PROJECT REPORT (CSF204)

A report submitted in partial fulfilment of the requirement for the course

OPERATING SYSTEM

Part of the degree of

BACHELOR OF TECHNOLOGY

In

CSE/IT/EE/CE/PE

Submitted to

Mr. Neeraj Rathore

Assistant Professor

Submitted by:

BHAVISH (1000015397)
200102404

Section: H

SCHOOL OF COMPUTING

DIT UNIVERSITY, DEHRADUN

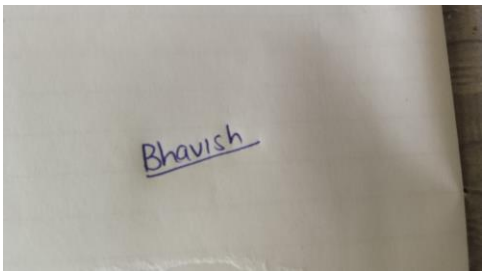
(State Private University through State Legislature Act No. 10 of 2013 of Uttarakhand and approved by UGC)

Mussoorie Diversion Road, Dehradun, Uttarakhand - 248009, India.

2021

CANDIDATES DECLARATION

I hereby certify that the work, which is being presented in the Report, entitled **Process Scheduling Solver,** in partial fulfilment of the requirement as part of the course **Operating System** of the Degree of **Bachelor of Technology** and submitted to the DIT University is an authentic record of my work carried out during the period ***11-04-2022*** to ***26-04-2022*** under the guidance of **Mr Neeraj Rathore.**



Signature of the Candidate

Date: 23-04-2022

ABSTRACTION

The purpose of this project is to outline impmentation of **Process Scheduling Solver** through a GUI application with the help of **javax.swing**, **java.awt**, **java.util packages** and **java collections frameworks**. The project is all about to demonstrate the working of algorithms like **FCFS** (First Come First Serve), **SJF** (Shortest Job First), **SRTF** (Shortest Remaining Time First) with different processes having different **arrival times** and different **burst times**. The user is being provided with choices of different algorithms to solve the process scheduling and in order, to provide their working operations GUI interface has been provided. The record for the **process ids**, **arrival times**, **burst times**, **completion time**, **turnaround times**, **waiting times** along with the **average Turn Around Time** and **average Waiting Time** has been provided for the details.

TABLE OF CONTENT

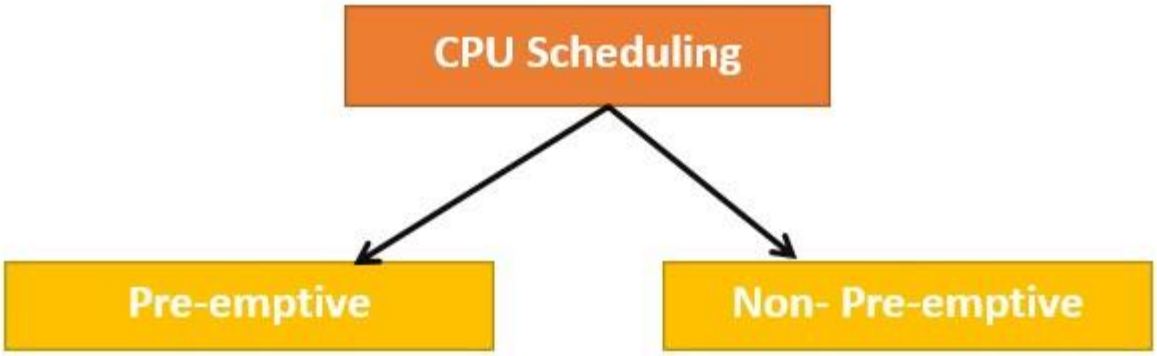
Chapter 1

What is CPU Scheduling

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Types of CPU Scheduling

Here are two kinds of Scheduling methods:



i) Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a **higher priority** before another **lower priority** task, even if the lower priority task is still running. **The lower priority task holds for some time and resumes when the higher priority task finishes its execution**.

ii) Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. **The process that keeps the CPU busy will release the CPU either by switching context or terminating**. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

a. A process switches from the running to the waiting state.
b. Specific process switches from the running state to the ready state.
c. Specific process switches from the waiting state to the ready state.
d. Process finished its execution and terminated.

Important CPU scheduling Terminologies

Burst Time/Execution Time: It is a time required by the process to complete execution. It is also called running time.

- **Arrival Time**: when a process enters in a ready state
- **Finish Time**: when process complete and exit from a system
- **Multiprogramming**: A number of programs which can be present in memory at the same time.
- **Jobs**: It is a type of program without any kind of user interaction.
- **User**: It is a kind of program having user interaction.
- **Process**: It is the reference that is used for both job and user.

Chapter 2

Project Description

Purpose:

The purpose of this project is to provide implementation of process scheduling through the GUI interface with the details of completion time, turn around time, waiting time to explain the working of various cpu schduling algorithms: **FCFS** (First Come First Serve) , **SJF** (Shortest Job First) , **SRTF** (Shortest Remaining Time Function) .

Problem statement:

Question 1: Write a GUI based application which does the following:

- Receive arrival time and burst time of different processes from the user.

- After pressing the solve button the Gantt chart and the complete table would be printed based on the selected CPU Scheduling algorithm.

Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
A	1	6	7	6	0
B	3	13	40	37	24
C	5	12	27	22	10
D	7	8	15	8	0
			Average	73 / 4 = 18.25	34 / 4 = 8.5

- You have to implement **FCFS, SJF (Preemptive and non-preemptive)** scheduling algorithm.

