CSN 6214- OPERATING SYSTEM ASSIGNMENT

Trimester 2430

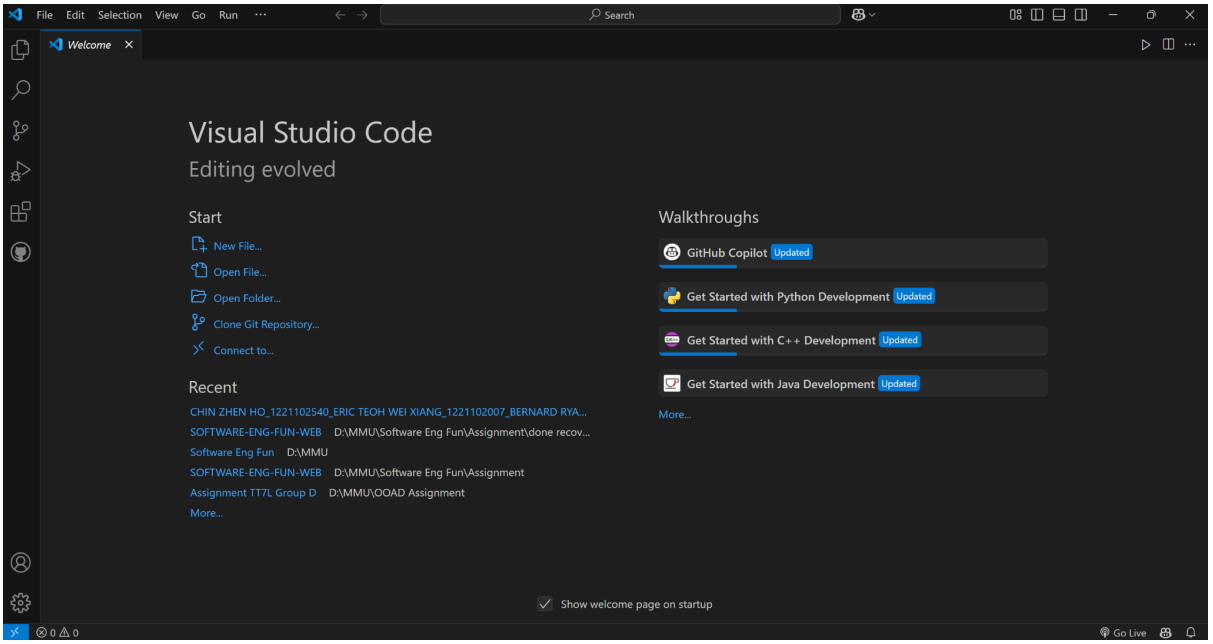
Student Name	Student ID	Email	Phone Number
CHIN ZHEN HO	1221102540	1221102540@student.mmu.edu.my	018-206 7278
ERIC TEOH WEI XIANG	1221102007	1221102007@student.mmu.edu.my	017-406 3708
BERNARD RYAN SIM KANG XUAN	1221101777	1221101777@student.mmu.edu.my	016-716 8548
GAN SHAO YANG	1221103201	1221103201@student.mmu.edu.my	01154218068

CONTENTS

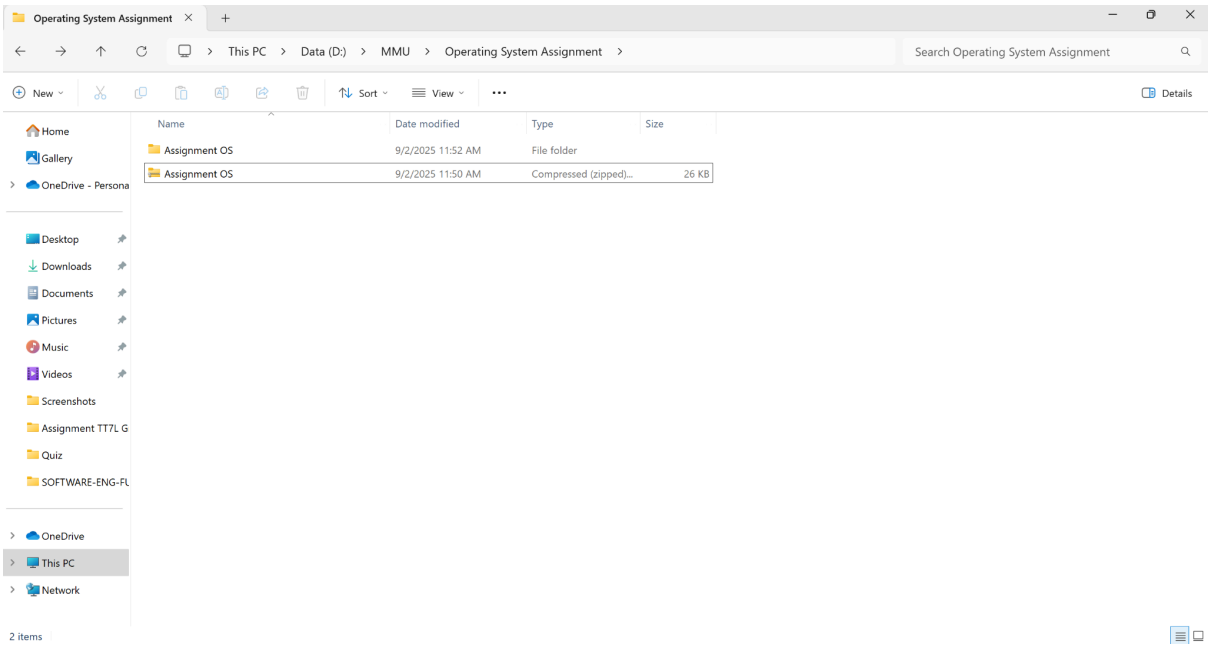
Table of contents:

1 User Manual

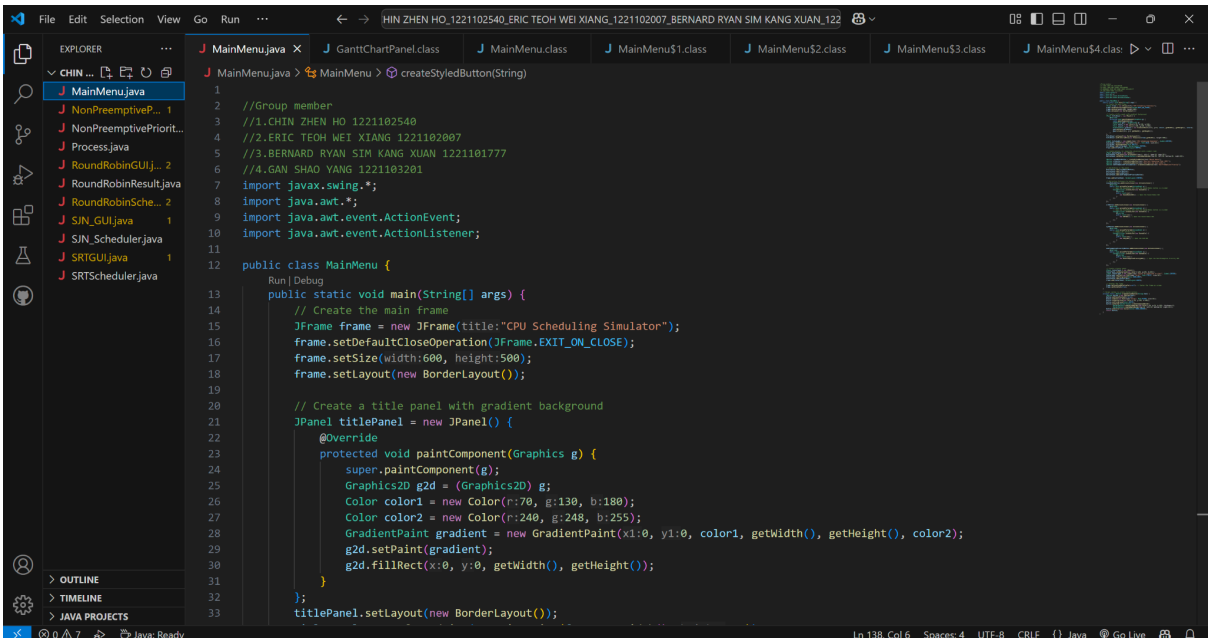
1. Open Visual Studio Code .



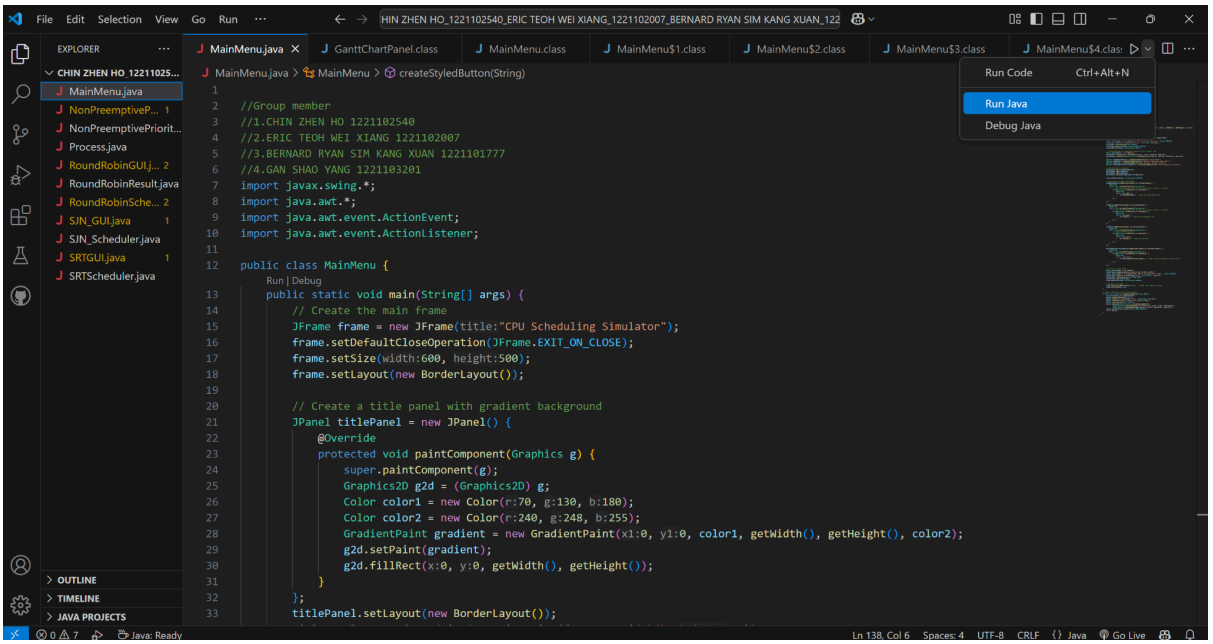
2. Extract the file zip file that you download



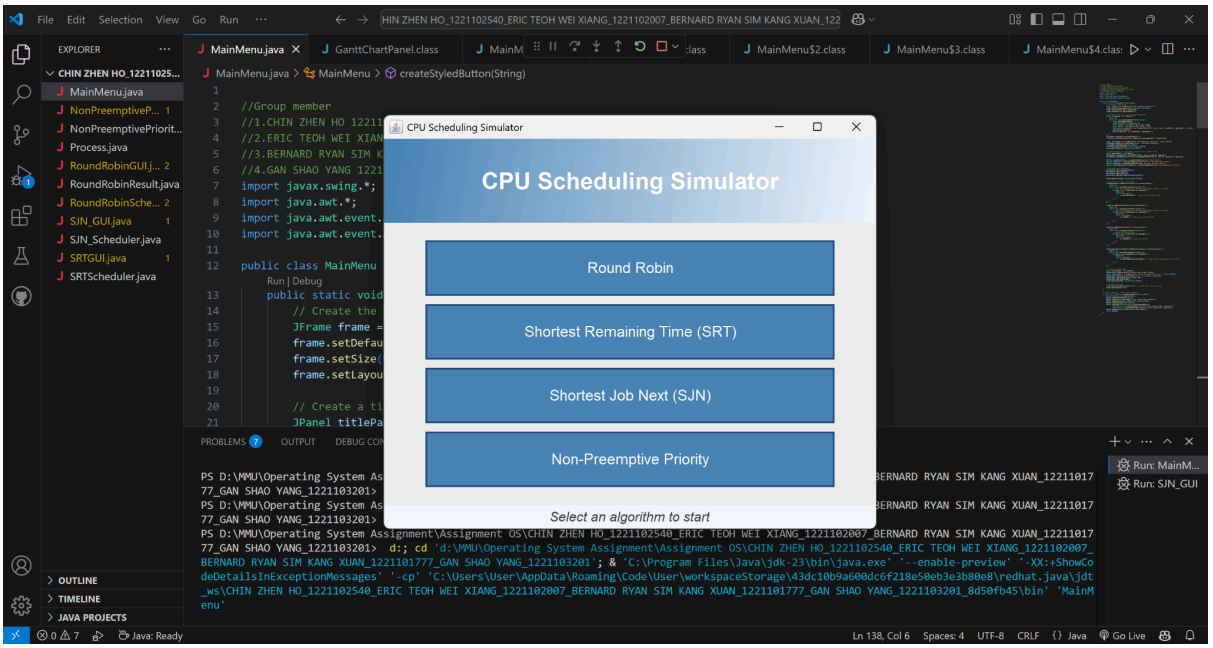
3.Use the “Open Folder” in VS Code to open the extracted file.And after you successfully open the all the .java file will show like below.



