Shortest-Remaining-Time-First
(Preemptive SJF)

Name: Waleed Saleh Ali Saleh ID: 202006448

Name: Mahmood Husain Ebrahim ID: 202008990

Name: Mohammed Hussain Mahdi ID: 20194444

Sec: 1

❖ In this project, we used Object-Oriented Programming (OOP) principles and we included Javadoc-style comments within the code to document all methods and classes. The program consists of three classes: "Process", "SRTFScheduler", and "SRTFMain".

1. Process class:

```java
/**
 * Represents a process with an ID, arrival time, and burst time.
 * Keeps track of the process's execution and scheduling information.
 *
 * @author Mahmood
 */
public class Process {
    private final int ProcessID;
    private final int ArrivalTime;
    private final int BurstTime;

    private int ResponseTime = -1;
    private int TurnAroundTime = -1;
    private int WaitingTime = 0;

    private boolean Terminate = false;
    private int ExecutedTime = 0;
    private int StartingTime = -1;
    private  int  FinishingTime;
    private int RemainingTime;

    /**
     * Constructs a new process with the specified ID, arrival time, and burst time.
     *
     * @param ID the process ID
     * @param ArrivalTime the arrival time of the process
     * @param BurstTime the burst (execution) time of the process
     */
    public Process(int ID, int ArrivalTime, int BurstTime){
        ProcessID = ID;
        this.ArrivalTime = ArrivalTime;
        this.BurstTime = BurstTime;
    }

    /**
     * Returns the remaining time (burst time - executed time) of the process.
     *
     * @return the remaining time of the process
     */
    public int getRemainingTime() {return RemainingTime;}

    /**
     * Returns the starting time of the process.
     *
     * @return the starting time of the process
     */
    public int getStartingTime() {return StartingTime;}

    /**
     * Returns the ID of the process.
     *
     * @return the ID of the process
     */
    public int getProcessID() {return ProcessID;}
```

```java
    /**
     * Returns the arrival time of the process.
     *
     * @return the arrival time of the process
     */
    public int getArrivalTime(){return ArrivalTime;}

    /**
     * Returns the burst time of the process.
     *
     * @return the burst time of the process
     */
    public int getBurstTime() {return BurstTime;}

    /**
     * Returns the executed time of the process.
     *
     * @return the executed time of the process
     */
    public int getExecutedTime() {return ExecutedTime;}

    /**
     * Returns the turn-around time of the process.
     *
     * @return the turn-around time of the process
     */
    public int getTurnAroundTime() {return TurnAroundTime;}

    /**
     * Returns the waiting time of the process.
     *
     * @return the waiting time of the process
     */
    public int getWaitingTime() {return WaitingTime;}

    /**
     * Returns the finishing time of the process.
     *
     * @return the finishing time of the process
     */
    public int getFinishingTime() {return FinishingTime;}

    /**
     * Returns the response time of the process.
     *
     * @return the response time of the process
     */
    public int getResponseTime() {return ResponseTime;}

    /**
     * Sets the response time of the process.
     *
     * @param response_time the response time of the process
     */
    public void setResponse_time(int response_time) {this.ResponseTime = response_time;}

    /**
     * Sets the turn-around time of the process.
     *
     * @param turnAroundTime the turn-around time of the process
     */
    public void setTurnAroundTime(int turnAroundTime) {this.TurnAroundTime =
```

```
turnAroundTime;}

    /**
     * Sets the waiting time of the process.
     *
     * @param waitingTime the waiting time of the process
     */
    public void setWaitingTime(int waitingTime) {this.WaitingTime = waitingTime;}

    /**
     * Sets the finishing time of the process.
     *
     * @param finishing_time the finishing time of the process
     */
    public void setFinishing_time(int finishing_time) {this.FinishingTime =
finishing_time;}

    /**
     * Sets the starting time of the process.
     *
     * @param start the starting time of the process
     */
    public void setStartingTime(int start) {this.StartingTime = start;}

    /**
     * Sets the remaining time (burst time - executed time) of the process.
     */
    public void setRemainingTime() {RemainingTime = getBurstTime() - getExecutedTime();}

    /**
     * Returns a boolean indicating whether the process has terminated (i.e., executed
for its entire burst time).
     *
     * @return true if the process has terminated, false otherwise
     */
    public boolean isTerminate() {return Terminate;}

    /**
     * Increments the executed time of the process by 1, and sets the process to
terminate if the executed time equals the burst time.
     */
    public void setExecutedTime() {
        ExecutedTime++;
        if (ExecutedTime == getBurstTime()) Terminate = true;
    }
}
```