Chapter 3

System Requirements

- **Hardware specification Processor**
  - **i5 Core Processor**
  - **Clock speed:** 2.5GHz
  - **Monitor:** 1024 * 768 Resolution Color
  - **Keyboard:** QWERTY
  - **RAM:** 1 GB
  - **Input Output Console for interaction**

- **Software specification**
  - **Operating system:** Windows10, or Linux Ubuntu (20.04 LTS).
  - **IDE:** Intellij Idea (community edition), Eclipse IDE, Net Beans.
  - **JDK**: Java JDK of 8 or above.

SOURCE CODE

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

class FCFS_Algorithm {
    ArrayList<String> ProcessIDs;
    ArrayList<Integer> ArrivalTime;
    ArrayList<Integer> BurstTime;
    ArrayList<String> DisposalProcessIDs;
    ArrayList<Integer> DisposalArrivalTime;
    ArrayList<Integer> DisposalBurstTime;
    LinkedList<String> FinalProcessID;
    LinkedList<Integer> FinalArrivalTime;
    LinkedList<Integer> FinalBurstTime;

    LinkedList<Integer> FinalSortedArrivalTime;
    LinkedList<Integer> FinalSortedBurstTime;
    LinkedList<String> FinalSortedProcessIDs;
    LinkedList<Integer> CompletionTime;
    LinkedList<Integer> TurnAroundTime;
    LinkedList<Integer> WaitingTime;



    void FCFS_Algorithm_driver(ArrayList Temp_Arrival_List, ArrayList
Temp_Burst_List, int SIZE) {

        ProcessIDs = new ArrayList<>();
        ArrivalTime = new ArrayList<>(Temp_Arrival_List);
        BurstTime = new ArrayList<>(Temp_Burst_List);

        for (int i = 0; i < SIZE; i++) {
            ProcessIDs.add("P " + Integer.toString(i + 1));
        }

        DisposalProcessIDs = new ArrayList<>(ProcessIDs);
        DisposalArrivalTime = new ArrayList<>(ArrivalTime);
        DisposalBurstTime = new ArrayList<>(BurstTime);
        FinalProcessID = new LinkedList<>();
        FinalArrivalTime = new LinkedList<>();
        FinalBurstTime = new LinkedList<>();
```

```java
        CompletionTime = new LinkedList<>();
        TurnAroundTime = new LinkedList<>();
        WaitingTime = new LinkedList<>();


        while (!DisposalArrivalTime.isEmpty()) {
            int TMP_MIN_Arrival = DisposalArrivalTime.get(0);
            int TMP_MIN_Burst = DisposalBurstTime.get(0);
            String TMP_MIN_PID = DisposalProcessIDs.get(0);
            int INDEX = 0;

            for (int i = 0; i < DisposalArrivalTime.size(); i++) {
                if (TMP_MIN_Arrival > DisposalArrivalTime.get(i)) {
                    TMP_MIN_Arrival = DisposalArrivalTime.get(i);
                    TMP_MIN_Burst = DisposalBurstTime.get(i);
                    TMP_MIN_PID = DisposalProcessIDs.get(i);
                    INDEX = i;
                }
            }

            FinalArrivalTime.addLast(TMP_MIN_Arrival);
            FinalBurstTime.addLast(TMP_MIN_Burst);
            FinalProcessID.addLast(TMP_MIN_PID);
            DisposalArrivalTime.remove(INDEX);
            DisposalBurstTime.remove(INDEX);
            DisposalProcessIDs.remove(INDEX);
        }

        FinalSortedArrivalTime = new LinkedList<>(FinalArrivalTime);
        FinalSortedBurstTime = new LinkedList<>(FinalBurstTime);
        FinalSortedProcessIDs = new LinkedList<>(FinalProcessID);

        int Completion_Time = 0;
        boolean Passer = true;
        while (!FinalArrivalTime.isEmpty()) {

            if (Completion_Time < FinalArrivalTime.getFirst()) {
                Completion_Time++;
            } else {
                if (Passer) {
                    Completion_Time = Completion_Time + FinalBurstTime.getFirst();
                    CompletionTime.addLast(Completion_Time);
                    Passer = false;
                    FinalArrivalTime.removeFirst();
                    FinalBurstTime.removeFirst();
```

```
                    FinalProcessID.removeFirst();
                } else {
                    Completion_Time = Completion_Time + FinalBurstTime.getFirst();
                    CompletionTime.addLast(Completion_Time);
                    FinalArrivalTime.removeFirst();
                    FinalBurstTime.removeFirst();
                    FinalProcessID.removeFirst();
                }
            }
        }

        for (int i = 0; i < FinalSortedProcessIDs.size(); i++) {
            TurnAroundTime.add(i, CompletionTime.get(i) -
FinalSortedArrivalTime.get(i));
            WaitingTime.add(i, TurnAroundTime.get(i) -
FinalSortedBurstTime.get(i));
        }

        System.out.println(FinalSortedProcessIDs);
        System.out.println(FinalSortedArrivalTime);
        System.out.println(FinalSortedBurstTime);
        System.out.println(CompletionTime);
        System.out.println(TurnAroundTime);
        System.out.println(WaitingTime);
    }

}

class SJF_Algorithm {
    ArrayList<String> OrgProcessID;
    ArrayList<Integer> OriginalArrivalTime;
    ArrayList<Integer> OriginalBurstTime;
    ArrayList<String> DuplicateProcessID;
    ArrayList<Integer> DuplicateArrivalTime;
    ArrayList<Integer> DuplicateBurstTime;
    LinkedList<String> ProcessID_ll;
    LinkedList<Integer> ArrivalTime_ll;
    LinkedList<Integer> BurstTime_ll;
    LinkedList<Integer> CompletionTime_ll;
    LinkedList<Integer> TurnAroundTime_ll;
    LinkedList<Integer> WaitingTime_ll;

    void SJF_Algorithm_driver(ArrayList local_SJF_arrival , ArrayList
local_SJF_burst , int SJF_SIZE)
    {
```