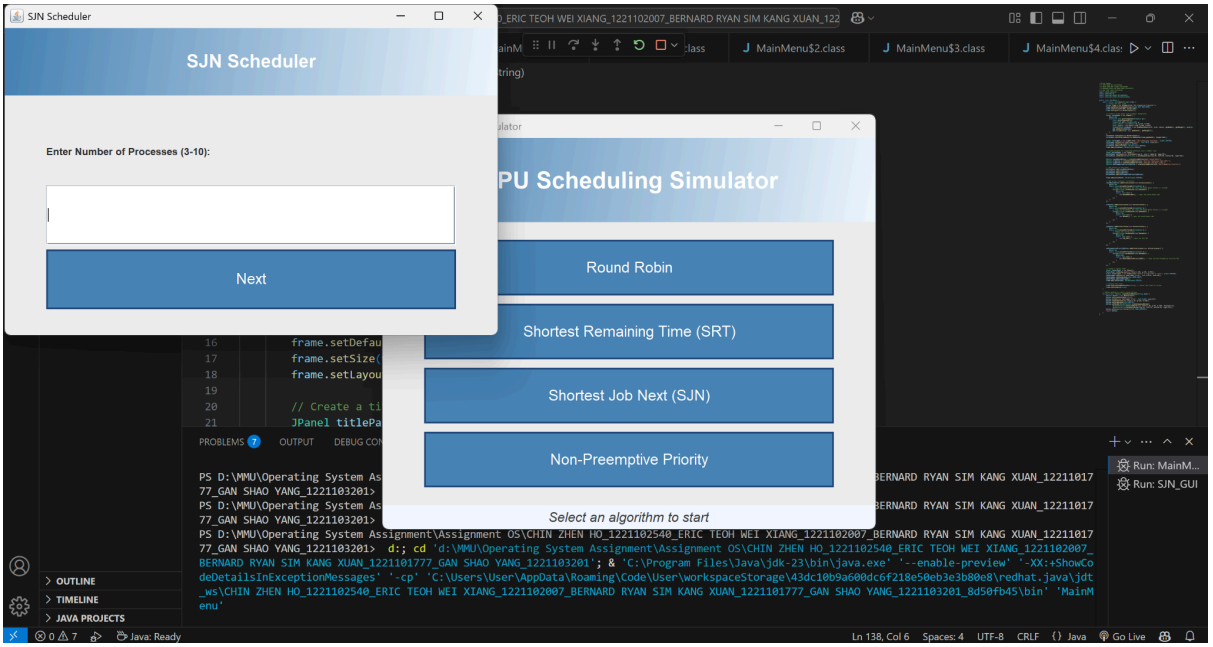
4. You must run the MainMenu.java file to go to our Main Page .



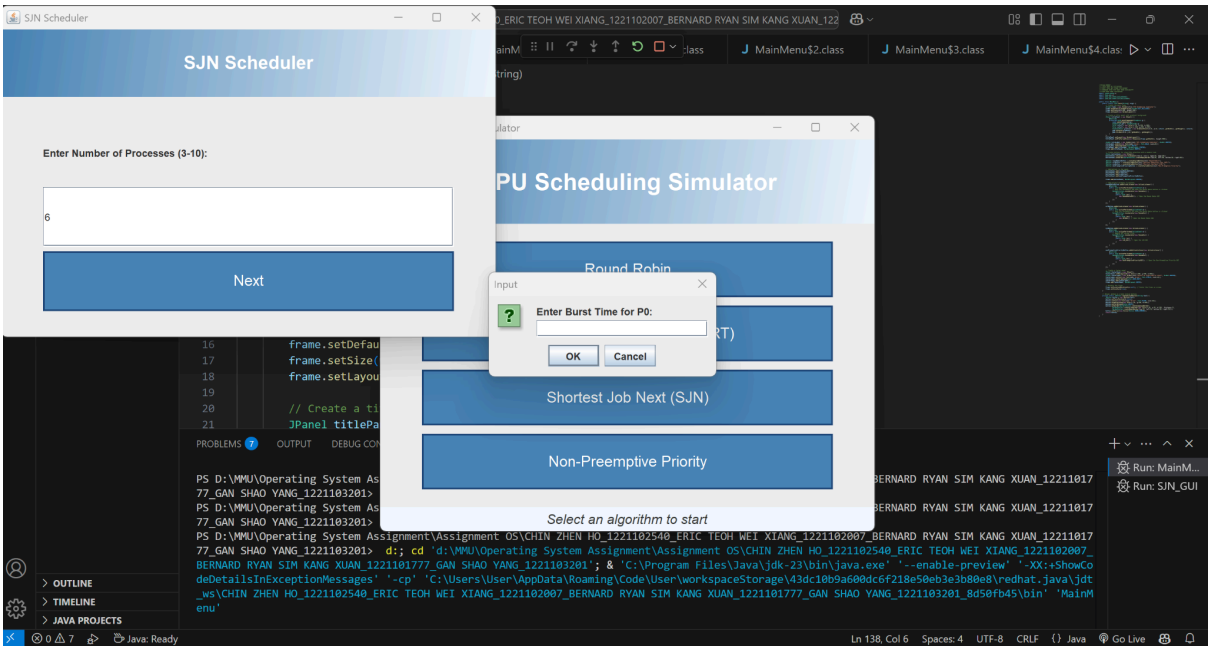
5.After successfully run the MainMenu.java file you will get the output like below screenshot.



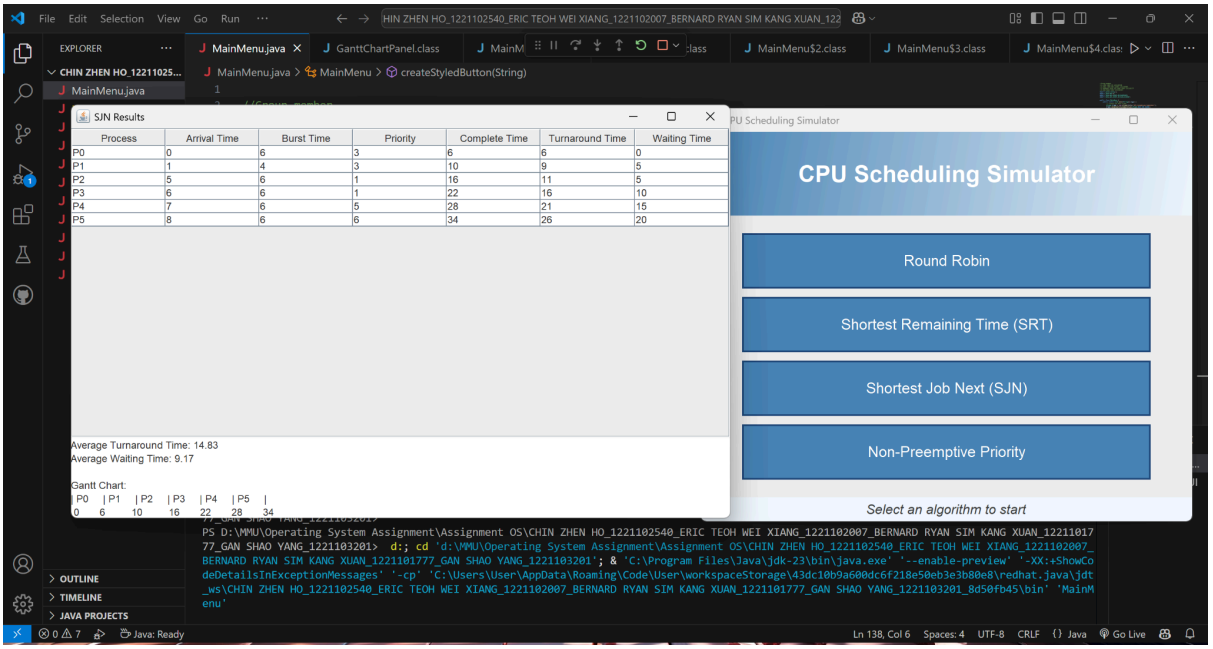
6. After that, you can choose the scheduling algorithm that you need to run . For example, I chose the Shortest Job Next (SJN).



7. Enter the number of processes and click the next button.Then, enter the burst time , arrival time and priority for very process.



8. After successfully entering all the data you will get the result like the below screenshot.



2 Code and explanation:

2.1 MainMenu.java

```
//Group member
//1.CHIN ZHEN HO 1221102540
//2.ERIC TEOH WEI XIANG 1221102007
//3.BERNARD RYAN SIM KANG XUAN 1221101777
//4.GAN SHAO YANG 1221103201
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MainMenu {
    public static void main(String[] args) {
        // Create the main frame
        JFrame frame = new JFrame("CPU Scheduling Simulator");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(600, 500);
        frame.setLayout(new BorderLayout());

        // Create a title panel with gradient background
        JPanel titlePanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2d = (Graphics2D) g;
                Color color1 = new Color(70, 130, 180);
                Color color2 = new Color(240, 248, 255);
                GradientPaint gradient = new GradientPaint(0, 0,
color1, getWidth(), getHeight(), color2);
                g2d.setPaint(gradient);
                g2d.fillRect(0, 0, getWidth(), getHeight());
            }
        };
        titlePanel.setLayout(new BorderLayout());
        titlePanel.setPreferredSize(new Dimension(frame.getWidth(),
100));
```

```
        JLabel titleLabel = new JLabel("CPU Scheduling Simulator",
JLabel.CENTER);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 28));
        titleLabel.setForeground(Color.WHITE);
        titlePanel.add(titleLabel, BorderLayout.CENTER);
        frame.add(titlePanel, BorderLayout.NORTH);

        // Create buttons for algorithm selection with a modern look
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(4, 1, 10, 10));
        buttonPanel.setBorder(BorderFactory.createEmptyBorder(20, 50,
20, 50));

        JButton roundRobinButton = createStyledButton("Round Robin");
        JButton srtButton = createStyledButton("Shortest Remaining Time
(SRT)");
        JButton sjnButton = createStyledButton("Shortest Job Next
(SJN)");
        JButton nonPreemptivePriorityButton =
createStyledButton("Non-Preemptive Priority");

        // Add buttons to the panel
        buttonPanel.add(roundRobinButton);
        buttonPanel.add(srtButton);
        buttonPanel.add(sjnButton);
        buttonPanel.add(nonPreemptivePriorityButton);

        frame.add(buttonPanel, BorderLayout.CENTER);

        // Add action listeners to buttons
        roundRobinButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Open the RoundRobin GUI when the Round Robin button
is clicked
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        new RoundRobinGUI(); // Open the Round Robin
GUI
                    }
                });
            }
```

```
        });

        srtButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Open the RoundRobin GUI when the Round Robin button
is clicked
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        new SRTGUI(); // Open the Round Robin GUI
                    }
                });
            }
        });

        sjnButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Handle SRT button click
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        new SJN_GUI(); // Open the SJN GUI
                    }
                });
            }
        });

        nonPreemptivePriorityButton.addActionListener(new
ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                SwingUtilities.invokeLater(new Runnable() {
                    @Override
                    public void run() {
                        new NonPreemptivePriorityGUI(); // Open the
Non-Preemptive Priority GUI
                    }
                });
            }
        });
```

```
        // Create a footer panel
        JPanel footerPanel = new JPanel();
        footerPanel.setBackground(new Color(240, 248, 255));
        JLabel footerLabel = new JLabel("Select an algorithm to start",
JLabel.CENTER);
        footerLabel.setFont(new Font("Arial", Font.ITALIC, 16));
        footerLabel.setForeground(Color.DARK_GRAY);
        footerPanel.add(footerLabel);
        frame.add(footerPanel, BorderLayout.SOUTH);

        // Display the frame
        frame.setLocationRelativeTo(null); // Center the frame on
screen
        frame.setVisible(true);
    }

    // Helper method to create styled buttons
    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text);
        button.setFocusPainted(false);
        button.setFont(new Font("Arial", Font.PLAIN, 18));
        button.setBackground(new Color(70, 130, 180));
        button.setForeground(Color.WHITE);
        button.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(30, 70, 120),
2),
                BorderFactory.createEmptyBorder(10, 15, 10, 15)));
        button.setCursor(new Cursor(Cursor.HAND_CURSOR));
        return button;
    }
}
```

The MainMenu.java file creates a graphical user interface (GUI) for a CPU Scheduling Simulator using Java Swing. It sets up a main frame with a gradient background title panel displaying "CPU Scheduling Simulator". It includes a button panel with styled buttons for selecting different scheduling algorithms: Round Robin, Shortest Remaining Time (SRT), Shortest Job Next (SJN), and Non-Preemptive Priority. Each button has an action listener that opens the corresponding GUI for the selected algorithm. The frame also includes a footer panel with a prompt to select an algorithm and centers the frame on the screen.

2.2 Process.java

```
public class Process {
    int processID;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int finishingTime;
    int priority; // Add priority field

    public Process(int processID, int arrivalTime, int burstTime, int
priority) {
        this.processID = processID;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.remainingTime = burstTime;
        this.priority = priority;
        this.finishingTime = 0;
    }
}
```

The Process.java file defines a Process class with attributes for processID, arrivalTime, burstTime, remainingTime, finishingTime, and priority. It includes a constructor to initialize these attributes, setting remainingTime to burstTime and finishingTime to 0. This class is used to represent and manage processes in an operating system simulation.