2. SRTFScheduler class:

```
import java.util.*;
/**
 * Implements the Shortest Remaining Time First (SRTF) scheduling algorithm for a set of
processes.
 *
 * @author waleed
 */
public class SRTFScheduler {
    private Process[] ReadyQueue;               // Array called ReadyQueue of type
Process that store all process
    private final Set<Integer> processIDs;      // Hash Set called ProcessIDs of type
Integer that store all process ID's
    private int NumOfProcess;                   // Number of process in the ReadyQueue
    public double AverageTurnAroundTime, AverageWaitingTime, AverageResponseTime;   //
Three variable to calculated average times

    /**
     * Constructs a new SRTF (Shortest Remaining Time First) scheduler with an empty
ready queue
     * and no processes. The ready queue is initially allocated with space one process.
     * Processes can be added to the scheduler using the AddProcess() method.
     */
    public SRTFScheduler() {
        ReadyQueue = new Process[10];
        processIDs = new HashSet<>();
        NumOfProcess = 0;
    }

    /**
     * Re-sorts the ready queue based on process arrival time, using a simple selection
sort algorithm.
     */
    public void ReSortBasedOnArriveTime() {
        Process temp1;
        for (int i = 0; i < NumOfProcess; i++)
            for (int j = i+1; j < NumOfProcess; j++)
                if (ReadyQueue[i].getArrivalTime() > ReadyQueue[j].getArrivalTime()){
                    temp1 = ReadyQueue[i];
                    ReadyQueue[i]= ReadyQueue[j];
                    ReadyQueue[j]=temp1;
                }
    }

    /**
     * Re-sorts the ready queue based on process ID, using a simple selection sort
algorithm.
     */
    public void ReSortBasedOnProcessID() {
        Process temp1;
        for (int i = 0; i < NumOfProcess; i++)
            for (int j = i+1; j < NumOfProcess; j++)
                if (ReadyQueue[i].getProcessID() > ReadyQueue[j].getProcessID()){
                    temp1 = ReadyQueue[i];
                    ReadyQueue[i]= ReadyQueue[j];
                    ReadyQueue[j]=temp1;
                }
    }

    /**
     * Re-sorts the given list of processes based on remaining time, using a simple
bubble sort algorithm.
```

```
     *
     * @param list the list of processes to be sorted
     */
    public void ReSortBasedOnRemainingTIme(ArrayList<Process> list) {
        int n = list.size();
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (list.get(j).getRemainingTime() > list.get(j + 1).getRemainingTime())
{
                    Process temp = list.get(j);
                    list.set(j, list.get(j + 1));
                    list.set(j + 1, temp);
        }
    }

    /**
     * Checks if a given process ID is valid (i.e., not already in the ready queue).
     *
     * @param id the process ID to check
     * @return true if the given ID is valid, false otherwise
     */
    public boolean ValidID(int id) {
        return !processIDs.contains(id);
    }

    /**
     * Returns true if the ready queue is empty (i.e., there are no processes in the
queue), false otherwise.
     *
     * @return true if the ready queue is empty, false otherwise.
     */
    public boolean isEmpty(){
        return NumOfProcess == 0;
    }

    /**
     * Adds a new process to the ready queue with the given ID, arrival time, and burst
time.
     *
     * @param id the ID of the new process
     * @param at the arrival time of the new process
     * @param bt the burst time of the new process
     */
    public void AddProcess(int id, int at, int bt) {
        if (ReadyQueue.length==NumOfProcess)
            ReadyQueue = Arrays.copyOf(ReadyQueue,ReadyQueue.length*2);

        ReadyQueue[NumOfProcess] = new Process(id, at, bt);
        NumOfProcess++;
        processIDs.add(id);
    }

    /**
     * Implements the Shortest Remaining Time First (SRTF) scheduling algorithm.
     *
     * @return true if the scheduling algorithm was successful, false otherwise
     */
    public boolean SchedulingAlgorithm() {
        if (NumOfProcess <= 0) return false;

        ArrayList<Process> WaitingArray = new ArrayList<>(NumOfProcess);
        ReSortBasedOnArriveTime();
```

```
        int TotalTime = 0;
        for (int i=0; i<NumOfProcess; i++)
            TotalTime += ReadyQueue[i].getBurstTime();

        int CurrentProcessID = -1, NewProcessID = -1 , counter=0;
        for (int CurrentTime = 0; CurrentTime < TotalTime; CurrentTime++) {
            // Add new processes to the waiting array based on:
            // 1. arrival time for the process is less than or equal CurrentTime
            // 2. process not terminate yet
            for (int i=0; i<NumOfProcess; i++) {
                if (ReadyQueue[i].getArrivalTime() <= CurrentTime &&
!ReadyQueue[i].isTerminate())
                    if (ReadyQueue[i].getRemainingTime() == 0 &&
ReadyQueue[i].getExecutedTime()==0)
                        WaitingArray.add(ReadyQueue[i]);
            }
            // Update the remaining time of waiting processes
            for (Process Pro: WaitingArray)
                Pro.setRemainingTime();

            // If a process has finished execution, move it from the waiting array to the
ready queue
            if (!WaitingArray.isEmpty() && WaitingArray.get(0).getRemainingTime()==0){
                ReadyQueue[counter]= WaitingArray.remove(0);
                counter++;
            }

            // Sort the waiting array based on remaining time
            ReSortBasedOnRemainingTIme(WaitingArray);

            // Determine the next process to execute
            if(WaitingArray.size() > 0){
                NewProcessID = WaitingArray.get(0).getProcessID();
            }

            // Execute the next process
            if (NewProcessID == -1) {
                System.out.print(CurrentTime + " | NP | ");
                TotalTime++;
                NewProcessID =-10;
            }
            else if (NewProcessID == -10){
                TotalTime++;
            }
            else if (NewProcessID >= 0 && NewProcessID != CurrentProcessID) {
                System.out.print(CurrentTime + " | P" + NewProcessID + " | ");
                CurrentProcessID = NewProcessID;
                WaitingArray.get(0).setExecutedTime();
                if (WaitingArray.get(0).getStartingTime() == -1) {
                    WaitingArray.get(0).setStartingTime(CurrentTime);
                }
            }
            else if (CurrentProcessID == NewProcessID) {