```
        OrgProcessID = new ArrayList<>();
        OriginalArrivalTime = new ArrayList<>(local_SJF_arrival);
        OriginalBurstTime = new ArrayList<>(local_SJF_burst);

        for(int i=0; i<SJF_SIZE; i++)
        {
            OrgProcessID.add("P "+ Integer.toString(i+1));
        }

        DuplicateArrivalTime = new ArrayList<>(OriginalArrivalTime);
        DuplicateBurstTime = new ArrayList<>(OriginalBurstTime);
        DuplicateProcessID = new ArrayList<>(OrgProcessID);
        BurstTime_ll = new LinkedList<>();
        ArrivalTime_ll = new LinkedList<>();
        ProcessID_ll = new LinkedList<>();
        CompletionTime_ll = new LinkedList<>();
        TurnAroundTime_ll = new LinkedList<>();
        WaitingTime_ll = new LinkedList<>();

        int CHECKPOINT =0 ; /* outside the loop */
        int EXTENDER = 0;
        int MINIMUM = DuplicateArrivalTime.get(0);
        int INDEX = 0;
        int MINIMUM_BurstTime;
        int INDEX_BurstTime;

        while(DuplicateProcessID.size() != 0 && DuplicateArrivalTime.size() != 0
&& DuplicateBurstTime.size() != 0)
        {
            if(CHECKPOINT == 0)
            {
                for(int i=0; i<DuplicateArrivalTime.size();i++)
                {
                    if(MINIMUM > DuplicateArrivalTime.get(i))
                    {
                        MINIMUM = DuplicateArrivalTime.get(i);
                        INDEX = i;
                    }
                }

                BurstTime_ll.addLast(DuplicateBurstTime.get(INDEX));
                ProcessID_ll.addLast(DuplicateProcessID.get(INDEX));
                ArrivalTime_ll.addLast(DuplicateArrivalTime.get(INDEX));
```

```
                EXTENDER = DuplicateArrivalTime.get(INDEX) +
DuplicateBurstTime.get(INDEX);
                CompletionTime_ll.addLast(EXTENDER);
                CHECKPOINT++;

                DuplicateBurstTime.remove(INDEX);
                DuplicateArrivalTime.remove(INDEX);
                DuplicateProcessID.remove(INDEX);
            }

            else{
                MINIMUM_BurstTime = DuplicateBurstTime.get(0);
                INDEX_BurstTime = 0;
                for(int i=0;  i<DuplicateArrivalTime.size(); i++)
                {
                    if(EXTENDER >= DuplicateArrivalTime.get(i))
                    {
                        if( MINIMUM_BurstTime > DuplicateBurstTime.get(i))
                        {
                            MINIMUM_BurstTime = DuplicateBurstTime.get(i);
                            INDEX_BurstTime = i;
                        }
                    }
                }

                BurstTime_ll.addLast(DuplicateBurstTime.get(INDEX_BurstTime));
                ProcessID_ll.addLast(DuplicateProcessID.get(INDEX_BurstTime));
                ArrivalTime_ll.addLast(DuplicateArrivalTime.get(INDEX_BurstTime));

                CHECKPOINT++;
                EXTENDER = EXTENDER + BurstTime_ll.getLast();
                CompletionTime_ll.addLast(EXTENDER);

                DuplicateProcessID.remove(INDEX_BurstTime);
                DuplicateBurstTime.remove(INDEX_BurstTime);
                DuplicateArrivalTime.remove(INDEX_BurstTime);
            }

        }

        for(int i=0; i<CompletionTime_ll.size(); i++)
        {
            TurnAroundTime_ll.addLast(CompletionTime_ll.get(i) -
ArrivalTime_ll.get(i));
        }
```

```
        for(int i=0; i<TurnAroundTime_ll.size(); i++)
        {
            WaitingTime_ll.addLast(TurnAroundTime_ll.get(i) -
BurstTime_ll.get(i));
        }

        System.out.println(ProcessID_ll);
        System.out.println(BurstTime_ll);
        System.out.println(ArrivalTime_ll);
        System.out.println(CompletionTime_ll);
        System.out.println(TurnAroundTime_ll);
        System.out.println(WaitingTime_ll);
    }

}

class SRTF_Algorithm {

    ArrayList<String> SRTF_OrgProcessID;
    ArrayList<Integer> SRTF_OrgArrivalTime;
    ArrayList<Integer> SRTF_OrgBurstTime;

    ArrayList<String> DuplicateSRTF_ProcessID;
    ArrayList<Integer> DuplicateSRTF_ArrivalTime;
    ArrayList<Integer> DuplicateSRTF_BurstTime;
    ArrayList<Integer> ReferencedSRTF_BurstTime;


    LinkedList<Integer> SRTF_Completion_ll;
    LinkedList<Integer> SRTF_TurnAroundTime_ll;
    LinkedList<Integer> SRTF_WaitingTime_ll;
    LinkedList<Integer> Process_Saver_ll;

    void SRTF_Algorithm_driver(ArrayList local_SRTF_arrival_time , ArrayList
local_SRTF_burst_time , int SRTF_SIZE)
    {
        SRTF_OrgProcessID = new ArrayList<>();
        SRTF_OrgArrivalTime = new ArrayList<>(local_SRTF_arrival_time);
        SRTF_OrgBurstTime = new ArrayList<>(local_SRTF_burst_time);
        ReferencedSRTF_BurstTime = new ArrayList<>();

        for(int i=0 ; i<SRTF_SIZE; i++)
        {
            ReferencedSRTF_BurstTime.add(i,0);
```

```
        }
        for(int i=0; i<SRTF_SIZE; i++)
        {
            SRTF_OrgProcessID.add("P "+ Integer.toString(i+1));
        }