2.3 SJN_GUI.java

```
import javax.swing.*; // Importing Swing classes for GUI components.
import javax.swing.table.DefaultTableModel; // Importing the table
model for the JTable.
import java.awt.*; // Importing AWT classes for layout and graphics.
import java.util.ArrayList; // Importing ArrayList for dynamic array
operations.
import java.util.List; // Importing List interface for defining list
operations.

// Main class extending JFrame to create the GUI application.
public class SJN_GUI extends JFrame {

    // Constructor for the GUI application.
    public SJN_GUI() {
        // Setting the title of the application window.
        setTitle("SJN Scheduler");

        // Setting the size of the application window.
        setSize(600, 400);

        // Ensuring the application closes when the window is closed.
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        // Creating a custom panel for the title with a gradient
background.
        JPanel titlePanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                // Overriding paintComponent to add a gradient
background.
                super.paintComponent(g);
                Graphics2D gToD = (Graphics2D) g;
                // Defining the gradient colors.
                Color colorStart = new Color(70, 130, 180); // Start
color.
                Color colorEnd = new Color(240, 248, 255); // End
color.
                GradientPaint gradient = new GradientPaint(0, 0,
colorStart, getWidth(), getHeight(), colorEnd);
                gToD.setPaint(gradient);
```

```
                gToD.fillRect(0, 0, getWidth(), getHeight()); //
Filling the panel with the gradient.
            }
        };

        // Setting layout and preferred size for the title panel.
        titlePanel.setLayout(new BorderLayout());
        titlePanel.setPreferredSize(new Dimension(getWidth(), 80));

        // Adding a title label to the panel.
        JLabel titleLabel = new JLabel("SJN Scheduler", JLabel.CENTER);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 22)); //
Setting font style and size.
        titleLabel.setForeground(Color.WHITE); // Setting text color.
        titlePanel.add(titleLabel, BorderLayout.CENTER); // Centering
the label in the panel.

        // Adding the title panel to the top of the frame.
        add(titlePanel, BorderLayout.NORTH);

        // Creating the input panel for user input.
        JPanel inputPanel = new JPanel(new GridLayout(0, 1, 5, 5)); //
GridLayout for vertical alignment.
        inputPanel.setBorder(BorderFactory.createEmptyBorder(30, 50,
30, 50)); // Adding padding around the panel.

        // Creating a text field for process input.
        JTextField processField = new JTextField();

        // Adding a label and text field to the input panel.
        inputPanel.add(new JLabel("Enter Number of Processes
(3-10):"));
        inputPanel.add(processField);

        // Creating a styled button for proceeding to the next step.
        JButton nextButton = createStyledButton("Next");
        inputPanel.add(nextButton);

        // Adding the input panel to the center of the frame.
        add(inputPanel, BorderLayout.CENTER);

        // Adding an action listener to the "Next" button.
        nextButton.addActionListener(e -> {
```

```
            try {
                // Parsing the number of processes from the input
field.
                int numberProcesses =
Integer.parseInt(processField.getText());

                // Validating the range of the input (3 to 10
processes).
                if (numberProcesses < 3 || numberProcesses > 10) {
                    JOptionPane.showMessageDialog(this, "Please enter a
number of processes between 3 and 10.", "Error",
JOptionPane.ERROR_MESSAGE);
                    return; // Stop further execution if the input is
invalid.
                }

                // Creating a list to store process details.
                List<Process> processes = new ArrayList<>();

                // Looping through the number of processes to collect
input for each process.
                for (int i = 0; i < numberProcesses; i++) {
                    int burstTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Burst Time for P" +
i + ":"));
                    int arrivalTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Arrival Time for P"
+ i + ":"));
                    int priority =
Integer.parseInt(JOptionPane.showInputDialog("Enter Priority for P" + i
+ ":"));

                    // Adding the process details to the list.
                    processes.add(new Process(i, arrivalTime,
burstTime, priority));
                }

                // Creating an instance of the scheduler and scheduling
the processes.
                SJN_Scheduler scheduler = new SJN_Scheduler();
                scheduler.schedule(processes);

                // Displaying the results in a table format.
```

```
                showTableGUI(processes, scheduler);

                // Closing the current input screen after clicking
"Next".
                SwingUtilities.getWindowAncestor(nextButton).dispose();
            } catch (NumberFormatException ex) {
                // Handling invalid input and displaying an error
message.
                JOptionPane.showMessageDialog(this, "Invalid input.
Please enter numeric values.", "Error", JOptionPane.ERROR_MESSAGE);
            }
        });

        // Making the frame visible.
        setVisible(true);
    }

    // Method to display the results in a table format.
    private void showTableGUI(List<Process> processes, SJN_Scheduler
scheduler) {
        JFrame tableFrame = new JFrame("SJN Results"); // Creating a
new frame for the results.
        tableFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
// Setting the close operation.
        tableFrame.setSize(800, 500); // Setting the frame size.

        // Defining column names for the table.
        String[] columns = {"Process", "Arrival Time", "Burst Time",
"Priority", "Complete Time", "Turnaround Time", "Waiting Time"};
        DefaultTableModel model = new DefaultTableModel(columns, 0); //
Creating a table model with the columns.
        JTable table = new JTable(model); // Creating a table with the
model.

        // Populating the table with process data.
        for (Process process : processes) {
            int turnaroundTime = process.finishingTime -
process.arrivalTime; // Calculating turnaround time.
            int waitingTime = turnaroundTime - process.burstTime; //
Calculating waiting time.
            model.addRow(new Object[]{
                    "P" + process.processID,
                    process.arrivalTime,
```

```
                    process.burstTime,
                    process.priority,
                    process.finishingTime,
                    turnaroundTime,
                    waitingTime
            });
        }

        // Adding a scroll pane to the table for better display.
        JScrollPane scrollPane = new JScrollPane(table);
        tableFrame.add(scrollPane, BorderLayout.CENTER);

        // Creating a text area for additional results.
        JTextArea resultArea = new JTextArea();
        resultArea.setEditable(false); // Making the text area
non-editable.

        // Calculating and displaying average turnaround and waiting
times.
        String averageTurnaround = String.format("%.2f",
scheduler.getAverageTurnaroundTime());
        String averageWaiting = String.format("%.2f",
scheduler.getAverageWaitingTime());
        resultArea.setText("Average Turnaround Time: " +
averageTurnaround + "\n");
        resultArea.append("Average Waiting Time: " + averageWaiting +
"\n\n");
        resultArea.append("Gantt Chart:\n" +
scheduler.getFormattedGanttChart()); // Displaying the Gantt chart.

        // Adding the text area to the bottom of the frame.
        tableFrame.add(resultArea, BorderLayout.SOUTH);

        // Making the results frame visible.
        tableFrame.setVisible(true);
    }

    // Method to create a styled button.
    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text); // Creating a button with
the given text.
        button.setFocusPainted(false); // Disabling focus painting.
```

```
        button.setFont(new Font("Arial", Font.PLAIN, 18)); // Setting
the font style and size.
        button.setBackground(new Color(70, 130, 180)); // Setting the
background color.
        button.setForeground(Color.WHITE); // Setting the text color.
        button.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(30, 70, 120),
2), // Adding a line border.
                BorderFactory.createEmptyBorder(10, 15, 10, 15) //
Adding padding inside the border.
        ));
        button.setCursor(new Cursor(Cursor.HAND_CURSOR)); // Changing
the cursor to a hand icon when hovered.
        return button; // Returning the styled button.
    }

    // Main method to launch the application.
    public static void main(String[] args) {
        new SJN_GUI(); // Creating an instance of the GUI.
    }
}
```

The SJN_GUI.java file creates a graphical user interface (GUI) for the Shortest Job Next (SJN) scheduling algorithm using Java Swing. It sets up a main frame with a gradient background title panel and an input panel for the user to enter the number of processes and their details (burst time, arrival time, and priority). Upon clicking the "Next" button, it validates the input, collects process details, schedules the processes using the SJN_Scheduler, and displays the results in a table format along with average turnaround and waiting times, and a Gantt chart. The GUI also includes styled buttons and handles invalid input gracefully.

2.4 SJN_Scheduler.java

```
import java.util.*; // Importing utility classes for data structures and collections.

// Main class for the Non-Preemptive Shortest Job Next (SJN) Scheduler.
public class SJN_Scheduler {

    // Variables to store total turnaround and waiting times for all processes.
    private double totalTurnaroundTime = 0;
    private double totalWaitingTime = 0;

    // List to maintain the Gantt chart representation of the schedule.
    private final List<String> ganttChart = new ArrayList<>();

    // List to store time markers for the Gantt chart timeline.
    private final List<Integer> timeMarkers = new ArrayList<>();

    // Method to perform scheduling based on Non-Preemptive SJN algorithm.
    public void schedule(List<Process> processes) {
        int currentTime = 0; // Initialize the current time to zero.

        // Priority queue to select the process with the shortest burst time first.
        Queue<Process> readyQueue = new
PriorityQueue<>(Comparator.comparingInt((Process p) -> p.burstTime) // Compare by burst
time.
                .thenComparingInt(p -> p.priority) // If burst times are equal, compare by priority.
                .thenComparingInt(p -> p.arrivalTime)); // If priority is also equal, compare by
arrival time.

        boolean allProcessesHandled = false; // Flag to indicate if all processes are scheduled.
        timeMarkers.add(currentTime); // Add the initial start time to the timeline.

        while (!allProcessesHandled) { // Loop until all processes are handled.

            // Add processes to the ready queue if they have arrived and are not yet completed.
            for (Process process : processes) {
                if (!readyQueue.contains(process) && process.remainingTime > 0 &&
process.arrivalTime <= currentTime) {
                    readyQueue.add(process);
                }
```

```
        }

        if (!readyQueue.isEmpty()) { // If the ready queue has processes.

            // Select the process with the shortest burst time.
            Process currentProcess = readyQueue.poll();

            // Add the process to the Gantt chart.
            ganttChart.add("P" + currentProcess.processID);

            // Update the current time by adding the burst time of the selected process.
            currentTime += currentProcess.burstTime;

            // Add the current time to the time markers after executing the process.
            timeMarkers.add(currentTime);

            // Calculate finishing time, turnaround time, and waiting time for the process.
            currentProcess.finishingTime = currentTime;
            int turnaroundTime = currentProcess.finishingTime - currentProcess.arrivalTime; // Total time from arrival to completion.
            int waitingTime = turnaroundTime - currentProcess.burstTime; // Time spent waiting in the ready queue.

            // Accumulate the turnaround and waiting times.
            totalTurnaroundTime += turnaroundTime;
            totalWaitingTime += waitingTime;

            // Mark the process as completed by setting its remaining time to zero.
            currentProcess.remainingTime = 0;

        } else {
            // If no process is ready, increment the current time.
            currentTime++;

            // Add an idle time marker to the timeline.
            timeMarkers.add(currentTime);
        }

        // Check if all processes are completed.
        allProcessesHandled = processes.stream().allMatch(p -> p.remainingTime == 0);
```

```
        }
    }

    // Method to calculate and return the average turnaround time.
    public double getAverageTurnaroundTime() {
        return totalTurnaroundTime / ganttChart.size(); // Divide total turnaround time by the
number of processes.
    }

    // Method to calculate and return the average waiting time.
    public double getAverageWaitingTime() {
        return totalWaitingTime / ganttChart.size(); // Divide total waiting time by the number of
processes.
    }

    // Method to format and return the Gantt chart representation.
    public String getFormattedGanttChart() {
        StringBuilder chartLine = new StringBuilder(); // Line for process names.
        StringBuilder timeLine = new StringBuilder(); // Line for time markers.

        // Counter for two-digit numbers
        int twoDigitCount = 0;

        // Initialize Gantt chart formatting.
        chartLine.append("|");
        for (String process : ganttChart) {
            chartLine.append(String.format(" %-7s|", process)); // Format process name in a
fixed-width block.
        }

        // Append aligned time markers below the Gantt chart
        timeLine.append(" "); // Add one space before the first number
        for (int i = 0; i < timeMarkers.size(); i++) {
            int marker = timeMarkers.get(i);

            if (marker >= 10) {
                // If the marker is a two-digit number, increment the counter
                twoDigitCount++;
            }
```

```
            if (i == 0) {
                // For the first time marker, add 8 spaces
                timeLine.append(String.format("%-8d", marker));
            } else if (i == 1) {
                // For the second time marker, add 10 spaces
                timeLine.append(String.format("%-10d", marker));
            } else if (twoDigitCount >= 2) {
                // After the second two-digit number, add 9 spaces
                timeLine.append(String.format("%-9d", marker));
            } else {
                // Default for other markers (before two-digit adjustment)
                timeLine.append(String.format("%-11d", marker));
            }
        }
        // Return the Gantt chart with aligned process names and time markers.
        return chartLine.toString() + "\n" + timeLine.toString();
    }
}
```

The SJN_Scheduler.java file implements the Non-Preemptive Shortest Job Next (SJN) scheduling algorithm. It schedules processes based on their burst time, priority, and arrival time, maintaining a Gantt chart and time markers to visualize the schedule. The class calculates total and average turnaround and waiting times for all processes. It includes methods to format and return the Gantt chart representation with aligned process names and time markers, ensuring a clear visual output of the scheduling process.