                WaitingArray.get(0).setExecutedTime();
            }

            // If a process has finished execution, update its finishing time and mark it
as terminated
            if (NewProcessID >= 0 && WaitingArray.get(0).isTerminate()){
                WaitingArray.get(0).setFinishing_time((CurrentTime) + 1);
                NewProcessID = -1;
            }
```

```
            // If all processes have finished execution, calculate the average turnaround
time, waiting time, and response time
            if (CurrentTime + 1 == TotalTime) {
                ReadyQueue[counter]= WaitingArray.remove(0);
                System.out.println(TotalTime);
                ReSortBasedOnProcessID();
                for (int i=0; i<NumOfProcess; i++) {
                    ReadyQueue[i].setResponse_time(ReadyQueue[i].getStartingTime() -
ReadyQueue[i].getArrivalTime());
                    ReadyQueue[i].setTurnAroundTime(ReadyQueue[i].getFinishingTime() -
ReadyQueue[i].getArrivalTime());
                    ReadyQueue[i].setWaitingTime(ReadyQueue[i].getTurnAroundTime() -
ReadyQueue[i].getBurstTime());
                }
            }

        }
        return true;
    }

    /**
     This method calculates and prints the average turnaround time, response time, and
waiting time for all processes in the ReadyQueue.
     The average response time, turnaround time, and waiting time are calculated by
dividing the total values by the total number of processes.
     The calculated average values are rounded to three decimal places using the
Math.round() method.
     The method then generates a formatted string containing the calculated average
values and prints it to the console.
     */
    public void PrintProcessesAverages() {
        if (NumOfProcess <= 0) return;
        double WaitingTime = 0, TurnAroundTime = 0, ResponseTime = 0;
        for (int i=0; i<NumOfProcess; i++) {
            WaitingTime += ReadyQueue[i].getWaitingTime();
            TurnAroundTime += ReadyQueue[i].getTurnAroundTime();
            ResponseTime += ReadyQueue[i].getResponseTime();
        }

        AverageTurnAroundTime = TurnAroundTime / NumOfProcess;
        AverageWaitingTime = WaitingTime / NumOfProcess;
        AverageResponseTime = ResponseTime / NumOfProcess;
        AverageResponseTime = Math.round(AverageResponseTime * 1000.0) / 1000.0;
        AverageTurnAroundTime = Math.round(AverageTurnAroundTime * 1000.0) / 1000.0;
        AverageWaitingTime = Math.round(AverageWaitingTime * 1000.0) / 1000.0;

        String Print =
"\n\n==========================================================================\n" +
                "=                    Average Times For All Process:
=\n" +

"==========================================================================\n" +
                String.format("%-2s %-2s %-40s %-2s\n",
                        "=", "Average Turnaround Time:", AverageTurnAroundTime + " ms",
"=") +
                String.format("%-2s %-2s %-40s %-2s\n",
                        "=", "Average Response Time  :", AverageResponseTime + " ms",
"=") +
                String.format("%-2s %-2s %-40s %-2s\n",
                        "=", "Average Waiting Time   :", AverageWaitingTime + " ms", "=")
+

"==========================================================================\n";
```

```
        System.out.print(Print);
    }

    /**
     This method prints the details of all processes in the ReadyQueue to the console.
     The method iterates through each process in the ReadyQueue and generates a formatted
     string containing the process ID, waiting time, turnaround time, and response time
for each process.
     */
    public void PrintProcessesDetails() {
        StringBuilder sb = new StringBuilder();

sb.append("\n\n=============================================================================\n")
;
        sb.append("=                    The Details For All Processes
=\n");

sb.append("=============================================================================\n");
        sb.append(String.format("%-2s %-11s %-2s %-13s %-2s %-16s %-2s %-2s %-1s\n",
                "=","Process ID","=","Waiting Time","=","Turnaround Time","=", "Response
Time",'='));

sb.append("=============================================================================\n");
        for (int i=0; i<NumOfProcess; i++) {
            sb.append(String.format(
                            "%-6s %-7s %-7s %-8s %-9s %-9s %-8s %-7s %-1s\n",
                            "=", ReadyQueue[i].getProcessID(),       "=",
                            ReadyQueue[i].getWaitingTime()+" ms",    "=",
                            ReadyQueue[i].getTurnAroundTime()+" ms", "=",
                            ReadyQueue[i].getResponseTime()+" ms",   "="));
        }

sb.append("=============================================================================\n");
        System.out.print(sb);
    }

     /**
      This method is responsible for printing the welcome screen of the SRTF Scheduler
program to the console.
      The welcome screen consists of a title, section, and a list of names prepared by
the team.
     */
    public static void printWelcomeScreen() {
        final String TITLE = "Project ITCS325: SRTF Scheduler";
        final String SECTION = "Section: 1";
        final String []  NAMES = {
                " Prepared by:",
                " Name: Waleed  Saleh Ali Saleh              ID: 202006448",
                " Name: Mahmood Husain Ebrahim               ID: 202008990",
                " Name: Mohammed Hussain Mahdi               ID: 20194444 "
        };
        int z = 0;
        int numRows = 10;
        int numCols = 60;
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                if (i==0 || i==2 || i==numRows-3 || i==numRows-1)
                    System.out.print('=');
                else if(i==1 && j==59) {
                    System.out.print("=\t\t\t  "+TITLE+"\t\t\t   =");
                }
                else if(i==8 && j==59){
                    System.out.print("=                              "+SECTION+"
```

```
=");
                }
                else if (i>2 && i<7 && j==59){
                    System.out.print("="+NAMES[z]);
                    z++;
                    if (i==3)
                        System.out.print("
");
                    else{
                        System.out.print(" ");
                    }
                    System.out.print("=");
                }
            }
            System.out.println();
        }
    }
}
```

