

- To understand the implementation details of different sorting and searching algorithms.
- To compare the time and space complexities of these algorithms.
- To analyze the performance of these algorithms on various datasets.
- To gain hands-on experience with algorithm design and analysis.

By the end of this project, we will have a comprehensive understanding of how different algorithms can be applied to sort and search data, and we will be able to choose the appropriate algorithm for specific scenarios based on their performance characteristics.

Code:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Arrays;

public class Main extends JFrame {

    private JTextField inputField;
    private JTextArea outputArea;
    private JComboBox<String> algorithmSelector;
    private JPanel backgroundPanel;

    public Main() {
        setTitle("Sort and Search Algorithms");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        backgroundPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                ImageIcon icon = new
ImageIcon("C:\\Users\\KemoMoh11\\Desktop\\Second
Semester\\Algorithms\\Project\\New folder\\3222795.jpg"); // Path to your
background image
                Image image = icon.getImage();
                g.drawImage(image, 0, 0, getWidth(), getHeight(), this);
            }
        };
        backgroundPanel.setLayout(new BorderLayout());
        setContentPane(backgroundPanel);

        inputField = new JTextField(20);
        inputField.setBackground(Color.LIGHT_GRAY);
        inputField.setForeground(Color.BLACK);
        outputArea = new JTextArea();
        outputArea.setEditable(false);
        outputArea.setBackground(Color.BLACK);
        outputArea.setForeground(Color.WHITE);

        JPanel inputPanel = new JPanel();
        inputPanel.setOpaque(false);
        inputPanel.setLayout(new FlowLayout());
        JLabel inputTextLabel = new JLabel("Input (comma-separated):");
        inputTextLabel.setForeground(Color.WHITE);
        inputPanel.add(inputTextLabel);
        inputPanel.add(inputField);

        algorithmSelector = new JComboBox<>(new String[]{
                "Merge Sort", "Bubble Sort", "Heap Sort", "Quick Sort",
                "Insertion Sort", "Selection Sort", "Binary Search", "Linear
Search"
        });
```

```
        JButton executeButton = new JButton("Execute");
        executeButton.setBackground(Color.DARK_GRAY);
        executeButton.setForeground(Color.WHITE);
        executeButton.setFocusPainted(false);
        executeButton.setBorderPainted(false);
        executeButton.addActionListener(new ExecuteButtonListener());
        executeButton.addMouseListener(new java.awt.event.MouseAdapter() {
            @Override
            public void mousePressed(java.awt.event.MouseEvent evt) {
                executeButton.setBackground(Color.GRAY);
            }

            @Override
            public void mouseReleased(java.awt.event.MouseEvent evt) {
                executeButton.setBackground(Color.DARK_GRAY);
            }
        });

        JPanel controlPanel = new JPanel();
        controlPanel.setOpaque(false);
        controlPanel.setLayout(new FlowLayout());
        controlPanel.add(algorithmSelector);
        controlPanel.add(executeButton);

        backgroundPanel.add(inputPanel, BorderLayout.NORTH);
        backgroundPanel.add(new JScrollPane(outputArea),
BorderLayout.CENTER);
        backgroundPanel.add(controlPanel, BorderLayout.SOUTH);
    }

    private class ExecuteButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            String inputText = inputField.getText();
            String[] inputArray = inputText.split(",");
            int[] array =
Arrays.stream(inputArray).mapToInt(Integer::parseInt).toArray();
            String selectedAlgorithm = (String)
algorithmSelector.getSelectedItem();

            switch (selectedAlgorithm) {
                case "Merge Sort":
                    mergeSort(array, 0, array.length - 1);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
                case "Bubble Sort":
                    bubbleSort(array);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
                case "Heap Sort":
                    heapSort(array);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
```

```
                case "Quick Sort":
                    quickSort(array, 0, array.length - 1);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
                case "Insertion Sort":
                    insertionSort(array);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
                case "Selection Sort":
                    selectionSort(array);
                    outputArea.setText("Sorted Array: " +
Arrays.toString(array));
                    break;
                case "Binary Search":
                    int binaryKey =
Integer.parseInt(JOptionPane.showInputDialog("Enter key to search:"));
                    int binaryResult = binarySearch(array, binaryKey);
                    outputArea.setText("Binary Search Result: " +
binaryResult);
                    break;
                case "Linear Search":
                    int linearKey =
Integer.parseInt(JOptionPane.showInputDialog("Enter key to search:"));
                    int linearResult = linearSearch(array, linearKey);
                    if (linearResult !=  -1) {
                        outputArea.setText("Found the number " + linearKey +
" at: " + linearResult);
                    } else {
                        outputArea.setText("Could not find the number " +
linearKey + " in the array!");
                    }
                    break;
            }
        }
    }

    // Algorithm implementations remain unchanged...
    private void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);
            merge(array, left, middle, right);
        }
    }

    private void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;

        int[] L = new int[n1];
        int[] R = new int[n2];

        System.arraycopy(array, left, L, 0, n1);
        System.arraycopy(array, middle + 1, R, 0, n2);
```

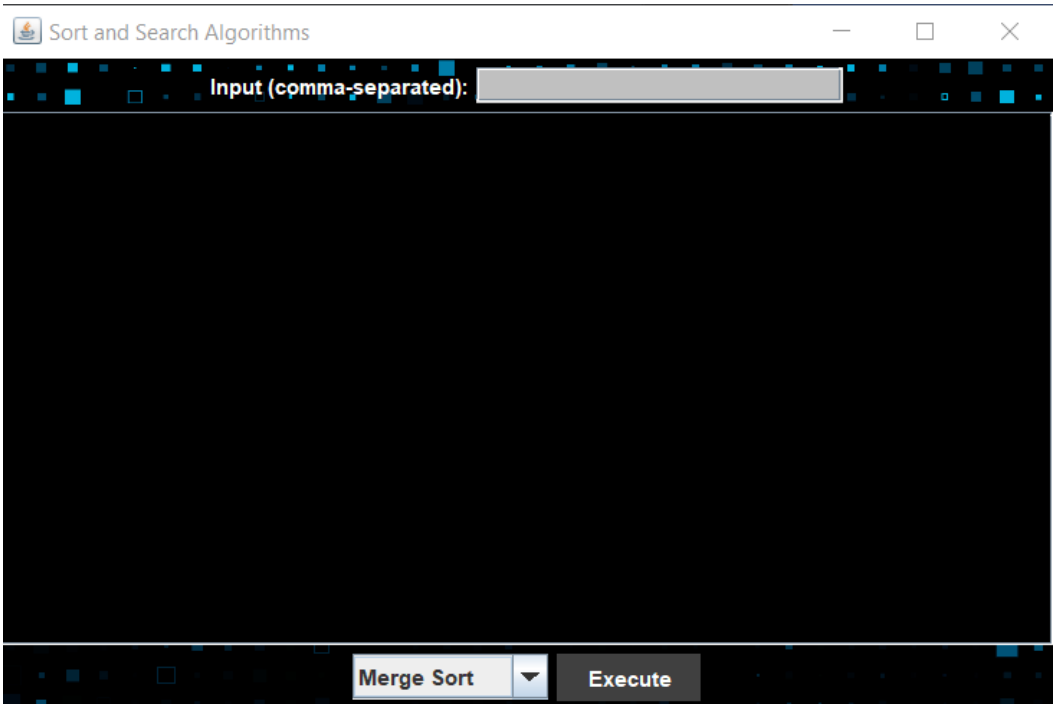
```
        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            } else {
                array[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }

    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;
                }
            }
        }
    }

    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, i, 0);
        }
    }

    private void heapify(int[] array, int n, int i) {
        int largest = i;
```

```
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && array[left] > array[largest]) {
            largest = left;
        }

        if (right < n && array[right] > array[largest]) {
            largest = right;
        }

        if (largest != i) {
            int swap = array[i];
            array[i] = array[largest];
            array[largest] = swap;
            heapify(array, n, largest);
        }
    }

    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);
        }
    }

    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;
        return i + 1;
    }

    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;
        }
    }
```

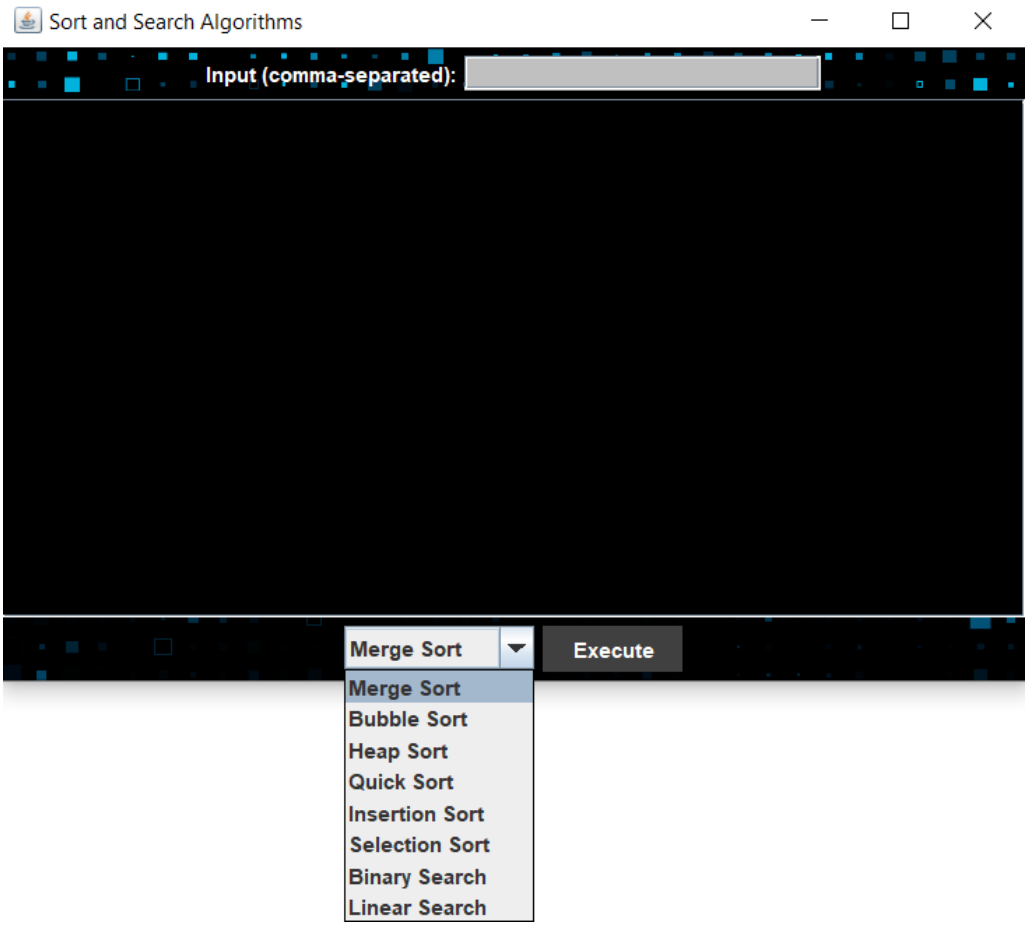
```
    private void selectionSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            int minIdx = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIdx]) {
                    minIdx = j;
                }
      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

    private int binarySearch(int[] array, int key) {
        Arrays.sort(array);  // Binary search requires sorted array
        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;
    }

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

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            Main frame = new Main();
            frame.setVisible(true);
        });
    }
}
```

GUI:

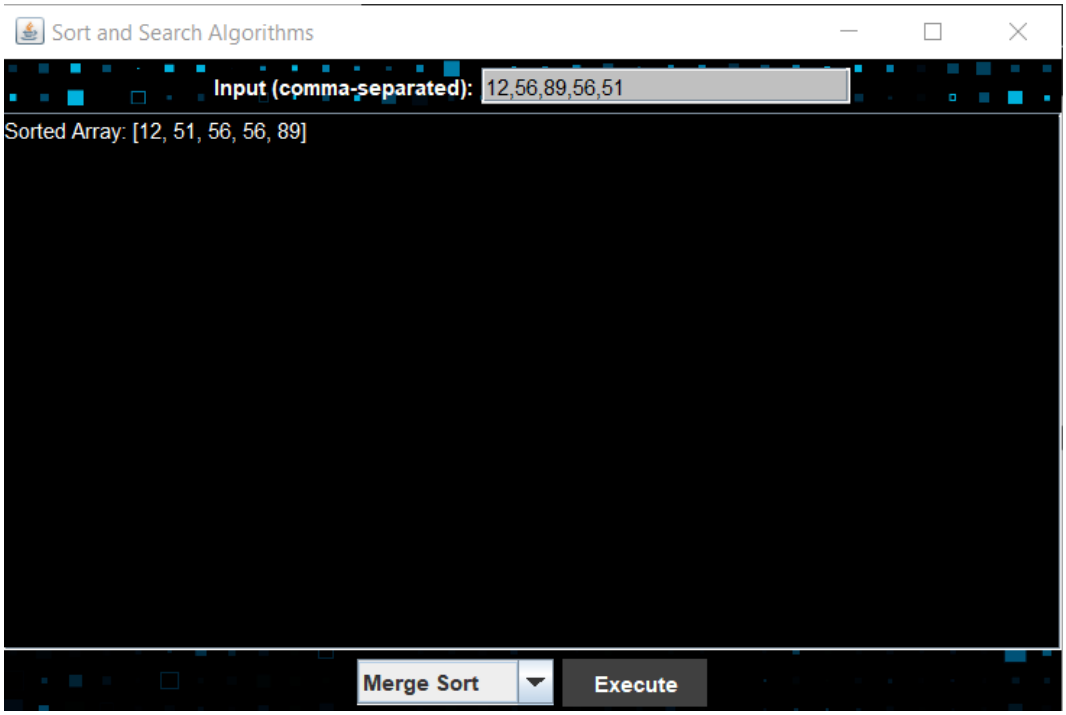
- The Main Page:



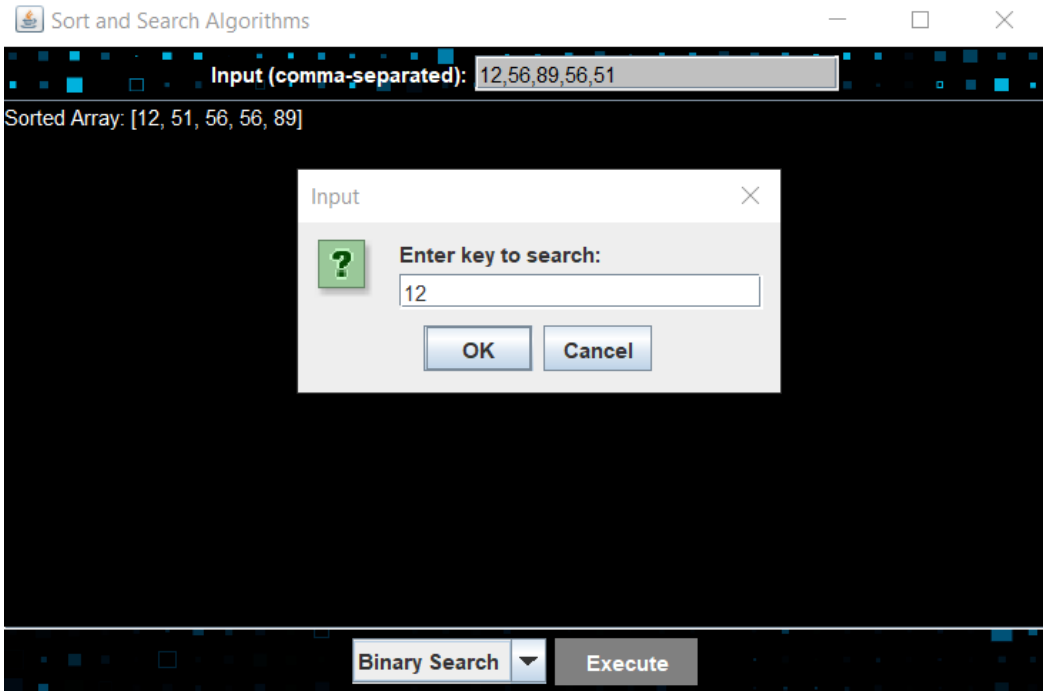
- Here You Select the Type of Sorting or Searching:



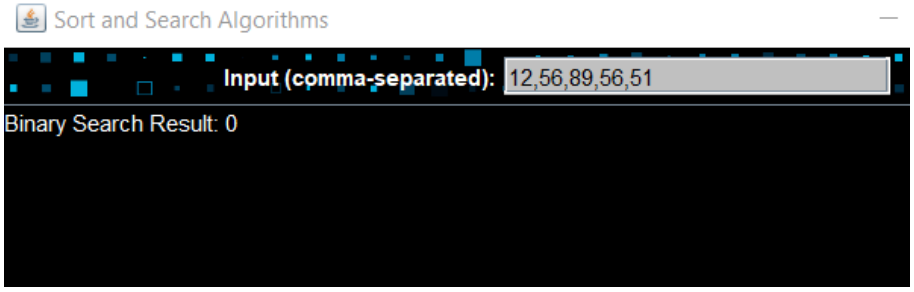
- If You Enter Array of Numbers and Use Any Type of Sorting:



- If You Enter Array of Numbers and Use Any Type of Searching:



- Here's The result of Searching:



Conclusion:

This project has provided an in-depth exploration of various sorting and searching algorithms, demonstrating their implementation and comparing their performance. By working with different algorithms, we have gained valuable insights into their strengths and weaknesses in various contexts.

Key Takeaways

1. **Sorting Algorithms**:
    - **Selection Sort** and **Bubble Sort** are easy to understand and implement but are generally inefficient for large datasets due to their high time complexity (O(n^2)).
    - **Merge Sort** and **Quick Sort** are more efficient for large datasets. Merge Sort guarantees O(n log n) time complexity, while Quick Sort often performs well in practice despite its worst-case O(n^2) complexity.
    - **Heap Sort** provides a good balance between time complexity and space usage, maintaining O(n log n) time complexity.
    - **Insertion Sort** is efficient for small datasets or nearly sorted arrays due to its simplicity and average-case O(n^2) time complexity.
2. **Searching Algorithms**:
    - **Linear Search** is simple and effective for small or unsorted datasets but becomes inefficient with larger arrays due to its O(n) time complexity.
    - **Binary Search** is highly efficient for sorted arrays, reducing the time complexity to O(log n). However, it requires the array to be sorted beforehand.

Performance Analysis

Through this project, we have observed that the choice of algorithm significantly impacts the performance of data processing tasks. For small datasets, simple algorithms like Insertion Sort and Linear Search may suffice. However, for larger datasets, more efficient algorithms like Merge Sort, Quick Sort, and Binary Search are preferable to optimize performance.

Thank You!