2.5 SRTGUI.java

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

// Main class extending JFrame to create the SRT GUI application
public class SRTGUI extends JFrame {

    // Constructor for SRT GUI application
    public SRTGUI() {
        setTitle("Shortest Remaining Time Scheduler");// set the title
of application window

        setSize(600, 400);// set window size

        // make sure the app is close when the window is closed
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        // Create a panel for the title with a gradient background
        JPanel headerPanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                // Overriding paintComponent to add a gradient
background
                super.paintComponent(g);
                Graphics2D gTod = (Graphics2D) g;
                // Defining the gradient colors.
                Color startcolor = new Color(70, 130, 180); // Start
color
                Color endcolor = new Color(240, 248, 255); // End color
                GradientPaint gradient = new GradientPaint(0, 0,
startcolor, getWidth(), getHeight(), endcolor);
                gTod.setPaint(gradient);
                gTod.fillRect(0, 0, getWidth(), getHeight()); // Fill
the panel with the gradient
            }

        };
```

```
        // Set layout and preferred size for the title panel
        headerPanel.setLayout(new BorderLayout()); // Sets the layout
of the header panel to BorderLayout.
        headerPanel.setPreferredSize(new Dimension(getWidth(), 80));//
Sets the preferred size of the header panel.

        // Add title label to the panel.
        // Creates a new JLabel with the specified text and centers it.
        JLabel headerLabel = new JLabel("Shortest Remaining Time
Scheduler", JLabel.CENTER);

        // Sets the font of the label to Arial, bold, size 22.
        headerLabel.setFont(new Font("Arial", Font.BOLD, 22)); // Set
the font style and size

        headerLabel.setForeground(Color.WHITE); // Sets the text color
of the label to white.
        headerPanel.add(headerLabel, BorderLayout.CENTER); // Center
the label in the panel

        // Add the title panel to the top of the frame
        add(headerPanel, BorderLayout.NORTH);

        // Create the input field panel for user to input
        JPanel inputFieldPanel = new JPanel(new GridLayout(0, 1, 5,
5)); // GridLayout for vertical alignment
        inputFieldPanel.setBorder(BorderFactory.createEmptyBorder(30,
50, 30, 50)); // Add padding around the panel.

        // text field for process input.
        JTextField processInputField = new JTextField();

        // Add label and text field to the input panel
        inputFieldPanel.add(new JLabel("Enter Number of Processes
(3-10):"));
        inputFieldPanel.add(processInputField);

        // styled button for proceed to the next step
        JButton nextButton = createStyledButton("Next");
        inputFieldPanel.add(nextButton);

        // Add the input field panel to the center of the frame
```

```
        add(inputFieldPanel, BorderLayout.CENTER);

        // Add an action listener to the "Next" button
        nextButton.addActionListener(e -> {
            try {
                // Parsing the number of processes from the input field
                int numProcesses =
Integer.parseInt(processInputField.getText());

                // Validate the range of the input is 3 to 10 processes
or not
                if (numProcesses < 3 || numProcesses > 10) {
                    JOptionPane.showMessageDialog(this, "Please enter a
number of processes between 3 and 10.", "Error",
                            JOptionPane.ERROR_MESSAGE);
                    return; // if the input is invalid stop further
execution
                }

                // list to store process details
                List<Process> processes = new ArrayList<>();

                // Looping through the number of processes to take the
input for each process
                for (int i = 0; i < numProcesses; i++) {

                    // Prompts the user to enter the burst time for
each process.
                    int burstTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Burst Time for P" +
i + ":"));
                    // Prompts the user to enter the arrival time for
each process.
                    int arrivalTime = Integer

.parseInt(JOptionPane.showInputDialog("Enter Arrival Time for P" + i +
":"));
                    // Prompts the user to enter the priority for each
process.
                    int priority =
Integer.parseInt(JOptionPane.showInputDialog("Enter Priority for P" + i
+ ":"));
```

```
                    // Add the process details to the list
                    processes.add(new Process(i, arrivalTime,
burstTime, priority));
                }

                // Create an instance of the Shortest Remaining Time
Scheduler and scheduling
                // the processes
                SRTScheduler srtScheduler = new SRTScheduler();
                srtScheduler.schedule(processes);

                // Display the results in a table format
                showTableGUI(processes, srtScheduler);

                // Close the current input screen when click "Next"
                SwingUtilities.getWindowAncestor(nextButton).dispose();
            } catch (NumberFormatException ex) {
                // Handle invalid input and display error message
                JOptionPane.showMessageDialog(this, "Invalid input.
Please enter numeric values.", "Error",
                        JOptionPane.ERROR_MESSAGE);
            }
        });

        // Make the frame visible
        setVisible(true);
    }

    // Method to display the results in a table format
    private void showTableGUI(List<Process> processes, SRTScheduler
srtScheduler) {
        JFrame tableFrame = new JFrame("Shortest Remaining Time
Results"); // Create new frame for the results
        tableFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
// Set the close operation
        tableFrame.setSize(800, 500); // Set the frame size

        // Define column names for the table
        String[] columns = { "Process", "Arrival Time", "Burst Time",
"Priority", "Finishing Time", "Turnaround Time",
                "Waiting Time" };
        DefaultTableModel model = new DefaultTableModel(columns, 0); //
Create table model with the columns
```

```
        JTable processResultTable = new JTable(model); // Create a
table with the model

        // Populating the table with process data
        for (Process process : processes) {
            int turnaroundTime = process.finishingTime -
process.arrivalTime; // Calculation of turnaround time
            int waitingTime = turnaroundTime - process.burstTime; //
Calculation of waiting time
            model.addRow(new Object[] {
                    "P" + process.processID,
                    process.arrivalTime,
                    process.burstTime,
                    process.priority,
                    process.finishingTime,
                    turnaroundTime,
                    waitingTime
            });
        }

        // Add a scroll pane to the table
        JScrollPane scrollPane = new JScrollPane(processResultTable);
        tableFrame.add(scrollPane, BorderLayout.CENTER);

        // Create text area for additional results
        JTextArea resultTextArea = new JTextArea();
        resultTextArea.setEditable(false); // Make the text area
non-editable

        // Get the number of processes
        int numProcesses = processes.size();

        // Calculate and display average turnaround and waiting times
        String averageTurnaroundTimeStr = String.format("%.2f",
srtScheduler.getAverageTurnaroundTime(numProcesses));
        // Calculates and formats the average waiting time.
        String averageWaitingTimeStr = String.format("%.2f",
srtScheduler.getAverageWaitingTime(numProcesses));
        // Sets the text for the average turnaround time.
        resultTextArea.setText("Average Turnaround Time: " +
averageTurnaroundTimeStr + "\n");
        // Appends the average waiting time to the text area.
```

```
        resultTextArea.append("Average Waiting Time: " +
averageWaitingTimeStr + "\n\n");
        resultTextArea.append("Gantt Chart:\n" +
srtScheduler.getFormattedGanttChart()); // Display the Gantt chart

        // Add the text area to the bottom of the frame
        tableFrame.add(resultTextArea, BorderLayout.SOUTH);

        // Make the results frame visible
        tableFrame.setVisible(true);
    }

    // Method to create a styled button
    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text); // Create a button with the
given text
        button.setFocusPainted(false); // Disabling focus painting
        button.setFont(new Font("Arial", Font.PLAIN, 18)); // Set the
font style and size
        button.setBackground(new Color(70, 130, 180)); // Set the
background color
        button.setForeground(Color.WHITE); // Set the text color
        button.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(30, 70, 120),
2), // Add line border
                BorderFactory.createEmptyBorder(10, 15, 10, 15) // Add
padding inside the border
        ));
        button.setCursor(new Cursor(Cursor.HAND_CURSOR)); // Changing
the cursor to a hand icon when hovered
        return button; // Returning the styled button.
    }

    // Main method to run the app
    public static void main(String[] args) {
        new SRTGUI(); // Create an instance of the SRT GUI
    }
}
```

The SRTGUI.java file creates a graphical user interface (GUI) for the Shortest Remaining Time (SRT) scheduling algorithm using Java Swing. It sets up a main frame with a gradient

background title panel and an input panel for the user to enter the number of processes and their details (burst time, arrival time, and priority). Upon clicking the "Next" button, it validates the input, collects process details, schedules the processes using the SRTScheduler, and displays the results in a table format along with average turnaround and waiting times, and a Gantt chart. The GUI also includes styled buttons and handles invalid input gracefully.

2.6 SRTScheduler.java

```
import java.util.*;

// Main class for the Shortest Remaining Time (SRT) Scheduler.
public class SRTScheduler {

    // Variables to store total turnaround and waiting times for all
processes
    private double totalTurnaroundTime = 0;
    private double totalWaitingTime = 0;

    // List to maintain the Gantt chart representation of the schedule
    private final List<String> ganttChartProcesses = new ArrayList<>();

    // List to store time markers for the Gantt chart timeline
    private final List<Integer> timeMarkers = new ArrayList<>();

    // Method to perform scheduling Shortest Remaining Time (SRT)
algorithm
    public void schedule(List<Process> processes) {
        int currentTime = 0; // Initialize the current time to zero

        // Priority queue to select the process with the shortest
remaining burst time
        // first
        Queue<Process> processReadyQueue = new
PriorityQueue<>(Comparator.comparingInt((Process p) -> p.remainingTime)
                // Compare by remainning burst time
                .thenComparingInt(p -> p.arrivalTime)); // If remaining
times are equal, compare by arrival time

        boolean allProcessesHandled = false; // Flag to indicate if all
processes are scheduled
```

```
        timeMarkers.add(currentTime); // Add the initial start time to
the timeline

        // Loop until all processes are handled
        while (!allProcessesHandled) {
            // Add processes to the ready queue if they have arrived
and are not yet
            // completed
            for (Process process : processes) {
                if (!processReadyQueue.contains(process) &&
process.remainingTime > 0
                        && process.arrivalTime <= currentTime) {
                    processReadyQueue.add(process);
                }
            }

            if (!processReadyQueue.isEmpty()) { // If the ready queue
has processes

                // Select the process with the shortest remaining burst
time
                Process currentProcess = processReadyQueue.poll();

                // Add the process to the Gantt chart
                ganttChartProcesses.add("P" +
currentProcess.processID);

                // Update the current time by incrementing it by 1
                currentTime++;

                // Add the current time to the time markers after
executing the process
                timeMarkers.add(currentTime);

                // Update the remaining time of the selected process
                currentProcess.remainingTime--;

                // If the process is finished, calculate finishing
time, turnaround time, and
                // waiting time
                if (currentProcess.remainingTime == 0) {
                    currentProcess.finishingTime = currentTime;
```

```
                    int turnaroundTime = currentProcess.finishingTime -
currentProcess.arrivalTime; // Total time from

// arrival to

// completion
                    int waitingTime = turnaroundTime -
currentProcess.burstTime; // Time spent waiting in the ready

// queue

                    // calculate the turnaround and waiting times
                    totalTurnaroundTime += turnaroundTime;
                    totalWaitingTime += waitingTime;
                }

            } else {
                // If no process is ready, increment the current time
                currentTime++;

                // Add an idle time marker to the timeline
                timeMarkers.add(currentTime);
            }

            // Check if all processes are completed
            allProcessesHandled = processes.stream().allMatch(p ->
p.remainingTime == 0);
        }
    }

    // Method to calculate and return the average turnaround time
    public double getAverageTurnaroundTime(int numProcesses) {
        return totalTurnaroundTime / numProcesses; // Divide total
turnaround time by the number of processes
    }

    // Method to calculate and return the average waiting time
    public double getAverageWaitingTime(int numProcesses) {
        return totalWaitingTime / numProcesses; // Divide total waiting
time by the number of processes
    }

    // Method to format and return the Gantt chart representation
```

```
    public String getFormattedGanttChart() {
        StringBuilder chartLine = new StringBuilder(); // Line for
process names
        StringBuilder timeLine = new StringBuilder(); // Line for time
markers

        // Initialize Gantt chart formatting for process names
(fixed-width for
        // alignment)
        chartLine.append("|");
        for (String process : ganttChartProcesses) {
            chartLine.append(String.format(" %-7s|", process)); //
Format process name to take 7 characters
        }

        // Append aligned time markers below the Gantt chart
        timeLine.append(" "); // Add one space before the first number
        for (int i = 0; i < timeMarkers.size(); i++) {
            int marker = timeMarkers.get(i);

            // For one-digit and two-digit markers, use 10 spaces for
consistency
            if (marker < 10) {
                timeLine.append(String.format("%-10d", marker)); // 10
spaces for one-digit
            } else {
                timeLine.append(String.format("%-9d", marker)); // 10
spaces for two-digit markers
            }
        }

        // Return the formatted Gantt chart with aligned process names
and time markers
        return chartLine.toString() + "\n" + timeLine.toString();
    }

}
```

The SRTScheduler class implements the Shortest Remaining Time (SRT) scheduling algorithm in Java. It maintains the total turnaround and waiting times for all processes, and uses lists to represent the Gantt chart and time markers. The schedule method processes a list of Process objects, using a priority queue to select the process with the shortest remaining burst time. It updates the current time, tracks process execution in the Gantt chart,

and calculates turnaround and waiting times. The class also provides methods to calculate average turnaround and waiting times, and to format and return the Gantt chart representation.

2.7 NonPreemptivePriorityGUI.java

```
import javax.swing.*; // Importing Swing classes for GUI components.
import javax.swing.table.DefaultTableModel; // Importing the table
model for the JTable.
import java.awt.*; // Importing AWT classes for layout and graphics.
import java.util.ArrayList; // Importing ArrayList for dynamic array
operations.
import java.util.List; // Importing List interface for defining list
operations.