3. SRTFMain class:

```
/*
 Name: Waleed  Saleh Ali Saleh              ID: 202006448
 Name: Mahmood Husain Ebrahim               ID: 202008990
 Name: Mohammed Hussain Mahdi               ID: 20194444
 Sec: 1
 */
import java.util.*;
/**
 * SRTFMain class contains the main method to run the SRTF (Shortest Remaining Time
First) scheduling algorithm.
 *
 * @author Mohammed
 */
public class SRTFMain {
    /**
     * The main method is the entry point of the program that runs the SRTF (Shortest
Remaining Time First) scheduling
     * algorithm. It takes input from the user about the process IDs, arrival times, and
burst times and then prints a
     * Gantt chart to show the execution sequence of the processes. It also displays the
process details and averages
     * after the scheduling is completed.
     *
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        SRTFScheduler.printWelcomeScreen();

        String ScheduleAgain;
        do {
            SRTFScheduler scheduler = new SRTFScheduler();
            int inputCounter = 0;
            System.out.println("\nEnter the Information About Processes: (Enter '0 0 0'
To Exit)");
            System.out.println("ID AT BT");

            while (true) {
                inputCounter++;
                int id, arrivalTime, burstTime;
                try {
                    id = scanner.nextInt();
                    arrivalTime = scanner.nextInt();
                    burstTime = scanner.nextInt();
                } catch (InputMismatchException e) {
                    System.out.printf("Error at input No. %d: Invalid input format.\n",
inputCounter);
                    scanner.nextLine(); // consume the invalid input to avoid an infinite
loop
                    System.out.println("Enter the Information About Processes: (Enter '0
0 0' to Exit)");
                    System.out.println("ID AT BT");
                    continue; // go back to the beginning of the loop to read the next
input
                }

                if ((id + arrivalTime + burstTime) == 0 && inputCounter == 1) {
                    break;
                } else if ((id + arrivalTime + burstTime) == 0) {
                    break;
                }
```