        DuplicateSRTF_ProcessID = new ArrayList<>(SRTF_OrgProcessID);
        DuplicateSRTF_ArrivalTime = new ArrayList<>(SRTF_OrgArrivalTime);
        DuplicateSRTF_BurstTime = new ArrayList<>(SRTF_OrgBurstTime);
        SRTF_Completion_ll = new LinkedList<>(SRTF_OrgBurstTime);
        SRTF_TurnAroundTime_ll = new LinkedList<>();
        SRTF_WaitingTime_ll = new LinkedList<>();
        Process_Saver_ll = new LinkedList<>();

        int MIN_ArrTime = DuplicateSRTF_ArrivalTime.get(0);
        int CURRENT_INDEX = 0;
        int CURRENT_TIME;
        int CURRENT_BURST_TIME;
        int EXTENDER;

        for(int i=0 ; i<SRTF_SIZE; i++)
        {
            if(MIN_ArrTime > DuplicateSRTF_ArrivalTime.get(i))
            {
                MIN_ArrTime = DuplicateSRTF_ArrivalTime.get(i);
                CURRENT_INDEX = i;
            }
        }

        CURRENT_TIME = DuplicateSRTF_ArrivalTime.get(CURRENT_INDEX);
        CURRENT_BURST_TIME = DuplicateSRTF_BurstTime.get(CURRENT_INDEX);

        while( !DuplicateSRTF_BurstTime.equals(ReferencedSRTF_BurstTime))
        {
            CURRENT_TIME++;
            CURRENT_BURST_TIME --;

            if(CURRENT_BURST_TIME == 0 && Process_Saver_ll.isEmpty())
            {
                SRTF_Completion_ll.set(CURRENT_INDEX,CURRENT_TIME);
            }

            if(CURRENT_BURST_TIME ==0 && !Process_Saver_ll.isEmpty())
            {
```

```
                DuplicateSRTF_BurstTime.set(CURRENT_INDEX,CURRENT_BURST_TIME);
                SRTF_Completion_ll.set(CURRENT_INDEX,CURRENT_TIME);
                CURRENT_BURST_TIME  =
DuplicateSRTF_BurstTime.get(Process_Saver_ll.getFirst());
                CURRENT_INDEX = Process_Saver_ll.getFirst();
                Process_Saver_ll.removeFirst();
            }

            DuplicateSRTF_BurstTime.set(CURRENT_INDEX , CURRENT_BURST_TIME);
            EXTENDER = CURRENT_TIME + 1;


            for(int i=0; i<DuplicateSRTF_BurstTime.size(); i++)
            {
                if(EXTENDER > DuplicateSRTF_ArrivalTime.get(i) &&
DuplicateSRTF_ArrivalTime.get(i) >= CURRENT_TIME )
                {
                    if(DuplicateSRTF_BurstTime.get(i) < CURRENT_BURST_TIME)
                    {
                        Process_Saver_ll.addFirst(CURRENT_INDEX);
                        CURRENT_BURST_TIME = DuplicateSRTF_BurstTime.get(i);
                        CURRENT_INDEX = i;
                    }

                    else {
                        boolean flag = false;
                        for(int j=0; j<Process_Saver_ll.size(); j++)
                        {

                            if(Process_Saver_ll.get(j) == i)
                            {
                                flag = true;
                            }
                        }
                        if(flag == false)
                        {
                            Process_Saver_ll.addFirst(i);
                        }

                    }

                }
            }

        }
```

```
        for(int i=0 ; i<SRTF_Completion_ll.size(); i++)
        {
            SRTF_TurnAroundTime_ll.addLast(SRTF_Completion_ll.get(i) -
SRTF_OrgArrivalTime.get(i));
        }

        for(int i=0; i<SRTF_TurnAroundTime_ll.size(); i++)
        {
            SRTF_WaitingTime_ll.addLast(SRTF_TurnAroundTime_ll.get(i) -
SRTF_OrgBurstTime.get(i));
        }
        System.out.println(SRTF_OrgProcessID);
        System.out.println(SRTF_OrgArrivalTime);
        System.out.println(SRTF_OrgBurstTime);
        System.out.println(SRTF_Completion_ll);
        System.out.println(SRTF_TurnAroundTime_ll);
        System.out.println(SRTF_WaitingTime_ll);
    }

}

public class operatingsystemproject{
    JFrame FoundationFrame;
    JPanel Panel;
    JComboBox Obj;
    JLabel Heading;
    JLabel Arrival_TimesGUI;
    JLabel Burst_TimesGUI;
    JTextField Arrival_TimesGUI_Field;
    JTextField Burst_TimesGUI_Field;
    JButton Ok;


     operatingsystemproject()
    {

        FoundationFrame = new JFrame("BHAVISH_1000015397");
        FoundationFrame.setSize(800,800);
        FoundationFrame.setLayout(null);

        Panel = new JPanel();
        Panel.setBounds(200,200,350,300);
        Panel.setLayout(new GridLayout(7,1,0,15));
```

```
        String names [] = {"FCFS (First Come First Serve)","SJF (Shortest Job
First)","SRTF (Shortest Remaining Time First)"};
        Obj = new JComboBox(names);
        Obj.setFont(new Font("Sans Serif",Font.BOLD,16));
        Obj.setForeground(Color.BLACK);

        Heading = new JLabel("Select any Algorithm");
        Heading.setFont(new Font("Serif" , Font.BOLD , 24));
        Heading.setForeground(Color.BLUE);

        Arrival_TimesGUI = new JLabel("Arrival Times");
        Arrival_TimesGUI.setFont(new Font("Serif" , Font.BOLD , 20));
        Arrival_TimesGUI.setForeground(Color.BLUE);