// Main class extending JFrame to create the GUI application.
public class NonPreemptivePriorityGUI extends JFrame {

    // Constructor for the GUI application.
    public NonPreemptivePriorityGUI() {
        // Setting the title of the application window.
        setTitle("Non-Preemptive Priority Scheduler");

        // Setting the size of the application window.
        setSize(600, 400);

        // Ensuring the application closes when the window is closed.
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        // Creating a custom panel for the title with a gradient
background.
        JPanel titlePanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                // Overriding paintComponent to add a gradient
background.
                super.paintComponent(g);
                Graphics2D gTod = (Graphics2D) g;
                // Defining the gradient colors.
                Color colorStart = new Color(70, 130, 180); // Start
color.
```

```
                Color colorEnd = new Color(240, 248, 255); // End
color.
                GradientPaint gradient = new GradientPaint(0, 0,
colorStart, getWidth(), getHeight(), colorEnd);
                gTod.setPaint(gradient);
                gTod.fillRect(0, 0, getWidth(), getHeight()); //
Filling the panel with the gradient.
            }
        };

        // Setting layout and preferred size for the title panel.
        titlePanel.setLayout(new BorderLayout());
        titlePanel.setPreferredSize(new Dimension(getWidth(), 80));

        // Adding a title label to the panel.
        JLabel titleLabel = new JLabel("Non-Preemptive Priority
Scheduler", JLabel.CENTER);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 22)); //
Setting font style and size.
        titleLabel.setForeground(Color.WHITE); // Setting text color.
        titlePanel.add(titleLabel, BorderLayout.CENTER); // Centering
the label in the panel.

        // Adding the title panel to the top of the frame.
        add(titlePanel, BorderLayout.NORTH);

        // Creating the input panel for user input.
        JPanel inputPanel = new JPanel(new GridLayout(0, 1, 5, 5)); //
GridLayout for vertical alignment.
        inputPanel.setBorder(BorderFactory.createEmptyBorder(30, 50,
30, 50)); // Adding padding around the panel.

        // Creating a text field for process input.
        JTextField processField = new JTextField();

        // Adding a label and text field to the input panel.
        inputPanel.add(new JLabel("Enter Number of Processes
(3-10):"));
        inputPanel.add(processField);

        // Creating a styled button for proceeding to the next step.
        JButton nextButton = createStyledButton("Next");
        inputPanel.add(nextButton);
```

```
        // Adding the input panel to the center of the frame.
        add(inputPanel, BorderLayout.CENTER);

        // Adding an action listener to the "Next" button.
        nextButton.addActionListener(e -> {
            try {
                // Parsing the number of processes from the input
field.
                int numProcesses =
Integer.parseInt(processField.getText());

                // Validating the range of the input (3 to 10
processes).
                if (numProcesses < 3 || numProcesses > 10) {
                    JOptionPane.showMessageDialog(this, "Please enter a
number of processes between 3 and 10.", "Error",
JOptionPane.ERROR_MESSAGE);
                    return; // Stop further execution if the input is
invalid.
                }

                // Creating a list to store process details.
                List<Process> processes = new ArrayList<>();

                // Looping through the number of processes to collect
input for each process.
                for (int i = 0; i < numProcesses; i++) {
                    int burstTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Burst Time for P" +
i + ":"));
                    int arrivalTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Arrival Time for P"
+ i + ":"));
                    int priority =
Integer.parseInt(JOptionPane.showInputDialog("Enter Priority for P" + i
+ ":"));

                    // Adding the process details to the list.
                    processes.add(new Process(i, arrivalTime,
burstTime, priority));
                }
```

```
                // Creating an instance of the Non-Preemptive Priority
Scheduler and scheduling the processes.
                NonPreemptivePriorityScheduler scheduler = new
NonPreemptivePriorityScheduler();
                scheduler.schedule(processes);

                // Displaying the results in a table format.
                showTableGUI(processes, scheduler);

                // Closing the current input screen after clicking
"Next".
                SwingUtilities.getWindowAncestor(nextButton).dispose();
            } catch (NumberFormatException ex) {
                // Handling invalid input and displaying an error
message.
                JOptionPane.showMessageDialog(this, "Invalid input.
Please enter numeric values.", "Error", JOptionPane.ERROR_MESSAGE);
            }
        });

        // Making the frame visible.
        setVisible(true);
    }

    // Method to display the results in a table format.
    private void showTableGUI(List<Process> processes,
NonPreemptivePriorityScheduler scheduler) {
        JFrame tableFrame = new JFrame("Non-Preemptive Priority
Results"); // Creating a new frame for the results.
        tableFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
// Setting the close operation.
        tableFrame.setSize(800, 500); // Setting the frame size.

        // Defining column names for the table.
        String[] columns = {"Process", "Arrival Time", "Burst Time",
"Priority", "Finishing Time", "Turnaround Time", "Waiting Time"};
        DefaultTableModel model = new DefaultTableModel(columns, 0); //
Creating a table model with the columns.
        JTable table = new JTable(model); // Creating a table with the
model.

        // Populating the table with process data.
        for (Process process : processes) {
```

```
            int turnaroundTime = process.finishingTime -
process.arrivalTime; // Calculating turnaround time.
            int waitingTime = turnaroundTime - process.burstTime; //
Calculating waiting time.
            model.addRow(new Object[]{
                    "P" + process.processID,
                    process.arrivalTime,
                    process.burstTime,
                    process.priority,
                    process.finishingTime,
                    turnaroundTime,
                    waitingTime
            });
        }

        // Adding a scroll pane to the table for better display.
        JScrollPane scrollPane = new JScrollPane(table);
        tableFrame.add(scrollPane, BorderLayout.CENTER);

        // Creating a text area for additional results.
        JTextArea resultArea = new JTextArea();
        resultArea.setEditable(false); // Making the text area
non-editable.

        // Calculating and displaying average turnaround and waiting
times.
        String averageTurnaround = String.format("%.2f",
scheduler.getAverageTurnaroundTime());
        String averageWaiting = String.format("%.2f",
scheduler.getAverageWaitingTime());
        resultArea.setText("Average Turnaround Time: " +
averageTurnaround + "\n");
        resultArea.append("Average Waiting Time: " + averageWaiting +
"\n\n");
        resultArea.append("Gantt Chart:\n" +
scheduler.getFormattedGanttChart()); // Displaying the Gantt chart.

        // Adding the text area to the bottom of the frame.
        tableFrame.add(resultArea, BorderLayout.SOUTH);

        // Making the results frame visible.
        tableFrame.setVisible(true);
    }
```

```
    // Method to create a styled button.
    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text); // Creating a button with
the given text.
        button.setFocusPainted(false); // Disabling focus painting.
        button.setFont(new Font("Arial", Font.PLAIN, 18)); // Setting
the font style and size.
        button.setBackground(new Color(70, 130, 180)); // Setting the
background color.
        button.setForeground(Color.WHITE); // Setting the text color.
        button.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(30, 70, 120),
2), // Adding a line border.
                BorderFactory.createEmptyBorder(10, 15, 10, 15) //
Adding padding inside the border.
        ));
        button.setCursor(new Cursor(Cursor.HAND_CURSOR)); // Changing
the cursor to a hand icon when hovered.
        return button; // Returning the styled button.
    }

    // Main method to launch the application.
    public static void main(String[] args) {
        new NonPreemptivePriorityGUI(); // Creating an instance of the
GUI.
    }
}
```

The NonPreemptivePriorityGUI.java file creates a graphical user interface (GUI) for the Non-Preemptive Priority scheduling algorithm using Java Swing. It sets up a main frame with a gradient background title panel and an input panel for the user to enter the number of processes and their details (burst time, arrival time, and priority). Upon clicking the "Next" button, it validates the input, collects process details, schedules the processes using the NonPreemptivePriorityScheduler, and displays the results in a table format along with average turnaround and waiting times, and a Gantt chart. The GUI also includes styled buttons and handles invalid input gracefully.

2.8 NonPreemptivePriorityScheduler.java

```
import java.util.*; // Importing utility classes for data structures
and collections.

// Main class for the Non-Preemptive Priority Scheduler.
public class NonPreemptivePriorityScheduler {

    // Variables to store total turnaround and waiting times for all
processes.
    private double totalTurnaroundTime = 0;
    private double totalWaitingTime = 0;

    // List to maintain the Gantt chart representation of the schedule.
    private final List<String> ganttChart = new ArrayList<>();

    // List to store time markers for the Gantt chart timeline.
    private final List<Integer> timeMarkers = new ArrayList<>();

    // Method to perform scheduling based on Non-Preemptive Priority
Scheduling algorithm.
    public void schedule(List<Process> processes) {
        int currentTime = 0; // Initialize the current time to zero.

        // Priority queue to select the process with the highest
priority and shortest burst time.
        Queue<Process> readyQueue = new PriorityQueue<>(
                Comparator.comparingInt((Process p) -> p.priority) //
Compare by priority.
                        .thenComparingInt(p -> p.burstTime) // If
priorities are equal, compare by burst time.
                        .thenComparingInt(p -> p.arrivalTime) // If
burst times are also equal, compare by arrival time.
        );

        boolean allProcessesHandled = false; // Flag to indicate if all
processes are scheduled.
        timeMarkers.add(currentTime); // Add the initial start time to
the timeline.

        while (!allProcessesHandled) { // Loop until all processes are
handled.
```

```
            // Add processes to the ready queue if they have arrived
and are not yet completed.
            for (Process process : processes) {
                if (!readyQueue.contains(process) &&
process.remainingTime > 0 && process.arrivalTime <= currentTime) {
                    readyQueue.add(process);
                }
            }

            if (!readyQueue.isEmpty()) { // If the ready queue has
processes.

                // Select the process with the highest priority and
shortest burst time.
                Process currentProcess = readyQueue.poll();

                // Add the process to the Gantt chart.
                ganttChart.add("P" + currentProcess.processID);

                // Update the current time by adding the burst time of
the selected process.
                currentTime += currentProcess.burstTime;

                // Add the current time to the time markers after
executing the process.
                timeMarkers.add(currentTime);

                // Calculate finishing time, turnaround time, and
waiting time for the process.
                currentProcess.finishingTime = currentTime;
                int turnaroundTime = currentProcess.finishingTime -
currentProcess.arrivalTime; // Total time from arrival to completion.
                int waitingTime = turnaroundTime -
currentProcess.burstTime; // Time spent waiting in the ready queue.

                // Accumulate the turnaround and waiting times.
                totalTurnaroundTime += turnaroundTime;
                totalWaitingTime += waitingTime;

                // Mark the process as completed by setting its
remaining time to zero.
                currentProcess.remainingTime = 0;
```

```
            } else {
                // If no process is ready, increment the current time.
                currentTime++;

                // Add an idle time marker to the timeline.
                ganttChart.add("Idle");
                timeMarkers.add(currentTime);
            }

            // Check if all processes are completed.
            allProcessesHandled = processes.stream().allMatch(p ->
p.remainingTime == 0);
        }
    }

    // Method to calculate and return the average turnaround time.
    public double getAverageTurnaroundTime() {
        return totalTurnaroundTime / ganttChart.size(); // Divide total
turnaround time by the number of processes.
    }

    // Method to calculate and return the average waiting time.
    public double getAverageWaitingTime() {
        return totalWaitingTime / ganttChart.size(); // Divide total
waiting time by the number of processes.
    }

    // Method to format and return the Gantt chart representation.
    public String getFormattedGanttChart() {
        StringBuilder chartLine = new StringBuilder(); // Line for
process names.
        StringBuilder timeLine = new StringBuilder(); // Line for time
markers.