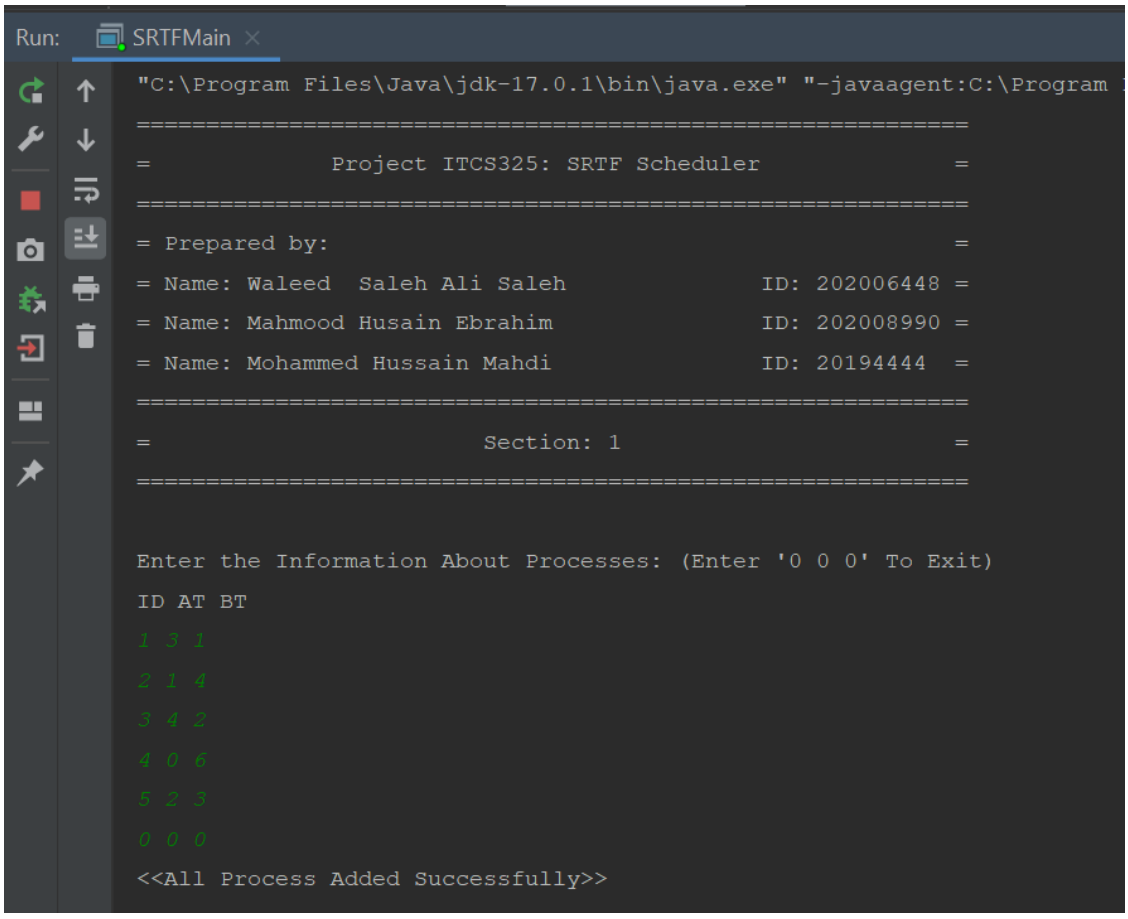
```
                boolean validID = scheduler.ValidID(id);
                boolean isValidInput = arrivalTime >= 0 && burstTime > 0;

                if (!validID || !isValidInput) {
                    System.out.printf("Error at input No. %d has incorrect values: ",
inputCounter);
                    if (!validID)
                        System.out.print("( The ID is already taken by another Process
)\n");
                    if (arrivalTime < 0)
                        System.out.print("( The Arrival time is not valid 'less than 0'
)\n");
                    if (burstTime <= 0)
                        System.out.print("( The burst time is not valid 'less than or
equaled 0' )\n");
                    System.out.println("\nEnter the Information About Processes: (Enter
'0 0 0' to Exit)");
                    System.out.println("ID AT BT");
                } else {
                    scheduler.AddProcess(id, arrivalTime, burstTime);
                }
            }
            if (inputCounter == 1 || scheduler.isEmpty()) {
                System.out.println("<<<There Are No Processes>>");
            } else {
                System.out.println("<<All Process Added Successfully>>\n");

System.out.println("=============================================================================
=");
                System.out.println("=             Gantt Chart For SJF Scheduling-
Preemptive:              =");

System.out.println("=============================================================================
=");
                System.out.println("Note: (NP Means There is no Process executed At This
Time)\n");
                System.out.print(" ");
                if (scheduler.SchedulingAlgorithm()) {

System.out.println("\n=============================================================================
===");
                    scheduler.PrintProcessesDetails();
                    scheduler.PrintProcessesAverages();
                }
            }

            do {
                System.out.println("\nDo you want to enter a new set of processes for
scheduling? (Y or N): ");
                ScheduleAgain = scanner.next();
            }while (!(ScheduleAgain.equalsIgnoreCase("y") ||
ScheduleAgain.equalsIgnoreCase("N")));

        } while (ScheduleAgain.equalsIgnoreCase("Y"));
        scanner.close();
    }
}
```

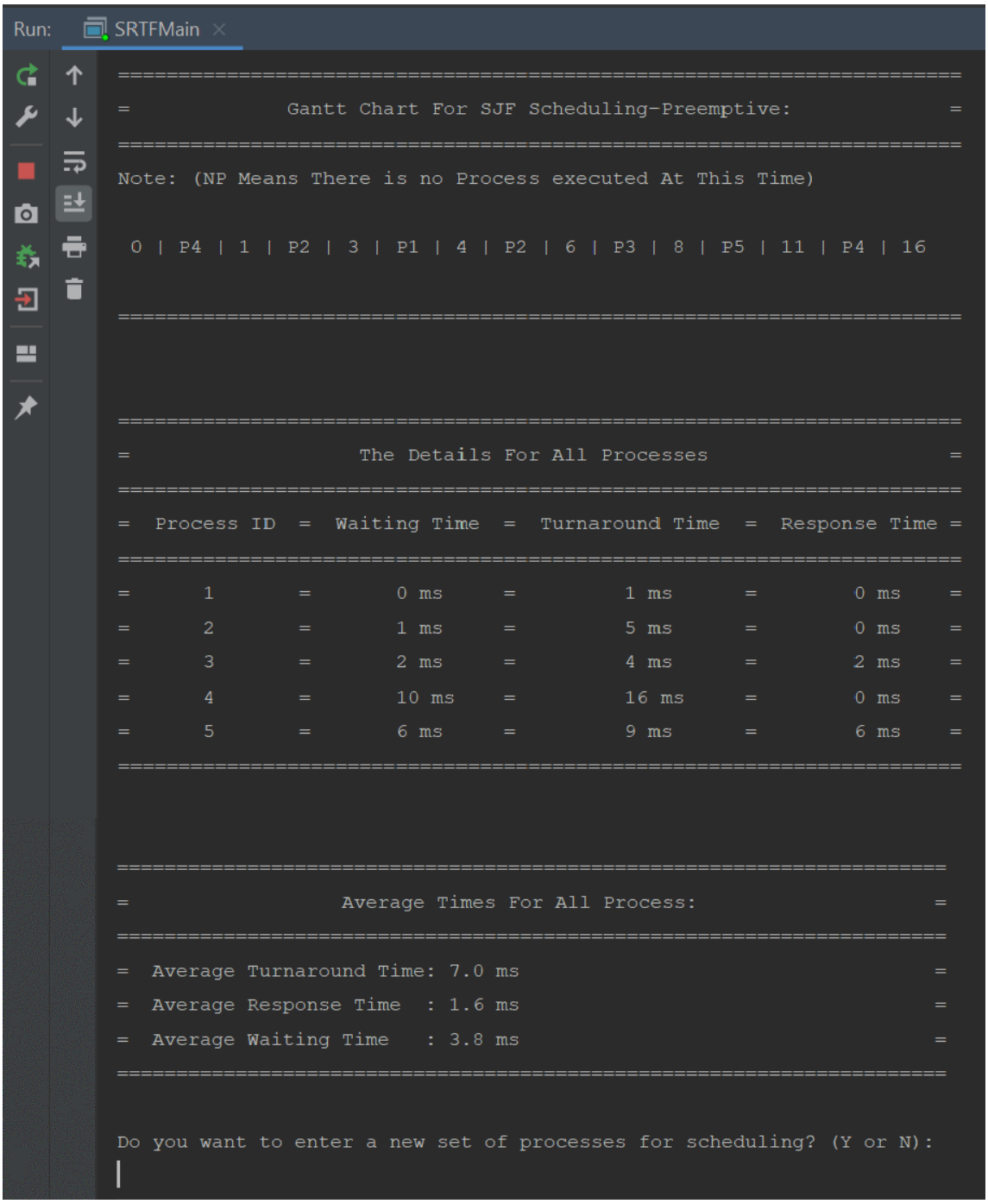
❖ Input/output Examples:

1. First Example:

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

Solution:

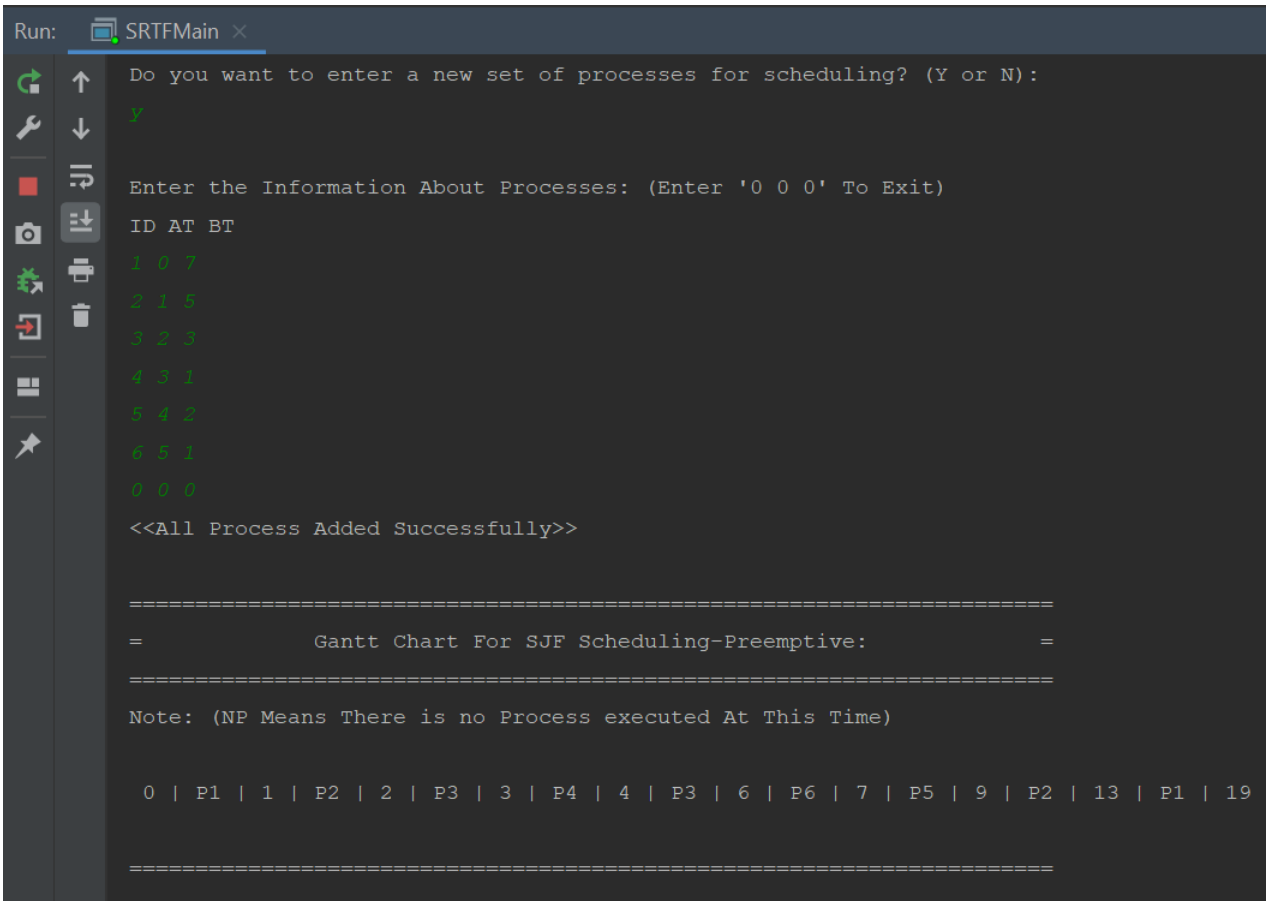


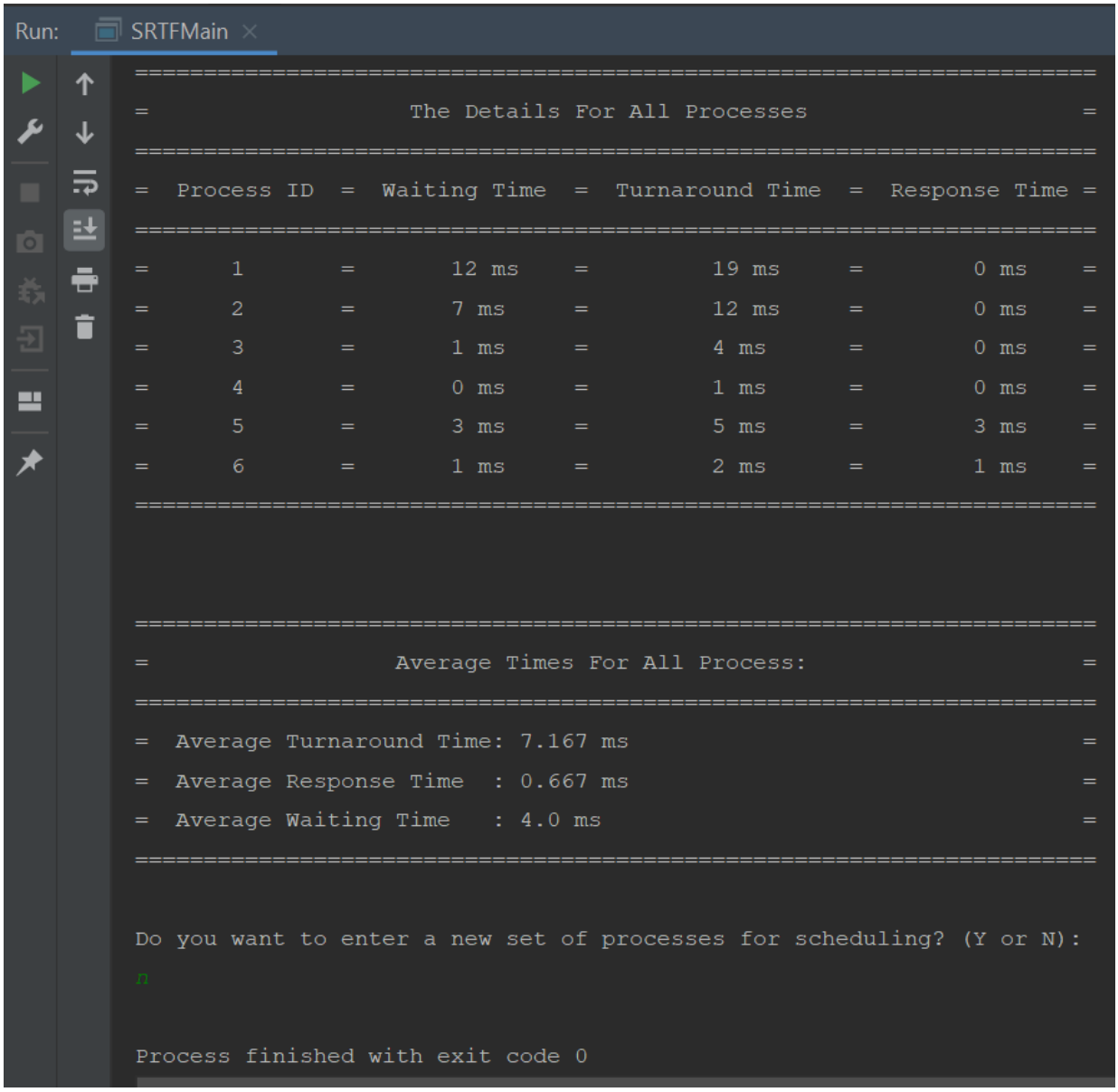

Run:
SRTFMain
Gantt Chart For SJF Scheduling-Preemptive:
Note: (NP Means There is no Process executed At This Time)
0 | P4 | 1 | P2 | 3 | P1 | 4 | P2 | 6 | P3 | 8 | P5 | 11 | P4 | 16
The Details For All Processes
Process ID = Waiting Time = Turnaround Time = Response Time
1 = 0 ms = 1 ms = 0 ms
2 = 1 ms = 5 ms = 0 ms
3 = 2 ms = 4 ms = 2 ms
4 = 10 ms = 16 ms = 0 ms
5 = 6 ms = 9 ms = 6 ms
Average Times For All Process:
Average Turnaround Time: 7.0 ms
Average Response Time : 1.6 ms
Average Waiting Time : 3.8 ms
Do you want to enter a new set of processes for scheduling? (Y or N):

2. Second Example:

Process Id	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Solution:




Run:
SRTFMain
The Details For All Processes
Process ID = Waiting Time = Turnaround Time = Response Time
1 = 12 ms = 19 ms = 0 ms
2 = 7 ms = 12 ms = 0 ms
3 = 1 ms = 4 ms = 0 ms
4 = 0 ms = 1 ms = 0 ms
5 = 3 ms = 5 ms = 3 ms
6 = 1 ms = 2 ms = 1 ms
Average Times For All Process:
Average Turnaround Time: 7.167 ms
Average Response Time : 0.667 ms
Average Waiting Time : 4.0 ms
Do you want to enter a new set of processes for scheduling? (Y or N):
n
Process finished with exit code 0