        Burst_TimesGUI = new JLabel("Burst Times");
        Burst_TimesGUI.setFont(new Font("Serif" , Font.BOLD , 20));
        Burst_TimesGUI.setForeground(Color.BLUE);

        Arrival_TimesGUI_Field = new JTextField();
        Burst_TimesGUI_Field = new JTextField();

        Ok = new JButton("SUBMIT");
        Ok.setBackground(Color.pink);
        Ok.setForeground(Color.black);


        Panel.add(Heading);
        Panel.add(Obj);
        Panel.add(Arrival_TimesGUI);
        Panel.add(Arrival_TimesGUI_Field);
        Panel.add(Burst_TimesGUI);
        Panel.add(Burst_TimesGUI_Field);
        Panel.add(Ok);
        FoundationFrame.add(Panel);

        FoundationFrame.setLayout(null);
        FoundationFrame.setVisible(true);
        FoundationFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Ok.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if(e.getSource() == Ok)
                {
                    boolean passer = true;
```

```
                    int choice = Obj.getSelectedIndex();
                    String column [] ={"Process_ID" , "Arrival_Time",
"Burst_Time","Completion_Time","TurnAround_Time","Waiting_Time"};
                    String data[][];

                    String Arrival_str = Arrival_TimesGUI_Field.getText();
                    String Burst_str = Burst_TimesGUI_Field.getText();

                    String Temporary_Transfer_Arrival_ArrayString [] =
Arrival_str.split("\\s");
                    String Temporary_Transfer_Burst_ArrayString [] =
Burst_str.split("\\s");

                    if(Arrival_str.isEmpty() || Burst_str.isEmpty() ||
(Arrival_str.isEmpty() && Burst_str.isEmpty()))
                    {
                        JOptionPane.showMessageDialog(FoundationFrame , "Mention
the Arrival times or Burst times or both!!!");
                        passer =  false;
                    }

                    if(Arrival_str.length() != Burst_str.length())
                    {
                        JOptionPane.showMessageDialog(FoundationFrame,"Arrival
times and Burst Times should be equal");
                        passer = false;
                    }

                    ArrayList<Integer> Temporary_ArrivalList = new ArrayList<>();
                    ArrayList<Integer> Temporary_BurstList = new ArrayList<>();


                    try
                    {
                        for(int i=0 ;
i<Temporary_Transfer_Arrival_ArrayString.length ; i++)
                        {

Temporary_ArrivalList.add(Integer.parseInt(Temporary_Transfer_Arrival_ArrayString[
i]));

Temporary_BurstList.add(Integer.parseInt(Temporary_Transfer_Burst_ArrayString[i]))
;
                        }
```

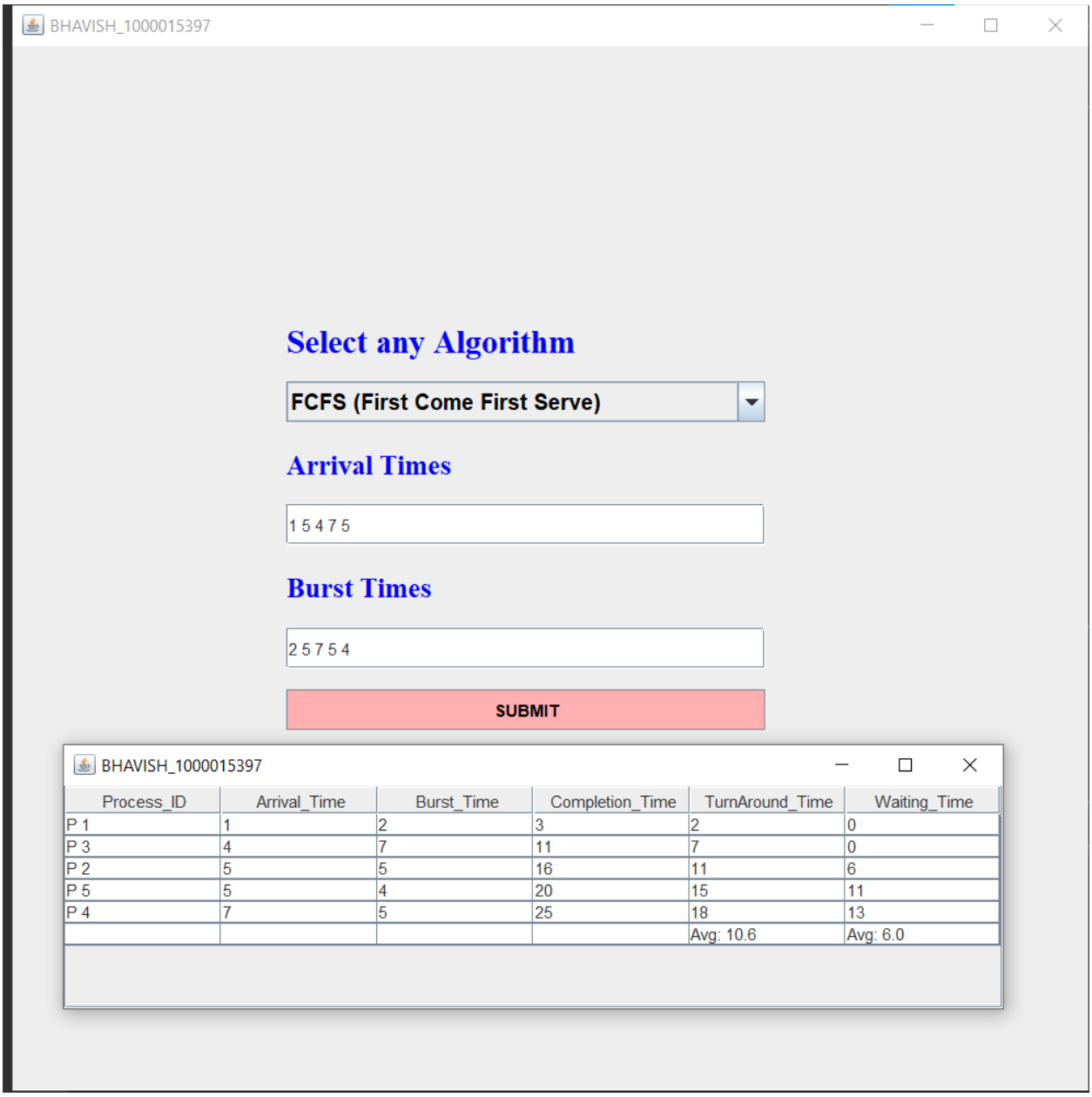
```
                    }
                  catch(Exception exception)
                  {
                      JOptionPane.showMessageDialog(FoundationFrame,"Arrival times
and Burst times should be same");
                      passer = false;
                  }

                    if(passer)
                    {
                        double sum_tat = 0;
                        double sum_wt = 0 ;
                        double avg_tat;
                        double avg_wt;

                       if(choice == 0)
                       {
                           data = new String[Temporary_ArrivalList.size()+1][6];
                           FCFS_Algorithm OBJ_FCFS = new FCFS_Algorithm();
                           OBJ_FCFS.FCFS_Algorithm_driver(Temporary_ArrivalList
,Temporary_BurstList , Temporary_ArrivalList.size());

                           for(int i=0; i<Temporary_ArrivalList.size();i++)
                           {
                                   data[i][0] =
OBJ_FCFS.FinalSortedProcessIDs.get(i);
                                   data[i][1] =
String.valueOf(OBJ_FCFS.FinalSortedArrivalTime.get(i));
                                   data[i][2] =
String.valueOf(OBJ_FCFS.FinalSortedBurstTime.get(i));
                                   data[i][3] =
String.valueOf(OBJ_FCFS.CompletionTime.get(i));
                                   data[i][4]  =
String.valueOf(OBJ_FCFS.TurnAroundTime.get(i));
                                   data[i][5] =
String.valueOf(OBJ_FCFS.WaitingTime.get(i));

                                   sum_tat = sum_tat +
OBJ_FCFS.TurnAroundTime.get(i);
                                   sum_wt = sum_wt + OBJ_FCFS.WaitingTime.get(i);

                           }
                           avg_tat = sum_tat/Temporary_ArrivalList.size();
                           avg_wt = sum_wt/Temporary_ArrivalList.size();
```

```
                            data[Temporary_ArrivalList.size()][4] = "Avg:
"+avg_tat;
                            data[Temporary_ArrivalList.size()][5] = "Avg: "+avg_wt;

                            JFrame FCFS_frame = new JFrame("BHAVISH_1000015397");
                            FCFS_frame.setSize(700,200);

                            JTable FCFS_table = new JTable(data,column);
                            JScrollPane FCFS_SP = new JScrollPane(FCFS_table);
                            FCFS_frame.add(FCFS_SP);
                            FCFS_frame.setVisible(true);

 FCFS_frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                        }

                        if(choice == 1)
                        {
                            data = new String[Temporary_ArrivalList.size()+1][6];
                            SJF_Algorithm OBJ_SJF = new SJF_Algorithm();
                            OBJ_SJF.SJF_Algorithm_driver(Temporary_ArrivalList,
Temporary_BurstList , Temporary_ArrivalList.size());

                            for(int i=0; i<Temporary_ArrivalList.size();i++)
                            {
                                data[i][0] = OBJ_SJF.ProcessID_ll.get(i);
                                data[i][1] =
String.valueOf(OBJ_SJF.ArrivalTime_ll.get(i));
                                data[i][2] =
String.valueOf(OBJ_SJF.BurstTime_ll.get(i));
                                data[i][3] =
String.valueOf(OBJ_SJF.CompletionTime_ll.get(i));
                                data[i][4]  =
String.valueOf(OBJ_SJF.TurnAroundTime_ll.get(i));
                                data[i][5] =
String.valueOf(OBJ_SJF.WaitingTime_ll.get(i));

                                sum_tat = sum_tat +
OBJ_SJF.TurnAroundTime_ll.get(i);
                                sum_wt = sum_wt + OBJ_SJF.WaitingTime_ll.get(i);
                            }

                            avg_tat = sum_tat/Temporary_ArrivalList.size();
                            avg_wt = sum_wt/Temporary_ArrivalList.size();
```

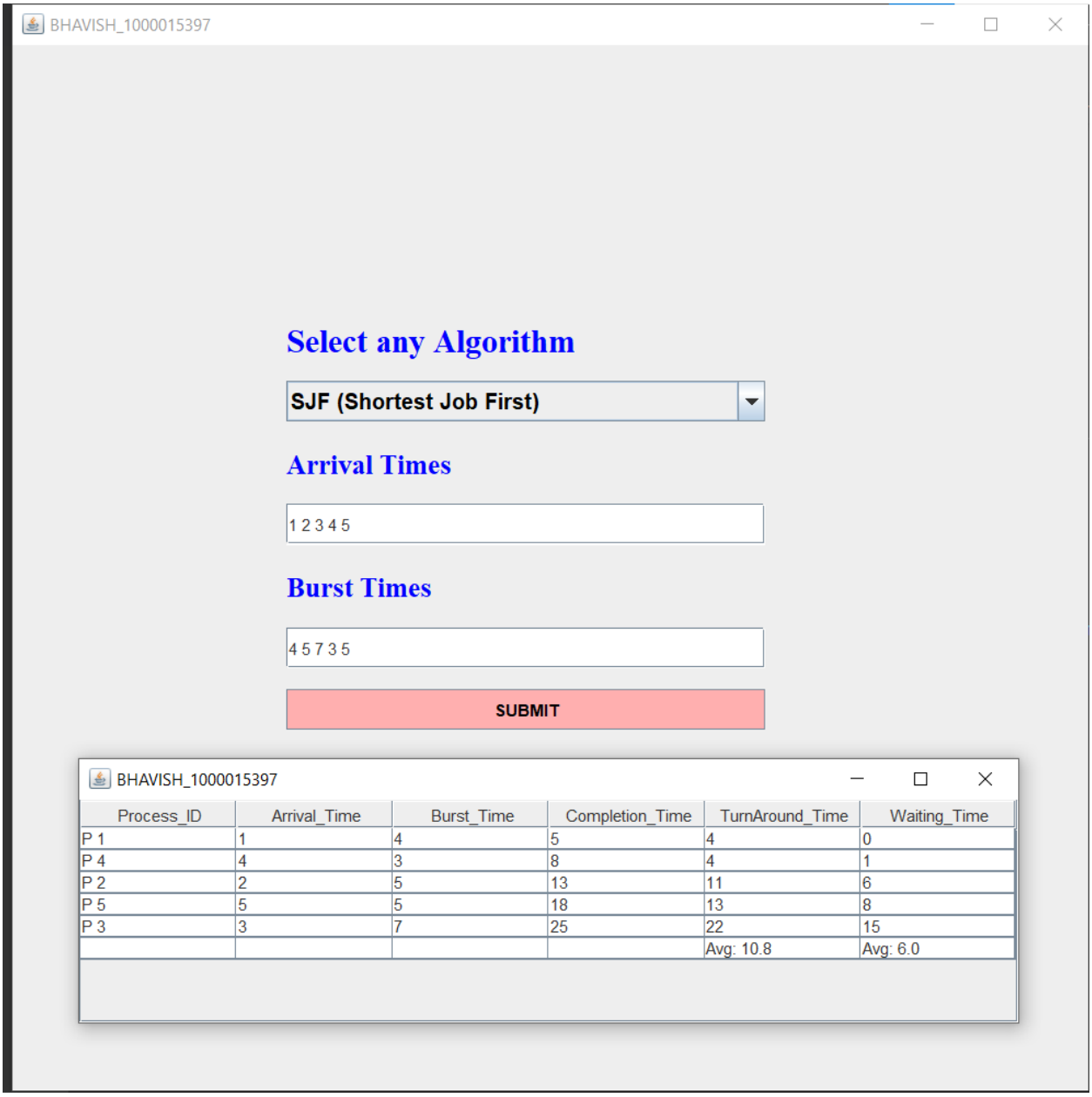
```
                            data[Temporary_ArrivalList.size()][4] = "Avg:
"+avg_tat;
                            data[Temporary_ArrivalList.size()][5] = "Avg: "+avg_wt;

                            JFrame SJF_frame = new JFrame("BHAVISH_1000015397");
                            SJF_frame.setSize(700,200);
                            JTable SJF_table = new JTable(data,column);
                            JScrollPane SJF_SP = new JScrollPane(SJF_table);
                            SJF_frame.add(SJF_SP);
                            SJF_frame.setVisible(true);

 SJF_frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

                        }

                        if(choice == 2)
                        {
                            data = new String[Temporary_ArrivalList.size()+1][6];
                            SRTF_Algorithm OBJ_SRTF = new SRTF_Algorithm();

 OBJ_SRTF.SRTF_Algorithm_driver(Temporary_ArrivalList,Temporary_BurstList ,
Temporary_ArrivalList.size());

                            for(int i=0; i<Temporary_ArrivalList.size();i++)
                            {
                                data[i][0] = OBJ_SRTF.SRTF_OrgProcessID.get(i);
                                data[i][1] =
String.valueOf(OBJ_SRTF.SRTF_OrgArrivalTime.get(i));
                                data[i][2] =
String.valueOf(OBJ_SRTF.SRTF_OrgBurstTime.get(i));
                                data[i][3] =
String.valueOf(OBJ_SRTF.SRTF_Completion_ll.get(i));
                                data[i][4]  =
String.valueOf(OBJ_SRTF.SRTF_TurnAroundTime_ll.get(i));
                                data[i][5] =
String.valueOf(OBJ_SRTF.SRTF_WaitingTime_ll.get(i));

                                sum_tat = sum_tat +
OBJ_SRTF.SRTF_TurnAroundTime_ll.get(i);
                                sum_wt = sum_wt +
OBJ_SRTF.SRTF_WaitingTime_ll.get(i);
                            }

                            avg_tat = sum_tat/Temporary_ArrivalList.size();
                            avg_wt = sum_wt/Temporary_ArrivalList.size();
```

```
                            data[Temporary_ArrivalList.size()][4] = "Avg:
"+avg_tat;
                            data[Temporary_ArrivalList.size()][5] = "Avg: "+avg_wt;

                            JFrame SRTF_frame = new JFrame("BHAVISH_1000015397");
                            SRTF_frame.setSize(700,200);
                            JTable FCFS_table = new JTable(data,column);
                            JScrollPane SRTF_SP = new JScrollPane(FCFS_table);
                            SRTF_frame.add(SRTF_SP);
                            SRTF_frame.setVisible(true);

 SRTF_frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                        }
                    }
                }
            }
        });

    }

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

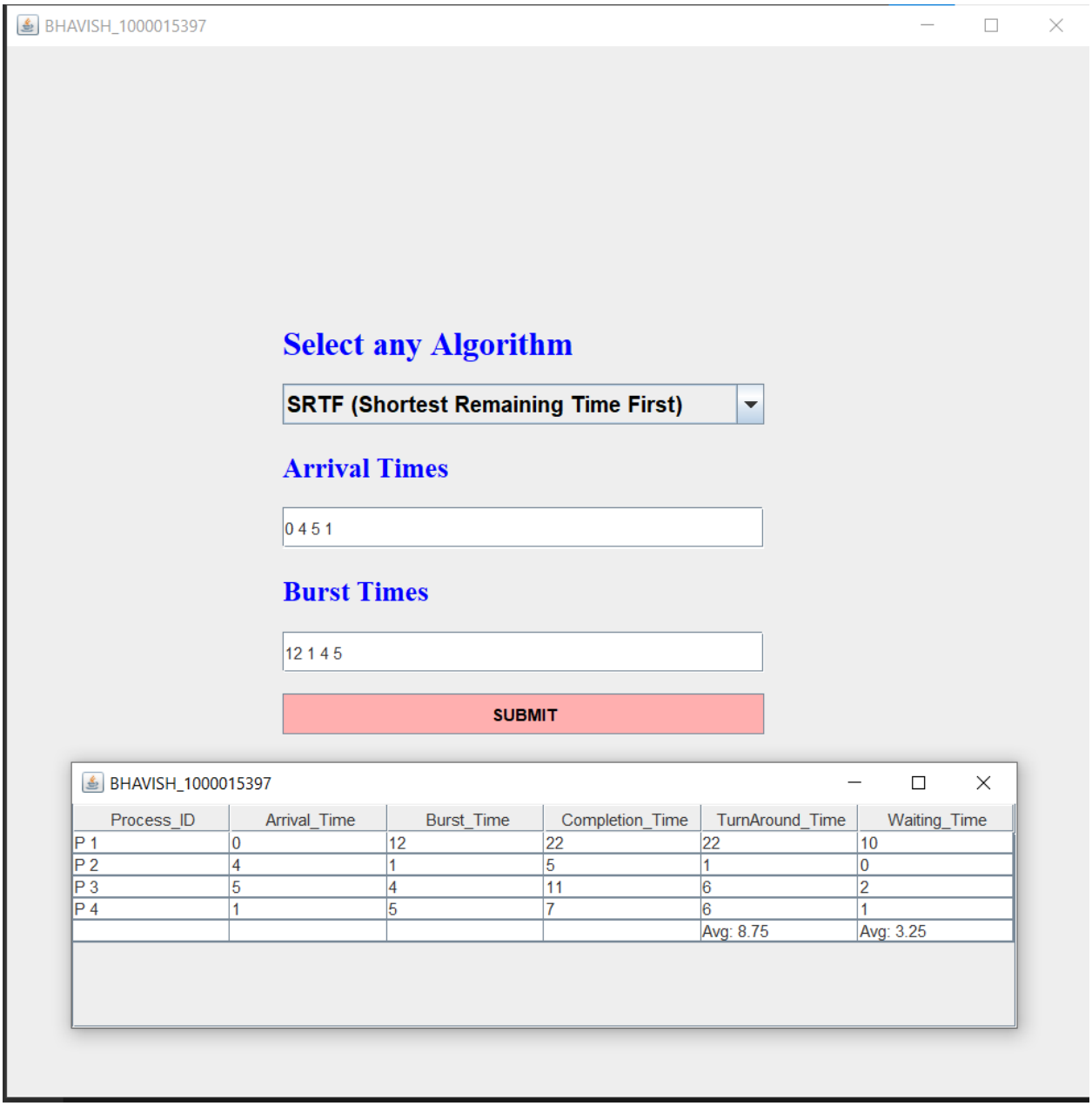
Output



Process_ID	Arrival_Time	Burst_Time	Completion_Time	TurnAround_Time	Waiting_Time
P 1	1	2	3	2	0
P 3	4	7	11	7	0
P 2	5	5	16	11	6
P 5	5	4	20	15	11
P 4	7	5	25	18	13
				Avg: 10.6	Avg: 6.0



Process_ID	Arrival_Time	Burst_Time	Completion_Time	TurnAround_Time	Waiting_Time
P 1	1	4	5	4	0
P 4	4	3	8	4	1
P 2	2	5	13	11	6
P 5	5	5	18	13	8
P 3	3	7	25	22	15
				Avg: 10.8	Avg: 6.0



Process_ID	Arrival_Time	Burst_Time	Completion_Time	TurnAround_Time	Waiting_Time
P 1	0	12	22	22	10
P 2	4	1	5	1	0
P 3	5	4	11	6	2
P 4	1	5	7	6	1
				Avg: 8.75	Avg: 3.25

Conclusion

The project **Process Scheduling Solver** is very much handy to solve the various problems related to process scheduling problems and to get the proper understanding regarding the working of CPU regarding the scheduling the processes to get maximum utilisation of CPU and fastest response as much as possible through the algorithms like **FCFS** (First Come First Serve), **SJF** (Shortest Job First), **SRTF** (Shortest Remaing Time Function). The **input (front end)** is implemented through the **GUI** implementation and the **scheduling algorithms (back end)** is implemented with the help of **java collection framework**. It was the whole great journey regarding to the process and the project is successful with all the source code and modules working properly without having any error that's why output values of all the inputs are correct.

Bibliography

- https://www.javatpoint.com/java-map **[Visited On : 16/04/2022]**
- https://boonsuen.com/process-scheduling-solver **[Visited On : 17/04/2022]**
- https://www.geeksforgeeks.org/java-program-to-sort-linkedhashmap-by-values/ **[Visited On : 17/04/2022]**
- https://www.thejavaprogrammer.com/java-program-shortest-job-first-sjf-scheduling/ **[Visited On : 17/04/2022]**
- https://www.thejavaprogrammer.com/java-program-first-come-first-serve-fcfs-schedulingalgorithm/ **[Visited On : 19/04/2022]**
- https://www.javatpoint.com/os-srtf-scheduling-algorithm **[Visited On : 20/04/2022]**