        // Counter for two-digit numbers
        int twoDigitCount = 0;

        // Initialize Gantt chart formatting.
        chartLine.append("|");
        for (String process : ganttChart) {
            chartLine.append(String.format(" %-7s|", process)); //
Format process name in a fixed-width block.
        }
```

```
        // Append aligned time markers below the Gantt chart
        timeLine.append(" "); // Add one space before the first number
        for (int i = 0; i < timeMarkers.size(); i++) {
            int marker = timeMarkers.get(i);

            if (marker >= 10) {
                // If the marker is a two-digit number, increment the
counter
                twoDigitCount++;
            }

            if (i == 0) {
                // For the first time marker, add 8 spaces
                timeLine.append(String.format("%-8d", marker));
            } else if (i == 1) {
                // For the second time marker, add 10 spaces
                timeLine.append(String.format("%-10d", marker));
            } else if (twoDigitCount >= 2) {
                // After the second two-digit number, add 9 spaces
                timeLine.append(String.format("%-9d", marker));
            } else {
                // Default for other markers (before two-digit
adjustment)
                timeLine.append(String.format("%-11d", marker));
            }
        }
        // Return the Gantt chart with aligned process names and time
markers.
        return chartLine.toString() + "\n" + timeLine.toString();
    }
}
```

The NonPreemptivePriorityScheduler.java file implements the Non-Preemptive Priority scheduling algorithm. It schedules processes based on their burst time, priority, and arrival time, maintaining a Gantt chart and time markers to visualize the schedule. The class calculates total and average turnaround and waiting times for all processes. It includes methods to format and return the Gantt chart representation with aligned process names and time markers, ensuring a clear visual output of the scheduling process.

2.9 RoundRobinGui.java

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

// Main class for the Round Robin Scheduler GUI
public class RoundRobinGUI extends JFrame {
    // Constructor to set up the GUI
    public RoundRobinGUI() {
        setTitle("Round Robin Scheduler"); // Set the title of the
window
        setSize(600, 400); // Set the size of the window
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Set the
default close operation

        // Title panel setup
        JPanel titlePanel = new JPanel() {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2d = (Graphics2D) g;
                Color color1 = new Color(70, 130, 180);
                Color color2 = new Color(240, 248, 255);
                GradientPaint gradient = new GradientPaint(0, 0,
color1, getWidth(), getHeight(), color2);
                g2d.setPaint(gradient);
                g2d.fillRect(0, 0, getWidth(), getHeight()); // Fill
the panel with a gradient color
            }
        };
        titlePanel.setLayout(new BorderLayout()); // Set the layout of
the title panel
        titlePanel.setPreferredSize(new Dimension(getWidth(), 80)); //
Set the preferred size of the title panel

        JLabel titleLabel = new JLabel("Round Robin Scheduler",
JLabel.CENTER); // Create a title label
        titleLabel.setFont(new Font("Arial", Font.BOLD, 22)); // Set
the font of the title label
```

```
        titleLabel.setForeground(Color.WHITE); // Set the color of the
title label
        titlePanel.add(titleLabel, BorderLayout.CENTER); // Add the
title label to the title panel
        add(titlePanel, BorderLayout.NORTH); // Add the title panel to
the frame

        // Input panel setup
        JPanel inputPanel = new JPanel(new GridLayout(0, 1, 10, 10));
// Create an input panel with a grid layout
        inputPanel.setBorder(BorderFactory.createEmptyBorder(30, 50,
30, 50)); // Set the border of the input panel

        JTextField quantumField = new JTextField(); // Create a text
field for the time quantum
        inputPanel.add(new JLabel("Enter Time Quantum:")); // Add a
label for the time quantum
        inputPanel.add(quantumField); // Add the text field for the
time quantum

        JTextField processField = new JTextField(); // Create a text
field for the number of processes
        inputPanel.add(new JLabel("Enter Number of Processes:")); //
Add a label for the number of processes
        inputPanel.add(processField); // Add the text field for the
number of processes

        JButton nextButton = createStyledButton("Next"); // Create a
styled button for the next action
        inputPanel.add(nextButton); // Add the next button to the input
panel

        add(inputPanel, BorderLayout.CENTER); // Add the input panel to
the frame

        // Action listener for the next button
        nextButton.addActionListener(e -> {
            try {
                int numProcesses =
Integer.parseInt(processField.getText()); // Get the number of
processes from the text field
```

```
                int timeQuantum =
Integer.parseInt(quantumField.getText()); // Get the time quantum from
the text field

                if (numProcesses <= 0 || timeQuantum <= 0) {
                    JOptionPane.showMessageDialog(this, "Please enter
valid positive numbers for processes and quantum.", "Error",
JOptionPane.ERROR_MESSAGE);
                    return;
                }

                List<Process> processes = new ArrayList<>(); // Create
a list to store the processes
                for (int i = 0; i < numProcesses; i++) {
                    int burstTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Burst Time for P" +
i + ":")); // Get the burst time for each process
                    int arrivalTime =
Integer.parseInt(JOptionPane.showInputDialog("Enter Arrival Time for P"
+ i + ":")); // Get the arrival time for each process
                    int priority =
Integer.parseInt(JOptionPane.showInputDialog("Enter Priority for P" + i
+ ":")); // Get the priority for each process
                    processes.add(new Process(i, arrivalTime,
burstTime, priority)); // Add the process to the list
                }

                // Pass the processes and time quantum to the scheduler
                RoundRobinScheduler scheduler = new
RoundRobinScheduler(timeQuantum);
                RoundRobinResult result =
scheduler.RoundRobin(processes.size(), timeQuantum,
                    processes.stream().mapToInt(p ->
p.burstTime).toArray(),
                    processes.stream().mapToInt(p ->
p.arrivalTime).toArray());

                // Show the result table
                showTableGUI(processes, scheduler, result);

            } catch (NumberFormatException ex) {
                JOptionPane.showMessageDialog(this, "Invalid input.
Please enter numeric values.", "Error", JOptionPane.ERROR_MESSAGE);
```

```
            }
        });

        setVisible(true); // Make the frame visible
    }

    // Method to show the result table
    private void showTableGUI(List<Process> processes,
RoundRobinScheduler scheduler, RoundRobinResult result) {
        JFrame tableFrame = new JFrame("Round Robin Results"); //
Create a new frame for the results
        tableFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
// Set the default close operation
        tableFrame.setSize(800, 400); // Set the size of the frame

        String[] columns = {"Process", "Arrival Time", "Burst Time",
"Priority", "Finishing Time", "Turnaround Time", "Waiting Time"}; //
Define the column names
        DefaultTableModel model = new DefaultTableModel(columns, 0); //
Create a table model with the column names
        JTable table = new JTable(model); // Create a table with the
model

        // Populate table using RoundRobinResult
        int[] finishTimes = result.getTimeValues().get("finishTime");
        int[] turnaroundTimes = result.getTimeValues().get("tat");
        int[] waitingTimes = result.getTimeValues().get("wt");

        for (int i = 0; i < processes.size(); i++) {
            Process process = processes.get(i);
            model.addRow(new Object[]{
                    "P" + process.processID,
                    process.arrivalTime,
                    process.burstTime,
                    process.priority,
                    finishTimes[i],
                    turnaroundTimes[i],
                    waitingTimes[i]
            });
        }

        JScrollPane scrollPane = new JScrollPane(table); // Create a
scroll pane for the table
```

```
        tableFrame.add(scrollPane, BorderLayout.CENTER); // Add the
scroll pane to the frame

        JButton avgButton = createStyledButton("Show Averages and Gantt
Chart"); // Create a styled button for showing averages and Gantt chart
        tableFrame.add(avgButton, BorderLayout.SOUTH); // Add the
button to the frame

        avgButton.addActionListener(e -> {
            scheduler.showResults(result); // Show the results when the
button is clicked
        });

        tableFrame.setVisible(true); // Make the frame visible
    }

    // Method to create a styled button
    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text); // Create a button with the
specified text
        button.setFocusPainted(false); // Remove focus painting
        button.setFont(new Font("Arial", Font.PLAIN, 18)); // Set the
font of the button
        button.setBackground(new Color(70, 130, 180)); // Set the
background color of the button
        button.setForeground(Color.WHITE); // Set the foreground color
of the button
        button.setBorder(BorderFactory.createCompoundBorder(
                BorderFactory.createLineBorder(new Color(30, 70, 120),
2),
                BorderFactory.createEmptyBorder(10, 15, 10, 15)
        )); // Set the border of the button
        button.setCursor(new Cursor(Cursor.HAND_CURSOR)); // Set the
cursor of the button
        return button; // Return the button
    }

    // Main method to run the application
    public static void main(String[] args) {
        new RoundRobinGUI(); // Create an instance of the GUI
    }
}
```

This Java Swing program provides a GUI for Round Robin CPU Scheduling. It allows users to input the Time Quantum and Number of Processes, then collects Burst Time, Arrival Time, and Priority for each process via pop-up dialogs.

Once inputs are validated, the RoundRobinScheduler processes the scheduling and computes Finishing Time, Turnaround Time, and Waiting Time. The results are displayed in a scrollable JTable, with an option to view averages and a Gantt chart.

The GUI features a gradient title panel, custom-styled buttons, and an intuitive layout. The main method launches the application, offering an interactive way to visualize Round Robin Scheduling.

2.10 RoundRobinScheduler.java

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Queue;
import java.util.LinkedList;

// Class to represent a process
public class RoundRobinScheduler {
    private final int timeQuantum; // Time quantum for the Round Robin
scheduler
    private final List<String> ganttChart = new ArrayList<>(); // Gantt
chart representation
    private final List<Integer> timeMarkers = new ArrayList<>();// Time
markers for the Gantt chart
    private int totalTurnaroundTime = 0;// Total turnaround time
    private int totalWaitingTime = 0;// Total waiting time

    // Constructor for RoundRobinScheduler
    public RoundRobinScheduler(int timeQuantum) {
        this.timeQuantum = timeQuantum;
    }

    // Getters for the variables
    public int getTimeQuantum() {
        return timeQuantum;
    }

    // Getters for the variables
    public void schedule(List<Process> processes) {
        Queue<Process> queue = new LinkedList<>();
        int currentTime = 0;

        // Sort processes by arrival time only (ignore priority)
        processes.sort(Comparator.comparingInt(p -> p.arrivalTime));

        // Iterate through the processes
```

```
        int index = 0;
        while (index < processes.size() || !queue.isEmpty()) {
            // Add processes to the queue based on arrival time
            while (index < processes.size() &&
processes.get(index).arrivalTime <= currentTime) {
                queue.add(processes.get(index));
                index++;
            }

            // Process the queue
            if (!queue.isEmpty()) {
                Process currentProcess = queue.poll(); // Get the first
process in the queue

                // Check if the process has finished
                if (currentProcess.remainingTime <= timeQuantum) {
                    currentTime += currentProcess.remainingTime; //
Update the current time
                    currentProcess.remainingTime = 0; // Set remaining
time to 0
                    currentProcess.finishingTime = currentTime; //
Update finishing time
                    ganttChart.add("P" + currentProcess.processID); //
Add the process to the Gantt chart
                    timeMarkers.add(currentTime); // Add the current
time to the time markers
                } else {
                    currentTime += timeQuantum; // Update the current
time
                    currentProcess.remainingTime -= timeQuantum; //
Deduct the time quantum from the remaining time
                    ganttChart.add("P" + currentProcess.processID); //
Add the process to the Gantt chart
                    timeMarkers.add(currentTime); // Add the current
time to the time markers
                    queue.add(currentProcess);// Add the process back
to the queue
                }
            } else {
                currentTime++; // Increment the current time
            }
        }
```

```
        // Calculate total turnaround time and waiting time
        for (Process process : processes) {
            // Calculate turnaround time and waiting time for each
process
            int turnaroundTime = process.finishingTime -
process.arrivalTime;
            // Calculate waiting time
            int waitingTime = turnaroundTime - process.burstTime;
            // Update the process with the calculated values
            totalTurnaroundTime += turnaroundTime;
            // Update the total turnaround time
            totalWaitingTime += waitingTime;
        }
    }

// Method to calculate the average turnaround time
    public RoundRobinResult RoundRobin(int ProcessNum, int quantumTime,
int burstTime[], int arrivalTime[]) {
        int[] tat = new int[ProcessNum]; // Turnaround time
        int[] wt = new int[ProcessNum];// Waiting time
        Process[] processes = new Process[ProcessNum];// Array of
processes
        String ganttChart = "| "; // Gantt chart representation

        // Initialize the processes
        for (int i = 0; i < ProcessNum; i++) {
            processes[i] = new Process(i, arrivalTime[i], burstTime[i],
0); // Create a new process
        }

        // Sort processes based on arrival time
        Arrays.sort(processes, Comparator.comparingInt(p ->
p.arrivalTime));

        // Calculate the TAT and WT
        int totalTAT = 0;
        int totalWT = 0;

        // Remaining time for each process
        int[] remainingTime = new int[ProcessNum];
        // Initialize the remaining time
        for (int i = 0; i < ProcessNum; i++) {
```

```
            remainingTime[i] = processes[i].burstTime; // Set the
remaining time to the burst time
        }
        // Calculate the total burst time
        int totalburstTime = 0;
        // Calculate the total burst time
        for (int i = 0; i < ProcessNum; i++) {
            totalburstTime += processes[i].burstTime; // Add the burst
time of each process
        }
        // Initialize the time before and after each process
        ArrayList<Integer> processqueue = new ArrayList<>();// Process
queue
        ArrayList<Integer> timebefore = new ArrayList<>();// Time
before each process
        timebefore.add(0);// Add the initial time
        ArrayList<Integer> timeafter = new ArrayList<>(); // Time after
each process
        // Add the time after the first process
        if (processes[0].burstTime < quantumTime) {
            timeafter.add(processes[0].burstTime); // Add the burst
time of the first process
        } else {
            timeafter.add(quantumTime);// Add the quantum time
        }
        // Initialize the finish time
        ArrayList<Integer> finishTime = new ArrayList<>();
        // Initialize the finish time
        for (int j = 0; j < ProcessNum; j++) {
            finishTime.add(0); // Add 0 to the finish time
        }
        // Initialize the waiting time and turnaround time
        for (int j = 0; j < ProcessNum; j++) {
            wt[j] = 0;// Waiting time
        }
        // Initialize the waiting time and turnaround time
        for (int j = 0; j < ProcessNum; j++) {
            tat[j] = 0;// Turnaround time
        }
        // Initialize the Gantt chart time
        ArrayList<Integer> ganttChartTime = new ArrayList<>();
        ganttChartTime.add(0);// Add 0 to the Gantt chart time
```

