SORTING TECHNIQUES REPORT

Prepared for

Eng/ Kirollos Victor

T.A., Data Structures and Algorithms

Faculty Of Computing and Data Science

Alexandria University

By

Maryam Ibrahim 23012132

Basmala Moataz 23012111

May 2024

Introduction:

Sorting is a foundational operation in computer science, essential for organizing data in a meaningful way. In this report, we'll explore three common sorting techniques: Bubble Sort, Quick Sort, and Counting Sort.

Bubble Sort, though not the most efficient, is straightforward to understand, making it a common starting point for learning about sorting algorithms.

Quick Sort, on the other hand, is known for its speed and efficiency, dividing data into smaller segments for sorting.

Lastly, Counting Sort is excellent for sorting integers within a specific range, offering linear time complexity.

In this report, we will include a coding example demonstrating the implementation of these three techniques. Additionally, we will assess their efficiency by comparing various factors such as time taken, number of swaps, and comparisons made during the sorting process.

Experimental Setup:

To conduct our analysis, we implemented the three sorting algorithms in Java and tested them on three different types of arrays:

1. Inversely sorted array: An array with elements sorted in descending order.
2. Sorted array: An array with elements already sorted in ascending order.
3. Shuffled array: An array with elements randomly shuffled.

Each sorting algorithm was executed on all three types of arrays, and the time taken for sorting, number of swaps, and comparisons made during the sorting process were recorded for each case.

Result Analysis:

1. Bubble Sort:

Bubble Sort demonstrates poor performance on inversely sorted and shuffled arrays due to its quadratic time complexity.

However, it performs well on already sorted arrays with minimal interchanges.

```
Bubble sort:

Inversly Array:
interchanges=49995000
comparisons=99990000
Time taken: 101.53169 milliseconds

Sorted Array:
interchanges=0
comparisons=99990000
Time taken: 27.9618 milliseconds

Shuffelled Array:
interchanges=25155081
comparisons=99990000
Time taken: 118.9538 milliseconds
```

2. Quick Sort:

Quick Sort exhibits consistent performance across all types of arrays, with significantly faster sorting times compared to Bubble Sort. In addition, it requires less interchanges and comparisons for inversely sorted and shuffled arrays.

```
Quick sort:

Inversly Array:
Interchanges = 5000
Comparisons = 9999
Time taken: 1.4421 milliseconds

Sorted Array:
Interchanges = 4999
Comparisons = 9999
Time taken: 0.3494 milliseconds

Shuffelled Array:
Interchanges = 7239
Comparisons = 9999
Time taken: 0.4043 milliseconds
```

3. Counting Sort:

Counting Sort demonstrates exceptional performance with linear time complexity, regardless of the input array's order.

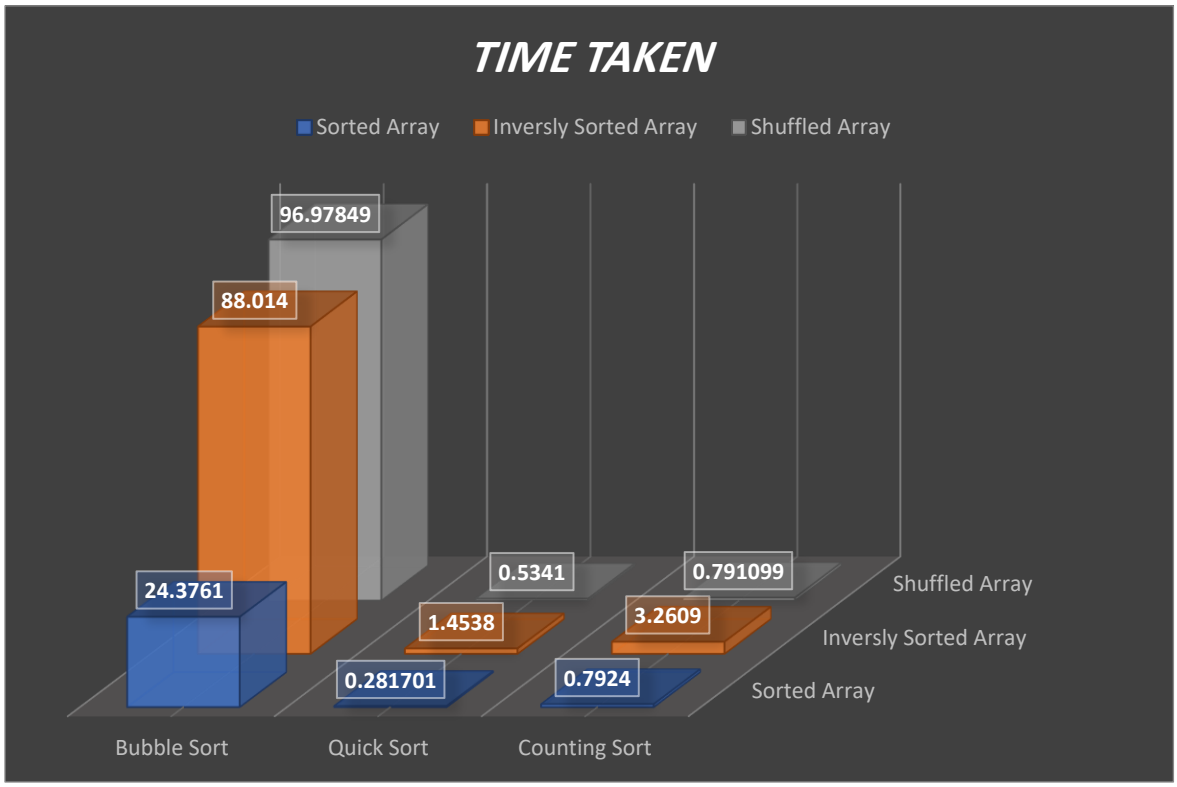
It requires no interchanges and performs a fixed number of comparisons, making it highly efficient for certain types of data.

```
Counting sort:

Inversly Array:
Time taken: 3.1163 milliseconds

Sorted Array:
Time taken: 0.9397 milliseconds

Shuffelled Array:
Time taken: 1.0792 milliseconds
```

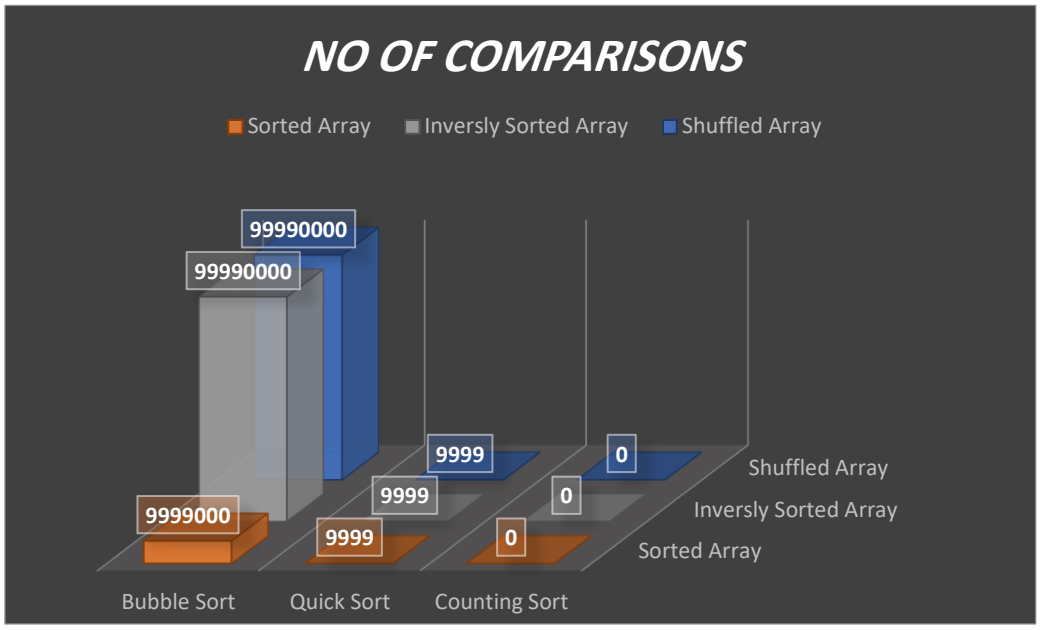
Visualizing the results:

In our report on the performance evaluation of various sorting techniques, we utilize bar plots to represent the comparative analysis of each algorithm's efficiency visually. Bar plots offer a clear and concise method of illustrating the runtime, number of comparisons, and number of interchanges of sorting algorithms, providing a better understanding of each technique's performance. Each bar in the plot represents the execution time, number of comparisons, and number of interchanges of a specific sorting algorithm under different scenarios, such as sorting already sorted data, inversely sorted data, and randomly shuffled data. By using bar plots, we aim to facilitate an easier understanding of each sorting technique's computational efficiency and scalability, aiding in informed decision-making for algorithm selection in diverse contexts.

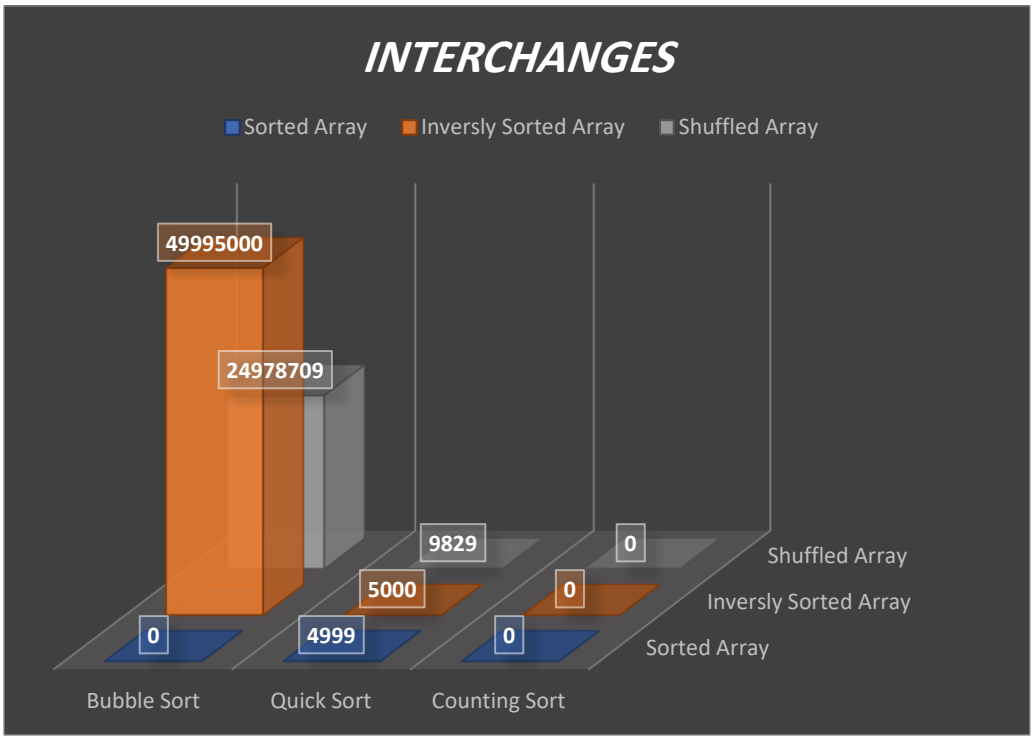
1. Execution time required in each technique:



2. No of comparisons made before committing any changes of the positions of the values:



3. Number of Interchanging in the values' positions:



Code Analysis:

1. Array preparation methods

As known, Array are from call by reference type; which means that their literal value will be changed whenever the're placed in a method. This won't be a problem for the Sorted array because the main aim of the code is to sort arrays, but the problem occurs when dealing with the inversely sorted and the shuffled array. Therefore; we implemented methods that recalculate the value of the arrays after each sorting technique for the process to be valid.

a. Implementing the inversely sorted array:

The inverseArray method takes an integer array as input and inverses the array by iterating over the array in reverse order and setting the index element of the array to i.

```
public static void inverseArray(int [] array) {
    int index =0;
    for(int i=array.length; i>0 ; i--) {
        array[index]=i;
        index++;
    }

}
```

b. Implementing the Sorted shuffled array:

the shuffled array method is implemented with two steps:

1. It is initialized with the sorted method in the same for loop, which place elements in the array equal to the index it's placed in.

2. The created array will be place in the shuffleArray method. This method takes an integer array as input and shuffles the array by swapping each element with a random element in the array. The method uses the random class to generate random indices.

```
for(int i=0; i<sortedArray.length ; i++) {
    sortedArray[i]=i;
    shuffledArray[i]=i;
}
```

```
public static void shuffleArray(int[] array) {
    Random rnd = new Random();
    for (int i = array.length - 1; i > 0; i--) {
        int index = rnd.nextInt(i + 1);

        int temp = array[i];
        array[i] = array[index];
        array[index] = temp;
    }
}
```

2. Sorting techniques methods implementation:

a. Bubble Sort:

The BubbleSort method takes an integer array as input and sorts the array using the Bubble Sort algorithm. The Bubble Sort algorithm sorts an array by repeatedly iterating over the array and swapping adjacent elements if they are in the wrong order. The method measures the number of comparisons and interchanges made during the sorting process.

```
public static void BubbleSort(int[] arr) {
    int comparisons=0;
    int interChanges=0;
    for (int pars = 0; pars < arr.length; pars++ )
        for (int i = 0; i < arr.length - 1; i++ ) {
            comparisons++;
            if ( arr[ i ] > arr[ i + 1 ] ) {
               int temp = (int) arr[i];
               arr[i] = arr[i + 1];
               arr[i + 1] = temp;
               interChanges++;
            }
        }

    System.out.println("interchanges=" + interChanges + "\ncomparisons=" + comparisons);

}
```

The method has two nested loops. The outer loop iterates over the array, while the inner loop compares adjacent elements and swaps them if they are in the wrong order. The method also prints the number of comparisons and interchanges made during the sorting process.

b. Quick sort:

The QuickSort method takes an integer array and two indices as input and sorts the array using the QuickSort algorithm. The Quick Sort algorithm sorts an array by selecting a pivot element and partitioning the array into two subarrays, one with elements less than the pivot and one with elements greater than the pivot. The method measures the number of comparisons made during the sorting process.

```
public static void QuickSort(int [] arr,int low,int high) {
    int comparisons=0;
    int interChanges=0;
    if (low < high) {
    int mid =(int) Math.round(low+((high-low)/2));
    int pivot =arr[mid];

    arr[mid]=arr[high];
    arr[high]=pivot;

    int pointer =low-1;
    for(int i=low;i<high-low;i++) {
        comparisons++;
        if(arr[i]<pivot) {
            pointer++;
            int value =arr[pointer];
            arr[pointer]=arr[i];
            arr[i]=value;
            interChanges++;

        }
    }

    arr[high]=arr[pointer+1];
    arr[pointer+1]=pivot;
    QuickSort(arr, low,pointer);
    QuickSort(arr,pointer+2, high);

    }

    if (low == 0 && high == arr.length - 1) {
        System.out.println("Interchanges = " + interChanges + "\nComparisons = " + comparisons);
    }
}
```

The method selects the pivot element as the middle element of the subarray and partitions the array using the Lomuto partition scheme. The method recursively sorts the two subarrays. The method also prints the number of comparisons made during the sorting process.

c. Counting Sort:

```
public static void CountingSort(int [] arr) {
    int min= Arrays.stream(arr).min().orElse(0);
    int max= Arrays.stream(arr).max().orElse(Integer.MAX_VALUE);
    int [] countArray = new int[max-min+1];

    for(int value:arr) {
        countArray[value-min]++;
    }
    int arrayIndex=0;
    for(int i=0; i< max-min +1;i++) {
        while(countArray[i]>0) {
            arr[arrayIndex]=i+min;
            countArray[i]--;
            arrayIndex++;
        }
    }

}
```

The CountingSort method takes an integer array as input and sorts the array using the Counting Sort algorithm.

The Counting Sort algorithm sorts an array of integers by counting the number of occurrences of each integer and using the counts to construct a sorted array.

The method first calculates the minimum and maximum values in the input array (so that if the minimum number is not equal to 0, the minimum number will be the first index and the rest of the array will be sorted depending on the shown equation).

The method then initializes a count array of size equal to the maximum value plus one. The method iterates over the input array and increments the count array at the index corresponding to each element. It then iterates over the count array and constructs the sorted array by placing each integer the number of times it occurs in the input array.

3. Testing:

After implementing the base that we're going to work with, we've come to the part where we'll combine all the methods to see how everything will interact together.

a. defining the attributes to be used and initializing their values:

```
public static void main(String[] args) {

    int[] inverslyArray= new int[10000];
    int[] shuffledArray= new int[10000];
    int[] sortedArray= new int[10000];
    int startTime;
    int endTime;
    float finalTime;

    for(int i=0; i<sortedArray.length ; i++) {
        sortedArray[i]=i;
        shuffledArray[i]=i;
    }


    int elements=1;

    while(elements<=3) {
        shuffleArray(shuffledArray);
        inverseArray(inverslyArray);
```

Firstly, we have initialized 3 types of arrays that will be used in the comparison of the 3 sorting techniques. Then, we declared their values using the previously defined methods for inversely and shuffled arrays in the while loop so that their values after sorting would return to their original form before being used in the next sorting algorithm.

Then, we've initialized 3 int attributes start time, end time, and final time which'll be used later in calculating the time taken by each algorithm to sort each array type.

b. Switch cases:

This block of code is a switch statement that sorts and times three arrays (inverslyArray, sortedArray, and shuffledArray) using three different sorting algorithms (Bubble Sort, Quick Sort, and Counting Sort). The elements variable determines which sorting algorithm to use.

If the element variable is 1, the code sorts the arrays using Bubble Sort.

If the element variable is 2, the code sorts the arrays using Quick Sort.

If the element variable is 3, the code sorts the arrays using Counting Sort.

For each sorting algorithm, the code performs the following steps:

1. Print the name of the sorting algorithm.
2. Sort the inverslyArray and print the time taken.
3. Sort the sortedArray and print the time taken.
4. Sort the shuffledArray and print the time taken.
5. Print a separator line (for separating the output, not necessary).

The startTime and endTime variables are used to measure the time taken to sort each array. The finalTime variable is calculated as the difference between endTime and startTime, divided by 1,000,000 to convert nanoseconds to milliseconds (the milliseconds method didn't work correctly with algorithms taking a short time, therefore we had to use the nano-second method and then convert it into milli-second).

```
switch(elements) {
    case(1):
        System.out.println("Bubble sort:\n");
        System.out.println("Inversly Array:");
        startTime = (int) System.nanoTime();
        BubbleSort(inverslyArray);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;
        System.out.println("Time taken: " + finalTime + " milliseconds \n");

        System.out.println("Sorted Array:");
        startTime = (int) System.nanoTime();
        BubbleSort(sortedArray);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;
        System.out.println("Time taken: " + finalTime + " milliseconds \n");

        System.out.println("Shuffelled Array:");
        startTime = (int) System.nanoTime();
        BubbleSort(shuffledArray);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;
        System.out.println("Time taken: " + finalTime + " milliseconds \n"+
                           "\n ----------------------");
    break;

    case(2):
        System.out.println("Quick sort:\n");
        System.out.println("Inversly Array:");
        startTime = (int) System.nanoTime();
        QuickSort(inverslyArray,0,9999);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;
        System.out.println("Time taken: " + finalTime + " milliseconds \n");

        System.out.println("Sorted Array:");
        startTime = (int) System.nanoTime();
        QuickSort(sortedArray,0,9999);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;
        System.out.println("Time taken: " + finalTime + " milliseconds \n");

        System.out.println("Shuffelled Array:");
        startTime = (int) System.nanoTime();
        QuickSort(shuffledArray,0,9999);
        endTime = (int) System.nanoTime();
        finalTime = (endTime - startTime)/ 1_000_000.0f;

         case(3):
            System.out.println("Counting sort: \n");
            System.out.println("Inversly Array:");
            startTime = (int) System.nanoTime();
            CountingSort(inverslyArray);
            endTime = (int) System.nanoTime();
            finalTime = (endTime - startTime)/ 1_000_000.0f;
            System.out.println("Time taken: " + finalTime + " milliseconds \n");

            System.out.println("Sorted Array:");
            startTime = (int) System.nanoTime();
            CountingSort(sortedArray);
            endTime = (int) System.nanoTime();
            finalTime = (endTime - startTime)/ 1_000_000.0f;
            System.out.println("Time taken: " + finalTime + " milliseconds \n");

            System.out.println("Shuffelled Array:");
            startTime = (int) System.nanoTime();
            CountingSort(shuffledArray);
            endTime = (int) System.nanoTime();
            finalTime = (endTime - startTime)/ 1_000_000.0f;
            System.out.println("Time taken: " + finalTime + " milliseconds \n"+
                               "\n ----------------------");
            break;
 }
 elements++;
}
```

The BubbleSort, QuickSort, and CountingSort methods are called to sort the arrays using the corresponding sorting algorithm.

After the switch statement, the elements variable is incremented to select the next sorting algorithm in the next iteration of the loop.

Bonus:

1) Code Implementation

a- Libraries used

```
package SortingTechniques;

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

- **import javax.swing.*;** : Imports the Swing package, which provides classes for creating GUI components.
- **import java.awt.*;** : Imports the Abstract Window Toolkit (AWT) package, which provides basic GUI functionality.
- **import java.awt.event.ActionEvent;** : Imports the ActionEvent class from the AWT package, which represents an event triggered by user action.
- **import java.util.Random;** : Imports the Random class from the java.util package, which allows generating random numbers.

```
public class Sorting extends JFrame {

    private final int[] numbers;
    private final JPanel panel;
    private final Random random;

    public Sorting() {
        super("Sorting Algorithm Visualization");
        random = new Random();
        numbers = new int[10000]; // Array size
```

- **private final int[] numbers;** : Declares an array named **numbers** to store the data to be sorted.
- **private final JPanel panel;** : Declares a JPanel named **panel** to display the sorting visualization.
- **private final Random random;** : Declares a Random object named **random** for generating random numbers.
- **public Sorting() {**: Constructor method for the **Sorting** class.
- **super("Sorting Algorithm Visualization") ;**: Calls the superclass constructor (JFrame) with the title "Sorting Algorithm Visualization".
- **random = new Random();**: Initializes the **random** object.
- **numbers = new int[10000];**: Initializes the **numbers** array with a size of 10000.

```
panel = new JPanel() {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        int width = getWidth() / numbers.length;
        width = Math.max(1, width); // Ensure bar width is at least 1
        int height = getHeight();
        for (int i = 0; i < numbers.length; i++) {
            int barHeight = (int) ((double) numbers[i] / numbers.length * height);
            g.setColor(Color.PINK);
            g.fillRect(i * width, height - barHeight, width, barHeight);
        }
    }
};
```

- **panel = new JPanel() { ... };** : Initializes the **panel** variable with an anonymous inner class extending JPanel.
- **@Override protected void paintComponent(Graphics g) { ... }**: Overrides the **paintComponent** method of JPanel to customize the rendering of the panel.
- **int width = getWidth() / (numbers.length / 2);** : Calculates the width of each bar in the visualization.
- **width = Math.max(6, width);** : Ensures that the width of each bar is at least 6 pixels.
- **int height = getHeight();** : Retrieves the height of the panel.
- **for (int i = 0; i < numbers.length; i++) { ... }**: Iterates over the elements in the **numbers** array.
- **int barHeight = (int) ((double) numbers[i] / numbers.length * height);** : Calculates the height of the current bar based on its value.
- **g.setColor(Color.BLUE);** : Sets the color of the graphics object to blue.
- **g.fillRect(i * width, height - barHeight, width, barHeight);**: Draws a filled rectangle representing the current bar.

```
panel.setPreferredSize(new Dimension(800, 600));
this.add(panel, BorderLayout.CENTER);

JButton bubbleSortButton = new JButton("Bubble Sort");
bubbleSortButton.addActionListener(this::startBubbleSort);
JButton quickSortButton = new JButton("Quick Sort");
quickSortButton.addActionListener(this::startQuickSort);
JButton countingSortButton = new JButton("Counting Sort");
countingSortButton.addActionListener(this::startCountingSort);
```

- **panel.setPreferredSize(new Dimension(800, 600));** : Sets the preferred size of the panel to 800x600 pixels.
- **this.add(panel, BorderLayout.CENTER) ;**: Adds the panel to the JFrame's content pane at the center position.
- **JButton bubbleSortButton = new JButton("Bubble Sort");** : Creates a JButton labeled "Bubble Sort".
- **bubbleSortButton.addActionListener(this::startBubbleSort);**: Registers an ActionListener to handle button clicks for bubble sort.
- **JButton quickSortButton = new JButton("Quick Sort");** : Creates a JButton labeled "Quick Sort".
- **quickSortButton.addActionListener(this::startQuickSort);** : Registers an ActionListener to handle button clicks for quick sort.
- **JButton countingSortButton = new JButton("Counting Sort");** : Creates a JButton labeled "Counting Sort".
- **countingSortButton.addActionListener(this::startCountingSort);** : Registers an ActionListener to handle button clicks for counting sort.

```
    JPanel buttonsPanel = new JPanel();
    buttonsPanel.add(bubbleSortButton);
    buttonsPanel.add(quickSortButton);
    buttonsPanel.add(countingSortButton);
    this.add(buttonsPanel, BorderLayout.SOUTH);

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.pack();
    this.setLocationRelativeTo(null);
    this.setVisible(true);

    initializeArray();
}
```

- **JPanel buttonsPanel = new JPanel();**: Creates a new JPanel to hold the sorting algorithm buttons.
- **buttonsPanel.add(bubbleSortButton);**: Adds the bubble sort button to the buttons panel.
- **buttonsPanel.add(quickSortButton);**: Adds the quick sort button to the buttons panel.
- **buttonsPanel.add(countingSortButton);**: Adds the counting sort button to the buttons panel.
- **this.add(buttonsPanel, BorderLayout.SOUTH);**: Adds the buttons panel to the JFrame's content pane at the bottom.
- **this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);**: Sets the default close operation of the JFrame to exit the application when closed.
- **this.pack();**: Sizes
- **this.setLocationRelativeTo(null); :**center the frame on the screen
- **this.setVisible(true); :**make the frame visible.

```
private void initializeArray() {
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] = random.nextInt(numbers.length);
    }
    repaint();
}
```

- **private void initializeArray() { ... }: t**his method initializes the numbers array with random integer values. It iterates over each index of the array and assigns a random integer using random.nextInt(numbers.length). After initializing the array, it calls repaint() to redraw the panel with the new data.

```
private void startBubbleSort(ActionEvent evt) {
    initializeArray();
    new Thread(this::bubbleSort).start();
}
```

- This method is invoked when the Bubble Sort button is clicked. It first calls **initializeArray()** to reset the array with random values. Then it starts a new thread to execute the **bubbleSort()** method, which sorts the array using the Bubble Sort algorithm.

```
private void startBubbleSort(ActionEvent evt) {
    initializeArray();
    new Thread(this::bubbleSort).start();
}

private void bubbleSort() {
    boolean swapped;
    for (int i = numbers.length - 1; i > 0; i--) {
        swapped = false;
        for (int j = 0; j < i; j++) {
            if (numbers[j] > numbers[j + 1]) {
                swap(j, j + 1);
                swapped = true;
            }
        }
        if (swapped) {
            repaint();
            sleep(20);
        }
    }
}
```

- This method implements the Bubble Sort algorithm to sort the **numbers** array. It iterates through the array and compares adjacent elements, swapping them if they are in the wrong order. It continues this process until the array is sorted. During each iteration where a swap occurs, it repaints the panel to visualize the sorting process and sleeps for a short duration to control the visualization speed.

- This method is triggered when the Quick Sort button is clicked. It initializes the array and starts a new SwingWorker thread to execute the **quickSort()** method, which implements the Quick Sort algorithm.

```
private void startQuickSort(ActionEvent e) {
    initializeArray();
    new SwingWorker<Void, int[]>() {
        @Override
        protected Void doInBackground() throws Exception {
            quickSort(0, numbers.length - 1);
            return null;
        }
```

Within the SwingWorker:

- doInBackground() method executes the Quick Sort algorithm recursively.
- quickSort() method partitions the array and recursively calls itself on the partitions.
- partition() method partitions the array based on a pivot element and swaps elements accordingly. It also publishes the current state of the array for visualization.
- process() method updates the array and repaints the panel to visualize the changes.

```
        private void quickSort(int low, int high) throws InterruptedException {
            if (low < high) {
                int pi = partition(low, high);
                quickSort(low, pi - 1);
                quickSort(pi + 1, high);
            }
        }

        private int partition(int low, int high) throws InterruptedException {
            int pivot = numbers[high];
            int i = (low - 1);
            for (int j = low; j < high; j++) {
                if (numbers[j] < pivot) {
                    i++;
                    swap(i, j);
                    publish(numbers.clone());
                    Thread.sleep(10);
                }
            }
            swap(i + 1, high);
            publish(numbers.clone());
            Thread.sleep(10);
            return i + 1 ;
                    }

        @Override
        protected void process(java.util.List<int[]> chunks) {
            int[] latestArray = chunks.get(chunks.size() - 1);
            System.arraycopy(latestArray, 0, numbers, 0, latestArray.length);
            repaint();
        }
    }.execute();
}
```

startCountingSort(ActionEvent e):

```
private void startCountingSort(ActionEvent e) {
    new SwingWorker<Void, int[]>() {
        @Override
        protected Void doInBackground() throws Exception {
            int max = getMax();
            int[] count = new int[max + 1];

            for (int number : numbers) {
                count[number]++;
            }

            int index = 0;
            for (int i = 0; i < count.length; i++) {
                while (count[i] > 0) {
                    numbers[index++] = i;
                    count[i]--;
                    publish(numbers.clone());
                    Thread.sleep(1);
                }
            }
            return null;
        }

        @Override
        protected void process(java.util.List<int[]> chunks) {
            int[] latestArray = chunks.get(chunks.size() - 1);
            System.arraycopy(latestArray, 0, numbers, 0, latestArray.length);
            repaint();
        }
    }.execute();
}
```

- This method is triggered when the Counting Sort button is clicked.
- It creates a new SwingWorker instance and overrides the doInBackground() method to perform the Counting Sort algorithm in the background.
- Inside doInBackground(), it calculates the maximum value (max) in the numbers array, initializes a count array to store the count of each element, counts the occurrences of each element in the numbers array, and reconstructs the sorted array based on the counts.
- The sorted array is published for visualization using publish() method.
- The process() method updates the GUI with the latest array data and repaints the panel to visualize the sorting process.

getMax():

This method calculates the maximum value in the numbers array, which is required for determining the size of the count array in the Counting Sort algorithm.

swap(int i, int j):

This method swaps the elements at indices i and j in the numbers array. It is used for array manipulation during sorting algorithms.

```
private int getMax() {
    int max = numbers[0];
    for (int num : numbers) {
        if (num > max) {
            max = num;
        }
    }
    return max;
}

private void swap(int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}

private void sleep(int millis) {
    try {
        Thread.sleep(millis);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

sleep(int millis):

This method pauses the current thread execution for the specified number of milliseconds. It is used to control the speed of visualization by introducing delays.

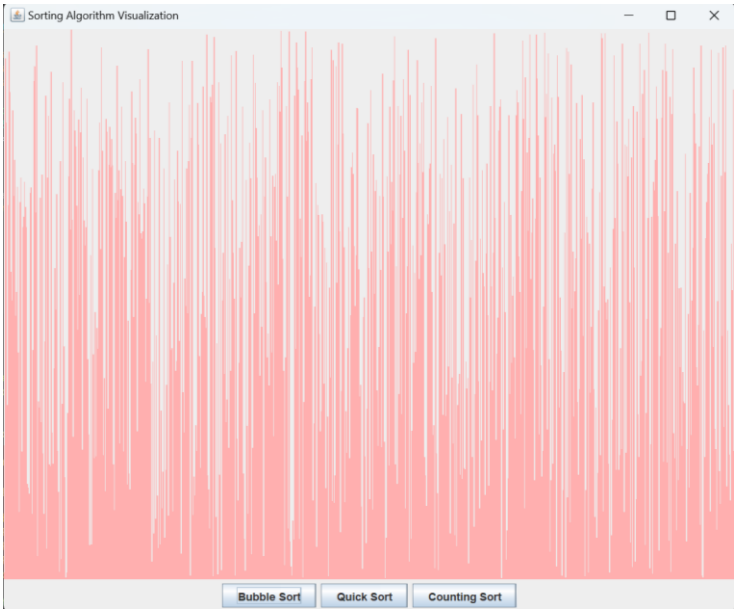
main(String[] args):

This is the entry point of the program. It creates an instance of Sorting, which initializes the GUI and starts the application.

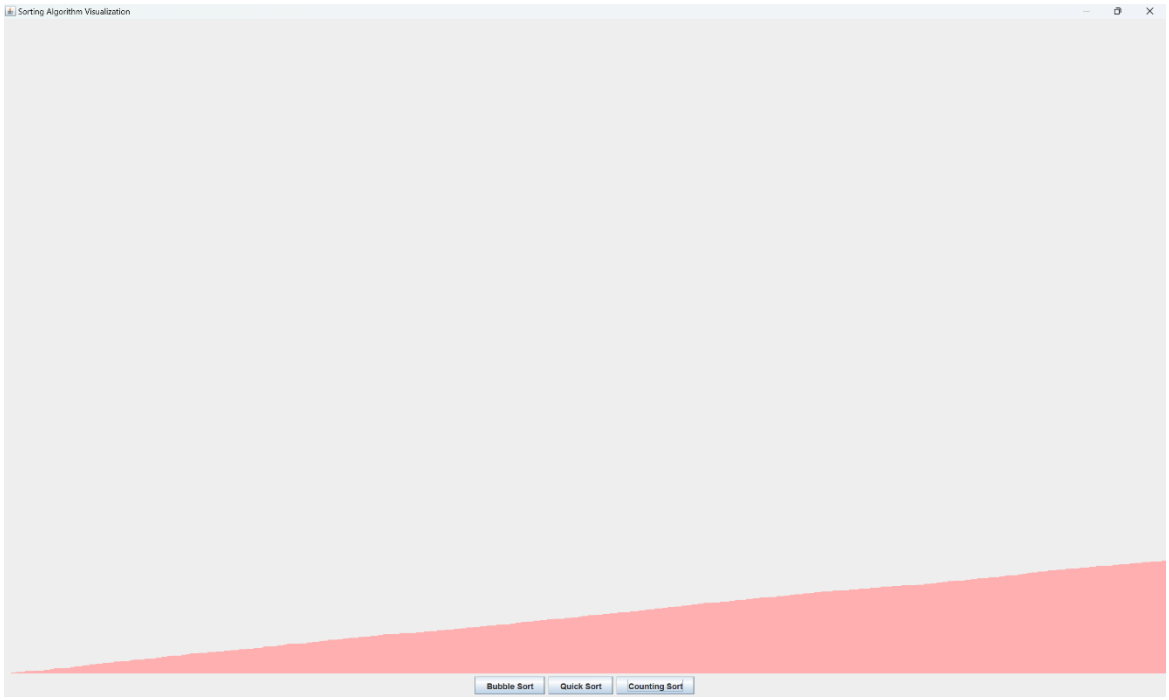
```
public static void main(String[] args) {
    new Sorting();
}
```

2) Output

Example on the array before applying any sorting alhorithm:



output after applying one of the sorting algorithms:



Conclusion

In conclusion, our analysis reveals that among the Counting Sort, Bubble Sort, and Quick Sort algorithms, Counting Sort outperforms Bubble Sort and Quick Sort in terms of time efficiency, particularly on uniformly distributed data. Quick Sort, on the other hand, exhibits better overall performance compared to Bubble Sort, especially on randomly shuffled arrays. However, the choice of sorting algorithm ultimately depends on the specific characteristics of the input data and the desired performance metrics. By understanding the strengths and weaknesses of each sorting technique, developers can make informed decisions when selecting the most suitable algorithm for their applications.

In addition, the Sorting class provides a GUI application for visualizing sorting algorithms, including Bubble Sort, Quick Sort, and Counting Sort. The program generates a random array of integers and allows users to trigger sorting algorithms through buttons. The sorting process is visually represented on a JPanel, where each bar represents an element in the array, and its height corresponds to the value of the element. The program utilizes Swing components, threading, and SwingWorker for GUI rendering and asynchronous execution of sorting algorithms. Overall, it offers an interactive platform for understanding and observing sorting algorithms in action.

Note: All measurements are based on experimental data obtained from running the provided Java code. Actual performance may vary depending on hardware specifications and other external factors.