```
        int currentTime = 0;// Current time
        int k = 0;// Index for the process queue
        int t = 0;// Index for the time markers
        while (true) {
            boolean allProcessesComplete = true; // Check if all
processes are complete

            // Iterate through the processes
            for (int o = 0; o < ProcessNum; o++) {
                if (processes[o].arrivalTime <= timeafter.get(t) &&
processes[o].arrivalTime >= timebefore.get(t))
                 {
                    // Check if the process has remaining time
                    if (remainingTime[o] > 0) {
                        allProcessesComplete = false; // Set
allProcessesComplete to false
                        // Check if the remaining time is greater than
the quantum time
                        if (remainingTime[o] > quantumTime)
                        {
                            currentTime += quantumTime;// Update the
current time
                            ganttChartTime.add(currentTime);// Add the
current time to the Gantt chart time
                            remainingTime[o] -= quantumTime;// Deduct
the quantum time from the remaining time
                            ganttChart += "P" + processes[o].processID
+ " | ";// Add the process to the Gantt chart

timebefore.add(timeafter.get(timeafter.size() - 1) + 1);// Add the time
before the process

timeafter.add(timeafter.get(timeafter.size() - 1) + quantumTime);// Add
the time after the process
                            processqueue.add(o); // Add the process to
the queue
                            continue; // Continue to the next process
                        } else {
                            currentTime += remainingTime[o]; // Update
the current time
                            ganttChartTime.add(currentTime);// Add the
current time to the Gantt chart time
```

```
                            tat[processes[o].processID] = currentTime -
processes[o].arrivalTime;// Calculate the turnaround time
                            remainingTime[o] = 0;// Set the remaining
time to 0
                            wt[processes[o].processID] =
tat[processes[o].processID] - processes[o]. burstTime; // Calculate the
waiting time
                            processes[o].finishingTime = currentTime;
// Update finishing time
                            ganttChart += "P" + processes[o].processID
+ " | "; // Add the process to the Gantt chart

timebefore.add(timeafter.get(timeafter.size() - 1) + 1);// Add the time
before the process

timeafter.add(timeafter.get(timeafter.size() - 1) +
processes[o].burstTime);// Add the time after the process
                            finishTime.set(o, currentTime); // Update
finish time in the list
                            continue;// Continue to the next process
                        }
                    }
                } else {
                    continue;
                }
            }
            // Check if the current time is equal to the total burst
time
            if (k > (processqueue.size() - 1)) {
                k = 0;
            }

            //  Check if the current time is equal to the total burst
time
            if ((remainingTime[processqueue.get(k)] > 0)) {
                allProcessesComplete = false; // Set
allProcessesComplete to false
                if (remainingTime[processqueue.get(k)] > quantumTime) {
                    currentTime += quantumTime; // Update the current
time
                    ganttChartTime.add(currentTime);// Add the current
time to the Gantt chart time
```

```
                    remainingTime[processqueue.get(k)] -=
quantumTime;// Deduct the quantum time from the remaining time
                    ganttChart += "P" +
processes[processqueue.get(k)].processID + " | ";// Add the process to
the Gantt chart
                    timebefore.add(timeafter.get(timeafter.size() - 1)
+ 1); // Add the time before the process
                    timeafter.add(timeafter.get(timeafter.size() - 1) +
quantumTime); // Add the time after the process
                    processqueue.add(processqueue.get(k));// Add the
process to the queue
                    k = k + 1;// Increment k
                    t = t + 1;// Increment t
                    continue;
                } else {
                    currentTime +=
remainingTime[processqueue.get(k)];// Update the current time
                    ganttChartTime.add(currentTime);// Add the current
time to the Gantt chart time
                    tat[processes[processqueue.get(k)].processID] =
currentTime - processes[processqueue.get(k)].arrivalTime;// Calculate
the turnaround time
                    remainingTime[processqueue.get(k)] = 0;// Set the
remaining time to 0
                    wt[processes[processqueue.get(k)].processID] =
tat[processes[processqueue.get(k)].processID] -
processes[processqueue.get(k)].burstTime;// Calculate the waiting time
                    processes[processqueue.get(k)].finishingTime =
currentTime; // Update finishing time
                    ganttChart += "P" +
processes[processqueue.get(k)].processID + " | ";
                    timebefore.add(timeafter.get(timeafter.size() - 1)
+ 1);// Add the time before the process
                    timeafter.add(timeafter.get(timeafter.size() - 1) +
processes[processqueue.get(k)].burstTime);// Add the time after the
process
                    finishTime.set(processqueue.get(k), currentTime);
// Update finish time in the list
                    k = k + 1;
                    t = t + 1;
                    continue;
                }
            }
```

```
            // Check if the current time is equal to the total burst
time
            if (currentTime == totalburstTime) {
                allProcessesComplete = true;
            }

            // Check if all processes are complete
            if (allProcessesComplete) {
                break;
            }
        }
        // Add the last process to the Gantt chart
        ganttChart += " |";

        // Remove the last element from the Gantt chart time
        ganttChart = ganttChart.substring(0, ganttChart.length() - 2);
        StringBuilder sb = new StringBuilder();// Create a
StringBuilder
        // Iterate through the Gantt chart time
        for (int i = 0; i < ganttChartTime.size(); i++) {
            sb.append(ganttChartTime.get(i));// Append the time to the
StringBuilder
            // Check if the time is greater than 9 and not the last
element
            if (ganttChartTime.get(i) > 9 && i < (ganttChartTime.size()
- 1))
            {
                sb.append("   ");// Append spaces to the StringBuilder
            } else {
                sb.append("    ");// Append spaces to the StringBuilder
            }
        }
        // Sort the processes based on process ID
        Arrays.sort(processes, Comparator.comparingInt(p ->
p.processID));

        // Create the final Gantt chart time
        String finalgctime = sb.toString();

        // Check if both ArrayLists have the same size
        if (burstTime.length != arrivalTime.length) {
```

```
            System.out.println("ArrayLists have different sizes. Cannot
create table.");
            return null;
        }

        // Calculate the total turnaround time and waiting time
        for (int i = 0; i < ProcessNum; i++) {
            totalTAT += tat[i];
        }

// Calculate the total turnaround time and waiting time
        for (int i = 0; i < ProcessNum; i++) {
            totalWT += wt[i];
        }
        double avtotalTAT = (double) totalTAT / ProcessNum; // Average
total turnaround time
        double avtotalWT = (double) totalWT / ProcessNum;// Average
total waiting time
        // Convert the finish time ArrayList to an array
        int[] finishTimeArray =
finishTime.stream().mapToInt(Integer::intValue).toArray();
        // Create a HashMap to store the time values
        HashMap<String, int[]> timeValues = new HashMap<>();
        timeValues.put("burstTime", burstTime);// Burst time
        timeValues.put("arrivalTime", arrivalTime);
        timeValues.put("finishTime", finishTimeArray);
        timeValues.put("wt", wt);
        timeValues.put("tat", tat);
        // Return the RoundRobinResult
        return new RoundRobinResult(timeValues, avtotalTAT, avtotalWT,
totalTAT, totalWT, ganttChart, finalgctime);
    }

    // Method to print the values of a HashMap
    public void printHashMapValues(HashMap<String, int[]> map) {
        for (Map.Entry<String, int[]> entry : map.entrySet()) {
            String key = entry.getKey();// Get the key
            int[] values = entry.getValue();// Get the values

            System.out.println("Key: " + key);// Print the key
            System.out.print("Values: ");// Print the values
            for (int value : values) {
                System.out.print(value + " ");
```

```
            }
            System.out.println();// Print a new line
        }
    }
    // Method to show the results
    public void showResults(RoundRobinResult result) {
        JFrame resultsFrame = new JFrame("Round Robin Results -
Averages and Gantt Chart");
        resultsFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        resultsFrame.setSize(600, 400);
        resultsFrame.setLayout(new BorderLayout());

        // Averages
        double avgTurnaroundTime = result.getavTotalTAT();
        double avgWaitingTime = result.getavTotalWT();

        JTextArea resultArea = new JTextArea();
        resultArea.setEditable(false);
        resultArea.setText("Average Turnaround Time: " +
avgTurnaroundTime + "\n");
        resultArea.append("Average Waiting Time: " + avgWaitingTime +
"\n\n");

        // Gantt Chart representation
        resultArea.append("Gantt Chart:\n" + result.getganttChart() +
"\n");
        resultArea.append("Time Markers: " + result.getfinalgctime() +
"\n\n");

        // Simple Gantt Chart representation
        resultArea.append("Simple Gantt Chart:\n");
        for (String process : result.getganttChart().split(" \\| ")) {
            resultArea.append(process + " | ");
        }
        resultArea.append("\n");

        String[] timeMarkers = result.getfinalgctime().split("\\s+");
for (int i = 0; i < timeMarkers.length; i++) {
    resultArea.append(timeMarkers[i]);
    if (i < timeMarkers.length - 1) {
        resultArea.append("    ");
    }
}
```

```
resultArea.append("\n");

        resultsFrame.add(new JScrollPane(resultArea),
BorderLayout.CENTER);
        resultsFrame.setVisible(true);
    }
}
```

Your Round Robin Scheduler implements the Round Robin (RR) CPU Scheduling Algorithm with a GUI to display results. It manages processes using a queue, allocating a fixed time quantum in a cyclic manner. If a process isn't completed within its time slice, it is requeued until it finishes. The scheduler calculates turnaround time (TAT) and waiting time (WT) after execution. A Gantt Chart is generated to visualize execution order, with time markers for clarity. The results, including average TAT, WT, and the Gantt Chart, are displayed in a JFrame GUI.

2.11 RoundRobinResult.java

```
import java.util.HashMap;

public class RoundRobinResult {
    private final HashMap<String, int[]> timeValues; // HashMap to
store the time values for each process
    private final double avTotalTAT; // Average total turnaround time
    private final double avTotalWT; // Average total waiting time
    private final int totalTAT; // Total turnaround time
    private final int totalWT; // Total waiting time
    private final String ganttChart; // Gantt chart
    private final String finalgctime; // Final gantt chart time

    // Constructor for RoundRobinResult
    public RoundRobinResult(HashMap<String, int[]> timeValues, double
avTotalTAT, double avTotalWT, int totalTAT, int totalWT, String
ganttChart, String finalgctime)
    {
        // Initialize the variables
        this.timeValues = timeValues;
        this.avTotalTAT = avTotalTAT;
        this.avTotalWT = avTotalWT;
        this.totalTAT = totalTAT;
        this.totalWT = totalWT;
        this.ganttChart = ganttChart;
        this.finalgctime = finalgctime;
    }

    // Getters for the variables
    public HashMap<String, int[]> getTimeValues() {
        return timeValues;
    }

    // Getters for the variables
    public double getavTotalTAT() {
        return avTotalTAT;
    }

    // Getters for the variables
    public double getavTotalWT() {
        return avTotalWT;
```

```
    }

    // Getters for the variables
    public int getTotalTAT() {
        return totalTAT;
    }

    // Getters for the variables
    public int getTotalWT() {
        return totalWT;
    }

    // Getters for the variables
    public String getganttChart() {
        return ganttChart;
    }

    // Getters for the variables
    public String getfinalgctime() {
        return finalgctime;
    }
}
```

The RoundRobinResult class stores and retrieves the results of the Round Robin Scheduling process. It uses a HashMap to store process time values and contains variables for average turnaround time (TAT), average waiting time (WT), total TAT, total WT, Gantt Chart, and final Gantt Chart time.

The constructor initializes these values, and getter methods allow access to them. This structure ensures efficient data handling for the scheduler's results.

3 Program Output

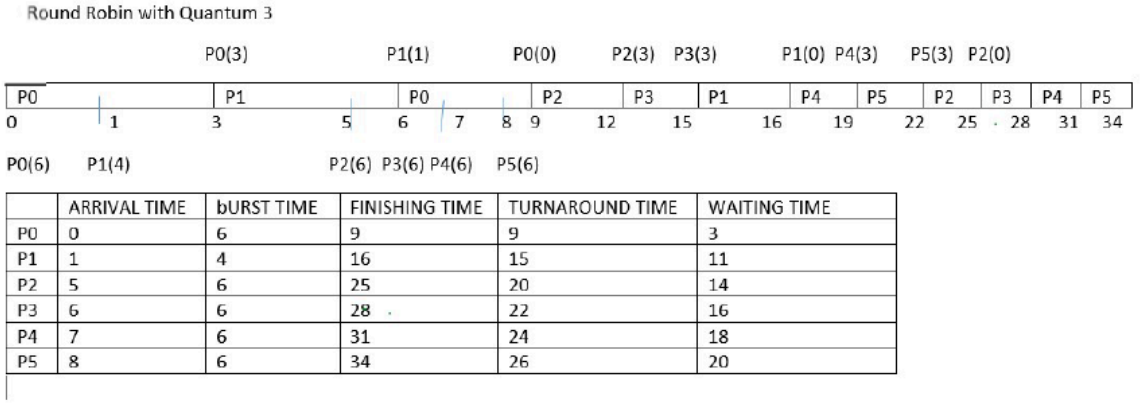
3.1 Output for Round Rubin :

Pre-assigned case

Process	Burst time	Arrival time	Priority
P0	6	0	3
P1	4	1	3
P2	6	5	1
P3	6	6	1
P4	6	7	5
P5	6	8	6

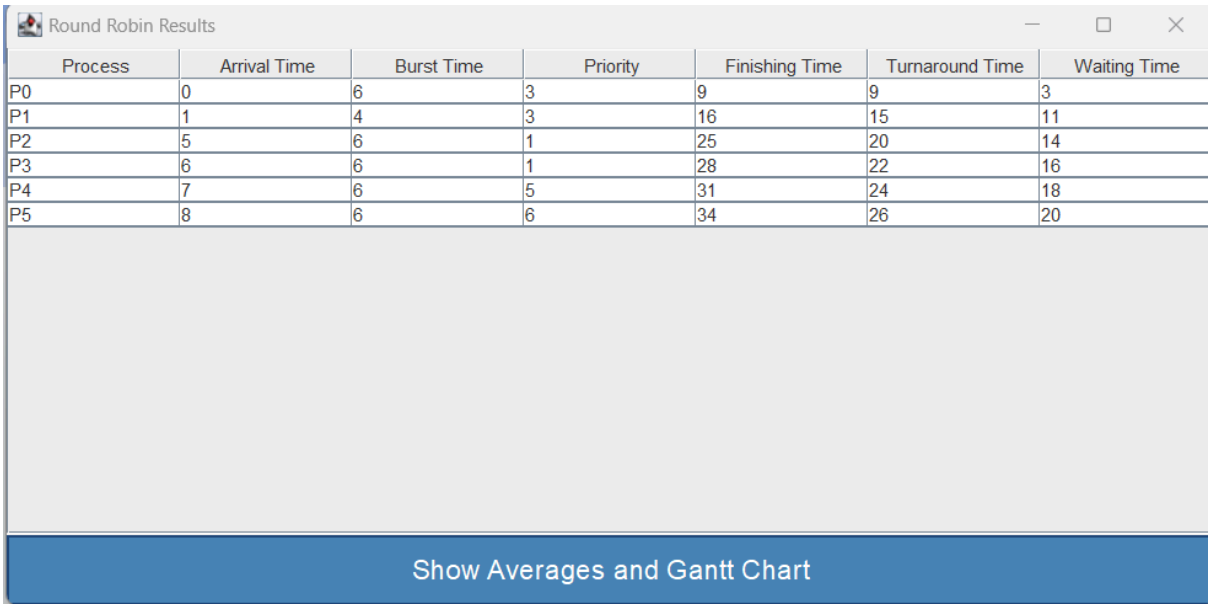
Expected Output from Assignment pdf :

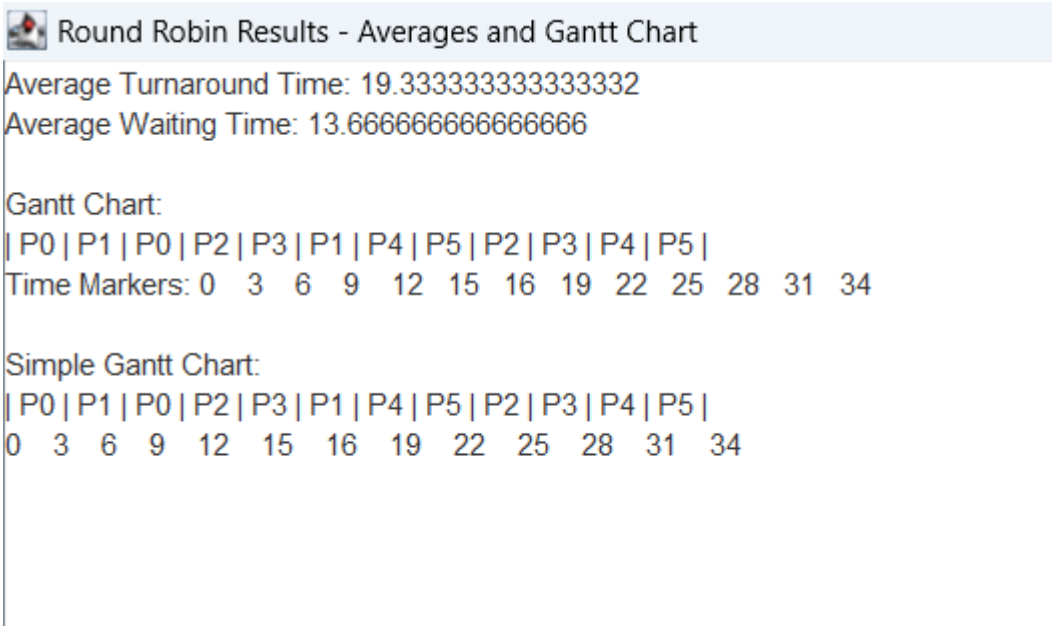
v. Round Robin with Quantum 3



	ARRIVAL TIME	bURST TIME	FINISHING TIME	TURNAROUND TIME	WAITING TIME
P0	0	6	9	9	3
P1	1	4	16	15	11
P2	5	6	25	20	14
P3	6	6	28	22	16
P4	7	6	31	24	18
P5	8	6	34	26	20

Output from My Code for Round Robin with Quantum 3:





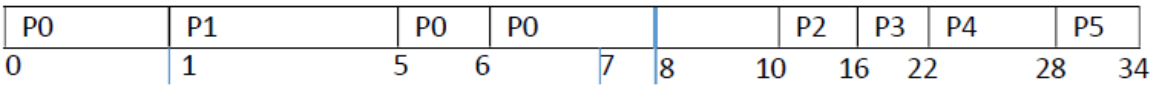
3.2 Output for SRT :

Pre-assigned case

Process	Burst time	Arrival time	Priority
P0	6	0	3
P1	4	1	3
P2	6	5	1
P3	6	6	1
P4	6	7	5
P5	6	8	6

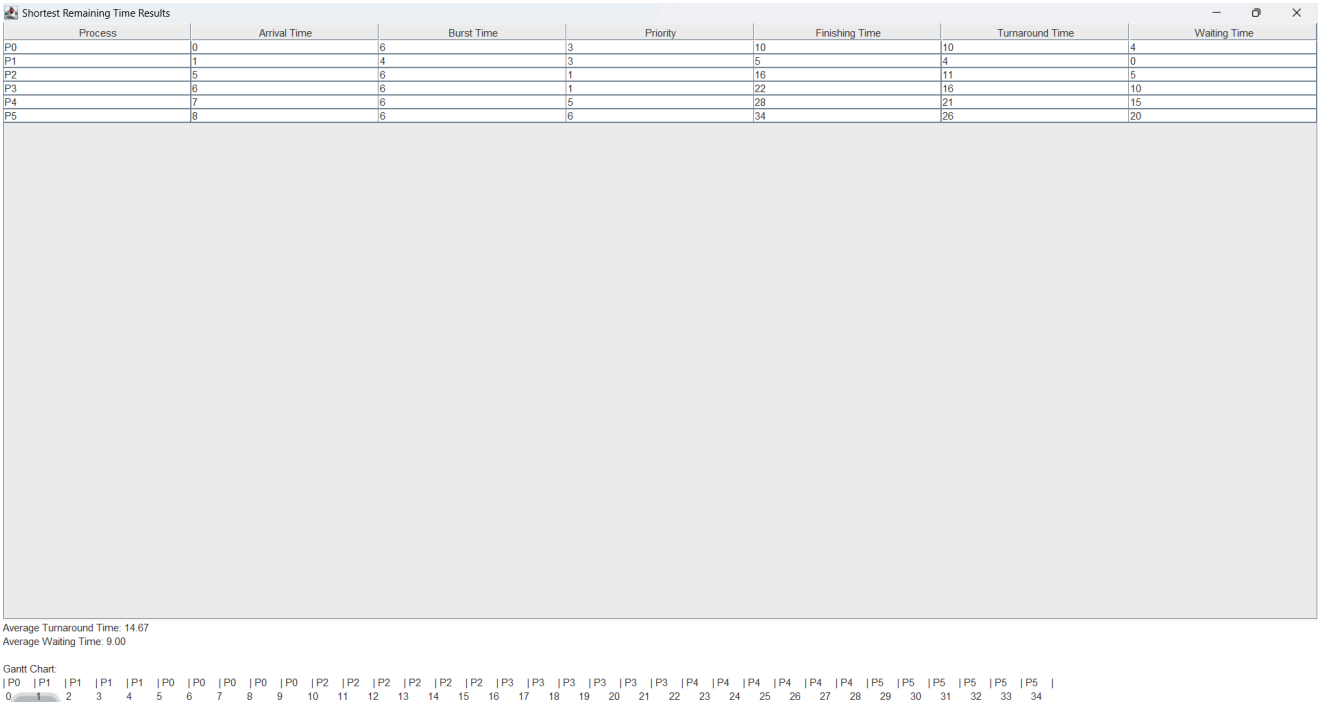
Expected Output from Assignment pdf :

iv. SRT



- If same burst time, look for FCFS

Output from My Code for SRT:



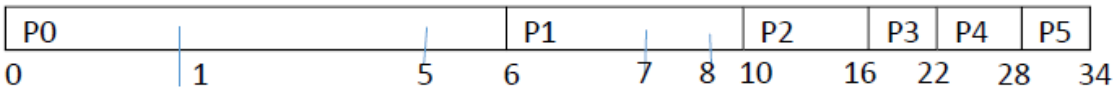
3.3 Output for SJN :

Pre-assigned case

Process	Burst time	Arrival time	Priority
P0	6	0	3
P1	4	1	3
P2	6	5	1
P3	6	6	1
P4	6	7	5
P5	6	8	6

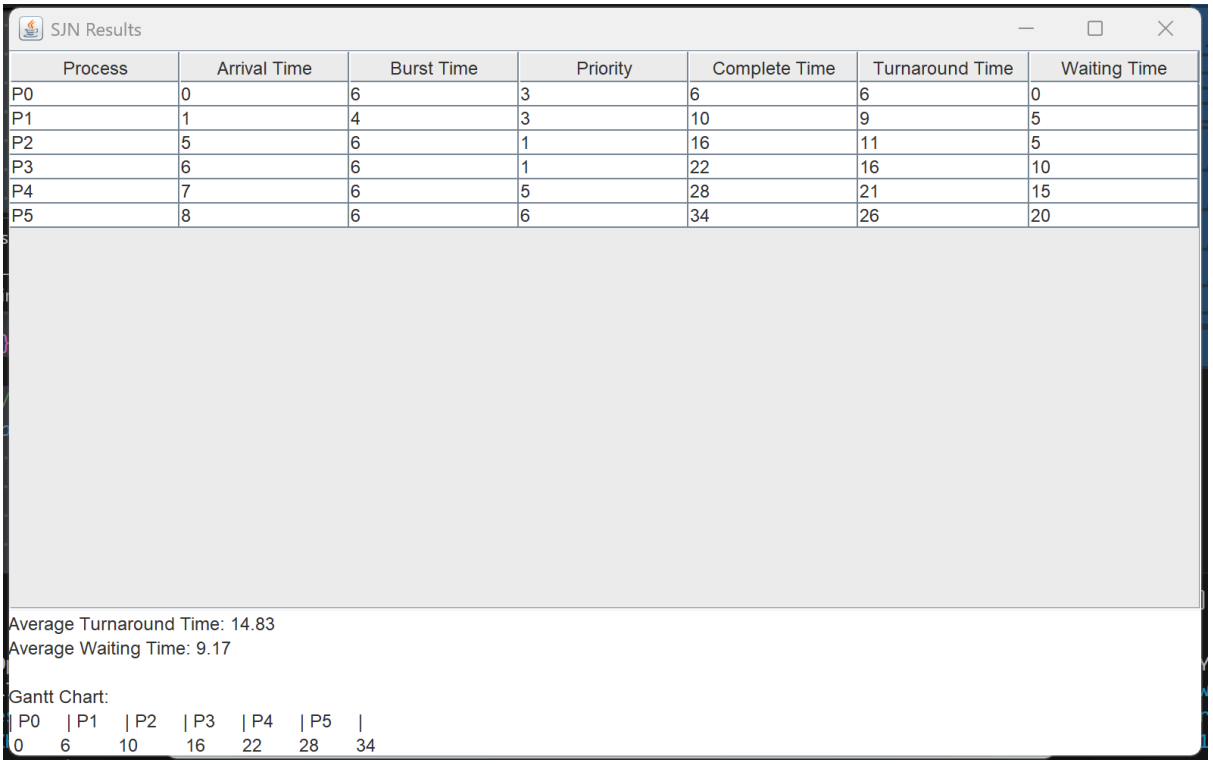
Expected Output from Assignment pdf :

iii. SJN



- IF BURST TIME SAME, CHOOSE FCFS

Output from My Code :



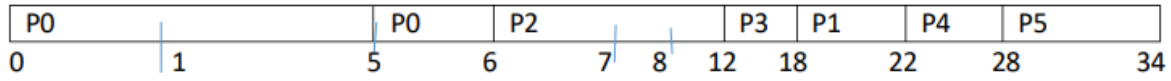
3.4 Output for Non-Preemptive Priority :

Pre-assigned case

Process	Burst time	Arrival time	Priority
P0	6	0	3
P1	4	1	3
P2	6	5	1
P3	6	6	1
P4	6	7	5
P5	6	8	6

Expected output for Non-Preemptive Priority:

i. NON-Preemptive Priority



SAME PRIORITY, NO PREEMPTION, TO REDUCE CONTEXT SWITCH

SAME PRIORITY, CHOOSE FCFS

Output from My Code for Non-Preemptive Priority: