LABORATORY PRACTICE I

Lab Journal

NAME :- OJUS P. JAISWAL

YEAR & DIV :- TE A

ROLL NO. :- TACO19108

SEAT NO. :- S191094290

Index

Assignment No. A1

Problem Statement :- Design suitable Data structures and implement Pass-I and Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. The output of Pass-I (intermediate code file and symbol table) should be input for Pass-II.

Solution :-

Program :

a) Pass-I =>

```
package A1.A1a;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

import A1.A1a.LitTuple;
import A1.A1a.SymTuple;
import A1.A1a.Tuple;

class Tuple {
	//m_class specifies class of the mnemonic such as IS, DL, or AD
	String mnemonic, m_class, opcode;
	int length;

	Tuple() {}

	Tuple(String s1, String s2, String s3, String s4) {
		mnemonic = s1;
```

```
                m_class = s2;
                opcode = s3;
                length = Integer.parseInt(s4);

        }
}

class SymTuple {
        String symbol, address;
        int length;

        SymTuple(String s1, String s2, int i1) {
                symbol = s1;
                address = s2;
                length = i1;
        }
}

class LitTuple {
        String literal, address;
        int length;

        LitTuple() {}

        LitTuple(String s1,  String s2, int i1) {
                literal = s1;
                address = s2;
                length = i1;
        }
}

public class Assembler_PassOne_V2{

        static int lc,iSymTabPtr=0, iLitTabPtr=0, iPoolTabPtr=0;
        static int poolTable[] = new int[10];
        static Map<String,Tuple> MOT;
        static Map<String,SymTuple> symtable;
        static ArrayList<LitTuple> littable;
        static Map<String, String> regAddressTable;
        static PrintWriter out_pass2;
        static PrintWriter out_pass1;
        static int line_no;


        public static void main(String[] args) throws Exception{
                initializeTables();
```

```
		System.out.println("====== PASS 1 OUTPUT ======\n");
		pass1();

	}

	static void pass1() throws Exception {
		BufferedReader input = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A1/A1a/input.txt")));
		out_pass1 = new PrintWriter(new FileWriter("src/A1/A1a/output_pass1.txt"),
true);
		PrintWriter out_symtable = new PrintWriter(new
FileWriter("src/A1/A1a/symtable.txt"), true);
		PrintWriter out_littable = new PrintWriter(new
FileWriter("src/A1/A1a/littable.txt"), true);
		String s;
		//Read from input file one line at a time
		lc=0;
		while((s = input.readLine()) != null) {
			StringTokenizer st = new StringTokenizer(s, " ", false);
			//For each line, separate out the tokens
			String s_arr[] = new String[st.countTokens()];
			for(int i=0 ; i < s_arr.length ; i++) {
				s_arr[i] = st.nextToken();
			}
			if(s_arr.length == 0){
				continue;
			}
			int curIndex = 0;
			//Contains a value in the label field
			if(s_arr.length == 3){
				String label = s_arr[0];
				insertIntoSymTab(label,lc+"");
				curIndex = 1;
			}

			String curToken = s_arr[curIndex];
			//Get current tuple from opcode Table
			Tuple curTuple = MOT.get(curToken);

			String intermediateStr="";
			//Analyze current token to check class of token (IS, DL, AD)
			if(curTuple.m_class.equalsIgnoreCase("IS")){
				intermediateStr += lc + " (" +  curTuple.m_class + "," +
curTuple.opcode + ")  ";
				lc += curTuple.length;
				intermediateStr += processOperands(s_arr[curIndex+1]);
```

```
                        }
                        else if(curTuple.m_class.equalsIgnoreCase("AD")){
                                if(curTuple.mnemonic.equalsIgnoreCase("START")){
                                        intermediateStr += lc + " (" +  curTuple.m_class + "," +
curTuple.opcode + ")  ";
                                        lc = Integer.parseInt(s_arr[curIndex+1]);
                                        intermediateStr += "(C," + (s_arr[curIndex+1]) + ") ";
                                }
                                else if(curTuple.mnemonic.equalsIgnoreCase("LTORG")){
                                        intermediateStr +=processLTORG();
                                }
                                else if(curTuple.mnemonic.equalsIgnoreCase("END")){
                                        intermediateStr += lc + " (" +  curTuple.m_class + "," +
curTuple.opcode + ")  \n";
                                        intermediateStr +=processLTORG();
                                        //break;
                                }
                        }
                        else if(curTuple.m_class.equalsIgnoreCase("DL")){
                                intermediateStr += lc + " (" +  curTuple.m_class + "," +
curTuple.opcode + ")  ";
                                if(curTuple.mnemonic.equalsIgnoreCase("DS")){
                                        lc += Integer.parseInt(s_arr[curIndex+1]);
                                }
                                else if(curTuple.mnemonic.equalsIgnoreCase("DC")){
                                        lc += curTuple.length;
                                }
                                intermediateStr += "(C," + s_arr[curIndex+1] + ") ";
                        }
                        //Print the instruction in the intermediate file
                        System.out.println(intermediateStr);
                        out_pass1.println(intermediateStr);
                        //Add the length of the instruction in the location counter
                }
                //Close intermediate file
                out_pass1.flush();
                out_pass1.close();
                //Print symbol table
                System.out.println("====== Symbol Table ======");
                SymTuple tuple;
                Iterator<SymTuple> it = symtable.values().iterator();
                String tableEntry;
                while(it.hasNext()){
                        tuple = it.next();
                        tableEntry = tuple.symbol + "\t" + tuple.address ;
                        out_symtable.println(tableEntry);
```

```
                System.out.println(tableEntry);
        }
        out_symtable.flush();
        out_symtable.close();

        //Print literal table
        System.out.println("====== Literal Table ======");
        LitTuple litTuple;
        //Iterator<LitTuple> iterator = littable.values().iterator();
        tableEntry = "";
        for(int i=0; i<littable.size(); i++){
                litTuple = littable.get(i);
                tableEntry = litTuple.literal + "\t" + litTuple.address ;
                out_littable.println(tableEntry);
                System.out.println(tableEntry);
        }
        out_littable.flush();
        out_littable.close();
}

static String processLTORG(){
        //Process literal table and assign addresses to every literal in the table
        LitTuple litTuple;
        String intermediateStr = "";
        for(int i=poolTable[iPoolTabPtr-1]; i<littable.size(); i++){
                litTuple = littable.get(i);
                litTuple.address = lc+"";
                intermediateStr += lc + " (DL,02)  (C," + litTuple.literal + ") \n";
                lc++;
        }
        //Make a new entry in pool table;
        poolTable[iPoolTabPtr] = iLitTabPtr;
        iPoolTabPtr++;
        return intermediateStr;
}
static String processOperands(String operands){
        StringTokenizer st = new StringTokenizer(operands, ",", false);
        //Separate out the tokens separated by comma
        String s_arr[] = new String[st.countTokens()];
        for(int i=0 ; i < s_arr.length ; i++) {
                s_arr[i] = st.nextToken();
        }
        String intermediateStr = "", curToken;
        for(int i=0; i <s_arr.length; i++){
                curToken = s_arr[i];
                if(curToken.startsWith("=")){
```

```
                        //Operand is a literal
                        //Extract literal from the string
                        StringTokenizer str = new StringTokenizer(curToken, "'", false);
                        //Separate out the tokens separated by comma
                        String tokens[] = new String[str.countTokens()];
                        for(int j=0 ; j < tokens.length ; j++) {
                                tokens[j] = str.nextToken();
                        }
                        String literal = tokens[1];
                        insertIntoLitTab(literal,"");
                        intermediateStr += "(L," + (iLitTabPtr -1) + ")";
                }
                else if(regAddressTable.containsKey(curToken)){
                        //Operand is a register name
                        intermediateStr += "(RG," + regAddressTable.get(curToken) + ")
";
                }
                else{
                        //Operand is a symbol
                        insertIntoSymTab(curToken,"");
                        intermediateStr += "(S," + (iSymTabPtr -1) + ")";
                }
        }
        return intermediateStr;

    }
    static void insertIntoSymTab(String symbol, String address){
        //Check if the symbol is already present in the symbol table
        if(symtable.containsKey(symbol)== true){
                //Extract entry from symbol table
                SymTuple s = symtable.get(symbol);
                //Update its address field
                s.address = address;
        }
        else{
                //If symbol is not present in the symbol table, create a new entry
                symtable.put(symbol, new SymTuple(symbol, address, 1));
        }
        iSymTabPtr++;
    }

    static void insertIntoLitTab(String literal, String address){
        //If label is not present in the literal table, create a new entry
        littable.add(iLitTabPtr, new LitTuple(literal, address, 1));
        iLitTabPtr++;
    }
```

```
	static void initializeTables() throws Exception {
		symtable = new LinkedHashMap<>();
		littable = new ArrayList<>();
		regAddressTable = new HashMap<>();
		MOT = new HashMap<>();
		String s,mnemonic;
		BufferedReader br;
		br = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A1/A1a/mot.txt")));
		while((s = br.readLine()) != null) {
			StringTokenizer st = new StringTokenizer(s, " ", false);
			mnemonic = st.nextToken();
			MOT.put(mnemonic, (new Tuple(mnemonic, st.nextToken(),
st.nextToken(), st.nextToken())));
		}
		br.close();
		//Initiallize register address table
		regAddressTable.put("AREG", "1");
		regAddressTable.put("BREG", "2");
		regAddressTable.put("CREG", "3");
		regAddressTable.put("DREG", "4");

		//Initiallize pool table
		poolTable[iPoolTabPtr] = iLitTabPtr;
		iPoolTabPtr++;
	}
}
```

b) Pass-II =>

```
package A1.A1b;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.StringTokenizer;

import A1.A1b.Tuple;
import A1.A1b.SymTuple;
import A1.A1b.LitTuple;

	class Tuple {
		//m_class specifies class of the mnemonic such as IS, DL, or AD
		String mnemonic, m_class, opcode;
		int length;

		Tuple() {}

		Tuple(String s1, String s2, String s3, String s4) {
			mnemonic = s1;
			m_class = s2;
			opcode = s3;
			length = Integer.parseInt(s4);

		}
	}

	class SymTuple {
		String symbol, address, length;

		SymTuple(String s1, String s2, String i1) {
			symbol = s1;
			address = s2;
			length = i1;
		}
	}

	class LitTuple {
```

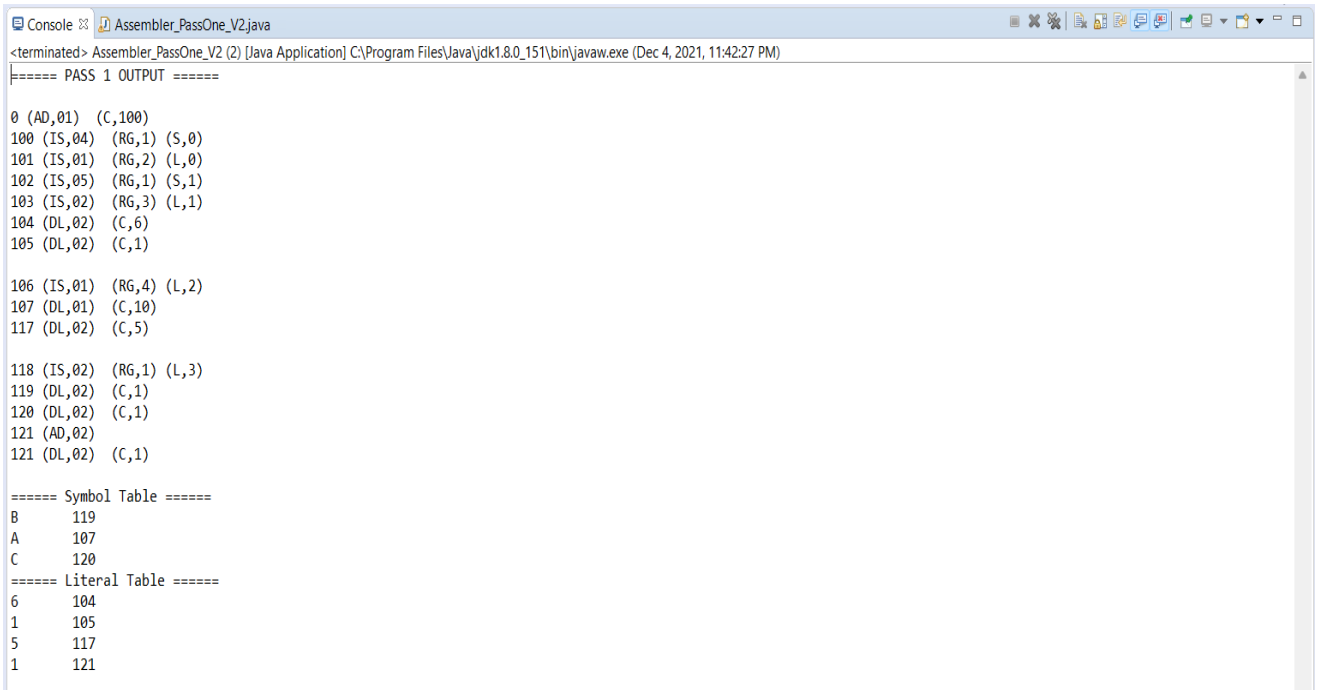
```
		String literal, address, length;

		LitTuple() {}

		LitTuple(String s1,  String s2, String i1) {
			literal = s1;
			address = s2;
			length = i1;
		}
	}

	public class Assembler_PassTwo {

	static int lc,iSymTabPtr=0, iLitTabPtr=0, iPoolTabPtr=0;
	static int poolTable[] = new int[10];
	static Map<String,Tuple> MOT;
	static ArrayList<SymTuple> symtable;
	static ArrayList<LitTuple> littable;
	static Map<String, String> regAddressTable;
	static PrintWriter out_pass2;



	static void initiallizeTables() throws Exception{
		symtable = new ArrayList<>();
		littable = new ArrayList<>();
		regAddressTable = new HashMap<>();
		//MOT = new HashMap<>();
		String s;
		BufferedReader br;
		br = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A1/A1b/symtable.txt")));
		while((s = br.readLine()) != null) {
			StringTokenizer st = new StringTokenizer(s, "\t", false);
			symtable.add(new SymTuple(st.nextToken(), st.nextToken(), ""));
		}
		br.close();
		br = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A1/A1b/littable.txt")));
		while((s = br.readLine()) != null) {
			StringTokenizer st = new StringTokenizer(s, "\t", false);
			littable.add(new LitTuple(st.nextToken(), st.nextToken(), ""));
		}
		br.close();
		//Initiallize register address table
		regAddressTable.put("AREG", "1");
```

```
		regAddressTable.put("BREG", "2");
		regAddressTable.put("CREG", "3");
		regAddressTable.put("DREG", "4");
	}

	static void pass2() throws Exception{
		BufferedReader input = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A1/A1b/output_pass1.txt")));
		out_pass2 = new PrintWriter(new FileWriter("src/A1/A1b/output_pass2.txt"),
true);
		String s;

		//Read from intermediate file one line at a time
		while((s = input.readLine()) != null) {
			//Replace all ( and ) characters by a blank string
			s=s.replaceAll("(\\()", " ");
			s=s.replaceAll("(\\))", " ");

			//For each line, separate out the tokens
			String ic_tokens[] = tokenizeString(s, " ");
			if(ic_tokens == null || ic_tokens.length==0){
				continue;
			}
			String output_str = "";

			//Second token contains mnemonic class and opcode
			String mnemonic_class = ic_tokens[1];
			//Separate the mnemonic and its opcode which are separated by a comma
			String m_tokens[] = tokenizeString(mnemonic_class, ",");

			//Write the second token as is in the output file

			if(m_tokens[0].equalsIgnoreCase("IS")){
				//First token is location counter which will be output as it is
				output_str += ic_tokens[0] + " ";
				//Output the opcode of the instruction
				output_str += m_tokens[1] + " ";
				String opr_tokens[];
				for(int i = 2; i <ic_tokens.length; i++){
					opr_tokens = tokenizeString(ic_tokens[i], ",");
					if(opr_tokens[0].equalsIgnoreCase("RG")){
						output_str += opr_tokens[1] + " ";
					}
					else if(opr_tokens[0].equalsIgnoreCase("S")){
						int index = Integer.parseInt(opr_tokens[1]);
						output_str += symtable.get(index).address + " ";
```

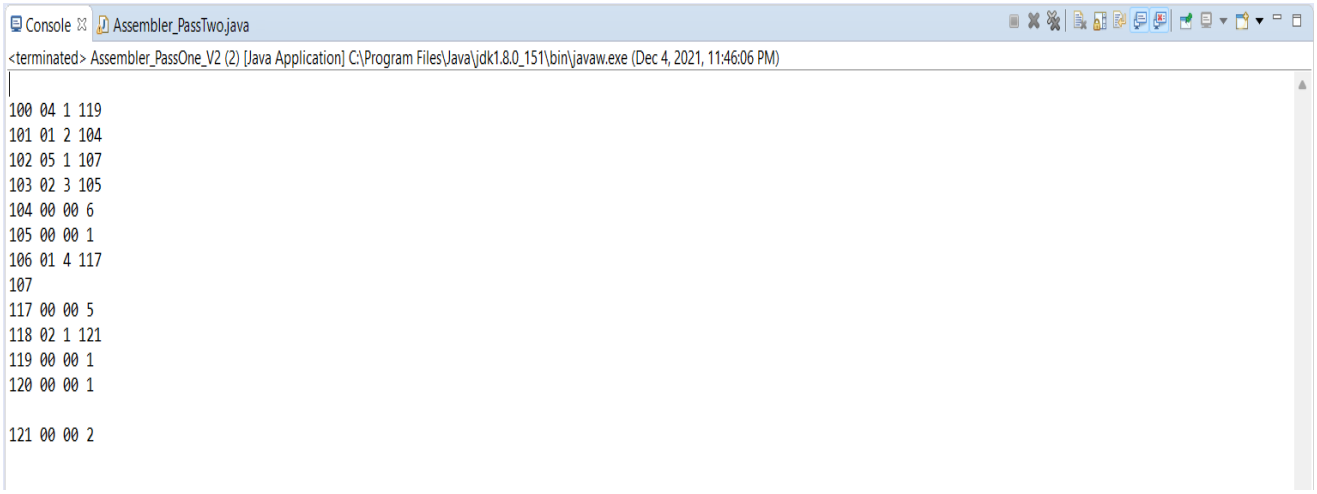
```
                                        }
                                        else if(opr_tokens[0].equalsIgnoreCase("L")){
                                                int index = Integer.parseInt(opr_tokens[1]);
                                                output_str += littable.get(index).address + " ";
                                        }
                                }
                        }
                        else if(m_tokens[0].equalsIgnoreCase("DL")){
                                //First token is location counter which will be output as it is
                                output_str += ic_tokens[0] + " ";
                                if(m_tokens[1].equalsIgnoreCase("02")){
                                        //Process for operands of mnemonic DC
                                        String opr_tokens[] = tokenizeString(ic_tokens[2], ",");
                                        output_str += "00 00 " + opr_tokens[1] + " ";
                                }
                        }
                        System.out.println(output_str);
                        out_pass2.println(output_str);

                }
        }

        static String[] tokenizeString(String str, String separator){
                StringTokenizer st = new StringTokenizer(str, separator, false);
                //Construct an array of the separated tokens
                String s_arr[] = new String[st.countTokens()];
                for(int i=0 ; i < s_arr.length ; i++) {
                        s_arr[i] = st.nextToken();
                }
                return s_arr;
        }
        public static void main(String[] args) throws Exception {
                initiallizeTables();
                pass2();
        }
}
```

Output :

a) Pass-I =>



b) Pass-II =>



Assignment No. A2

Problem Statement :- Design suitable data structures and implement Pass-I and Pass-II of a two-pass macro-processor. The output of Pass-I (MNT, MDT and intermediate code file without any macro definitions) should be input for Pass-II.

Solution :-

Program :

a) Pass-I =>

```
package A2.A2a;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

public class MacroProcessor_PassOne {
	static List<String> MDT;
	static Map<String, String> MNT;
	static int mntPtr, mdtPtr;
	static Map<String,String> ALA;


	public static void main(String[] args) {
		try{
			pass1();
		}catch(Exception ex){
			ex.printStackTrace();
		}
	}

	static void pass1() throws Exception {
```

```
		//Initiallize data structures
		MDT = new ArrayList<String>();
		MNT = new LinkedHashMap<String, String>();
		ALA = new HashMap<String,String>();
		mntPtr =0; mdtPtr = 0;

		BufferedReader input = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A2/A2a/input.txt")));
		PrintWriter out_pass1 = new PrintWriter(new
FileWriter("src/A2/A2a/output_pass1.txt"), true);
		PrintWriter out_mnt = new PrintWriter(new FileWriter("src/A2/A2a/MNT.txt"),
true);
		PrintWriter out_mdt = new PrintWriter(new FileWriter("src/A2/A2a/MDT.txt"),
true);

		String s;
		boolean processingMacroDefinition = false;
		boolean processMacroName = false;

		System.out.println("============= Pass 1 Output ==============");
		//Read from input file one line at a time
		while((s = input.readLine()) != null) {
			//For each line, separate out the tokens
			String s_arr[] = tokenizeString(s," ");

			//Analyze first token to check if it is a macro definition
			String curToken = s_arr[0];
			if(curToken.equalsIgnoreCase("MACRO")){
				processingMacroDefinition = true;
				processMacroName = true;
			}
			else if(processingMacroDefinition == true){
				if(curToken.equalsIgnoreCase("MEND")){
					MDT.add(mdtPtr++, s);
					processingMacroDefinition = false;
					continue;
				}
				//Insert Macro Name into MNT
				if(processMacroName == true){
					MNT.put(curToken, mdtPtr+"");
					mntPtr++;
					processMacroName = false;
					processArgumentList(s_arr[1]);
					MDT.add(mdtPtr,s);
					mdtPtr++;
```

```
                                continue;
                        }
                        //Convert arguments in the definition into corresponding indexed notation
                        //ADD &REG,&X == ADD #2,#1
                        String indexedArgList = processArguments(s_arr[1]);
                        MDT.add(mdtPtr++, curToken + " " + indexedArgList);
                    }
                    else{
                        //If line is not part of a Macro definition print the line as it is in the output file
                        System.out.println(s);
                        out_pass1.println(s);
                    }
                }
                input.close();
                //Print MNT
                System.out.println("============= MNT ==============");
                Iterator<String> itMNT = MNT.keySet().iterator();
                String key, mntRow, mdtRow;
                while(itMNT.hasNext()){
                    key = (String)itMNT.next();
                    mntRow = key + " " + MNT.get(key);
                    System.out.println(mntRow);
                    out_mnt.println(mntRow);

                }
                //Print MDT
                System.out.println("============= MDT ==============");
                for(int i = 0; i < MDT.size(); i++){
                    mdtRow = i + "  " + MDT.get(i);
                    System.out.println(mdtRow);
                    out_mdt.println(mdtRow);
                }
                out_pass1.close();
                out_mnt.close();
                out_mdt.close();
        }

        static void processArgumentList(String argList){
                StringTokenizer st = new StringTokenizer(argList, ",", false);
                //For each macro definition, remove contents of the HashMap
                //which are arguments from previous macro definition
                ALA.clear();
                int argCount = st.countTokens();
                //Put all arguments for current macro definition in the HashMap
```

```
		//with argument as key and argument index as value
		String curArg;
		for(int i=1 ; i <= argCount ; i++) {
			curArg = st.nextToken();
			if(curArg.contains("=")){
				curArg = curArg.substring(0,curArg.indexOf("="));
			}
			ALA.put(curArg, "#"+i);
		}
	}

	static String processArguments(String argList){
		StringTokenizer st = new StringTokenizer(argList, ",", false);
		int argCount = st.countTokens();
		String curArg, argIndexed;
		for(int i=0 ; i < argCount ; i++) {
			curArg = st.nextToken();
			argIndexed = ALA.get(curArg);
			argList = argList.replaceAll(curArg, argIndexed);
		}
		return argList;
	}

	static String[] tokenizeString(String str, String separator){
		StringTokenizer st = new StringTokenizer(str, separator, false);
		//Construct an array of the separated tokens
		String s_arr[] = new String[st.countTokens()];
		for(int i=0 ; i < s_arr.length ; i++) {
			s_arr[i] = st.nextToken();
		}
		return s_arr;
	}

}
```

b) Pass-II =>

```
package A2.A2b;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;


public class MacroProcessor_PassTwo {
	static List<String> MDT;
	static Map<String, String> MNT;
	static int mntPtr, mdtPtr;
	static List<String> formalParams, actualParams;

	public static void main(String[] args) {
		try{
			initiallizeTables();
			pass2();
		}catch(Exception ex){
			ex.printStackTrace();
		}
	}

	static void pass2() throws Exception {
		BufferedReader input = new BufferedReader(new InputStreamReader(new
FileInputStream("src/A2/A2b/output_pass1.txt")));
		PrintWriter out_pass2 = new PrintWriter(new
FileWriter("src/A2/A2b/output_pass2.txt"), true);

		System.out.println("============= Pass 2 Output ==============");
		//Read from input file one line at a time
		String s;
		while((s = input.readLine()) != null) {
			String s_arr[] = tokenizeString(s, " ");
			//First token will either be a mnemonic or a macro call
			if(MNT.containsKey(s_arr[0])){
				//It is a macro call
				//Create an array list of formal parameters
```

```
                        String actual_params[] = tokenizeString(s_arr[1], ",");
                        String param;
                        actualParams.clear();
                        for(int i =0; i <actual_params.length; i++){
                                param = actual_params[i];
                                if(param.contains("=")){
                                        //If parameter specified a default value, the value will
go in the list instead of param name
                                        param = param.substring(param.indexOf("=")+1,
param.length());
                                }
                                actualParams.add(param);
                        }
                        //Expand the macro call
                        mdtPtr = Integer.parseInt(MNT.get(s_arr[0]));
                        //Read macro definitaion starting from mdtPtr till MEND
                        String macroDef;
                        boolean createParamArray = true;
                        String def_tokens[] = {}, paramStr = "", printStr;
                        while(true){
                                //First line of macro definition is name and arglist
                                macroDef = MDT.get(mdtPtr);
                                if(createParamArray == true){
                                        createFormalParamList(macroDef);
                                        createParamArray = false;
                                }
                                else{
                                        //Tokenize line of macro definition
                                        def_tokens = tokenizeString(macroDef, " ");
                                        //If the line is MEND, exit loop
                                        if(def_tokens[0].equalsIgnoreCase("MEND")){
                                                break;
                                        }
                                        else{
                                                //Replace formal parameters with actual
parameters
                                                paramStr =
replaceFormalParams(def_tokens[1]);
                                        }
                                        printStr = "+" + def_tokens[0] + " " + paramStr;
                                        System.out.println(printStr);
                                        out_pass2.println(printStr);
                                }
                                mdtPtr++;
                        }
                }
```

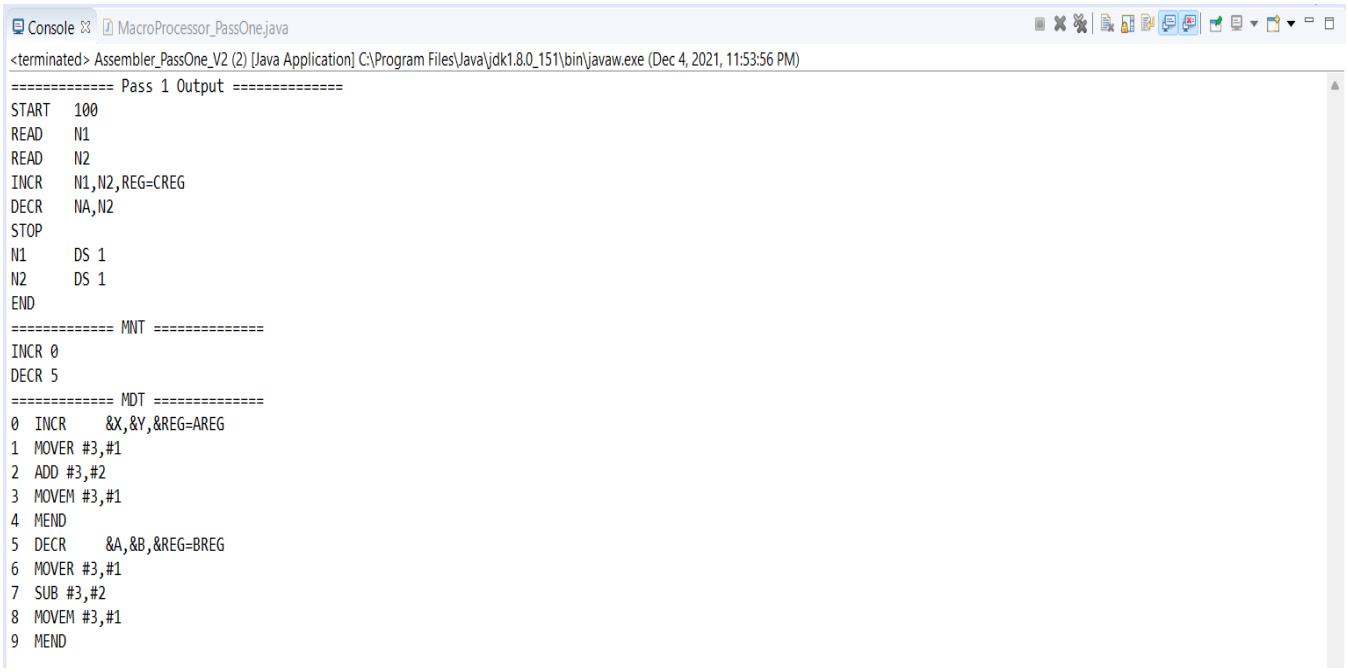
```
                else{
                        //It is a line of normal assembly code
                        //Print the line as it is in the output file
                        System.out.println(s);
                        out_pass2.println(s);
                }
        }
        input.close();
        out_pass2.close();
}

static String replaceFormalParams(String formalParamList){
        String returnStr = "";
        //Replace # by blank string
        formalParamList = formalParamList.replace("#", "");
        //Separate formal params
        String param_array[] = tokenizeString(formalParamList, ",");
        int index;
        String actualParam;
        //For every parameter in the formal parameter list
        for(int i = 0; i < param_array.length; i++){
                index = Integer.parseInt(param_array[i]);
                if(index <= actualParams.size()){
                        actualParam = actualParams.get(index-1);
                }
                else{
                        actualParam = formalParams.get(index-1);
                }
                returnStr += actualParam + ",";
        }

        //Strip last comma
        returnStr = returnStr.substring(0,returnStr.length() -1);
        return returnStr;
}

static void createFormalParamList(String macroDef){
        //By processing macro call generate array of actual parameters
        String argList, arg_array[];
        String s_arr[] = tokenizeString(macroDef, " ");
        //First array element will be macro name and second will be argument list
        argList = s_arr[1];
        //Separate the arguments in the list
        arg_array = tokenizeString(argList, ",");
        String param;
        formalParams.clear();
```

```
		for(int i=0; i <arg_array.length; i++){
			param = arg_array[i];
			if(param.contains("=")){
				//If parameter specified a default value, the value will go in the list instead of param name
				param = param.substring(param.indexOf("=")+1, param.length());
			}
			formalParams.add(param);
		}
	}

	static void initiallizeTables() throws Exception{
		MDT = new ArrayList<String>();
		MNT = new LinkedHashMap<String, String>();
		formalParams = new ArrayList<String>();
		actualParams = new ArrayList<String>();

		//Read contents of MNT.txt and create internal data structure
		BufferedReader br;
		String s;
		br = new BufferedReader(new InputStreamReader(new FileInputStream("src/A2/A2b/MNT.txt")));
		while((s = br.readLine()) != null) {
			StringTokenizer st = new StringTokenizer(s, " ", false);
			MNT.put(st.nextToken(), st.nextToken());
		}
		br.close();

		//Read contents of MDT.txt and create internal data structure
		br = new BufferedReader(new InputStreamReader(new FileInputStream("src/A2/A2b/MDT.txt")));
		while((s = br.readLine()) != null) {
			//For each line, separate out the tokens
			String s_arr[] = tokenizeString(s," ");
			if(s_arr.length == 0){
				continue;
			}
			int index = Integer.parseInt(s_arr[0]);

			if(s_arr.length == 2){
				MDT.add(index, s_arr[1]);
			}
			else if(s_arr.length == 3){
				MDT.add(index, s_arr[1] + " " + s_arr[2]);
			}
```

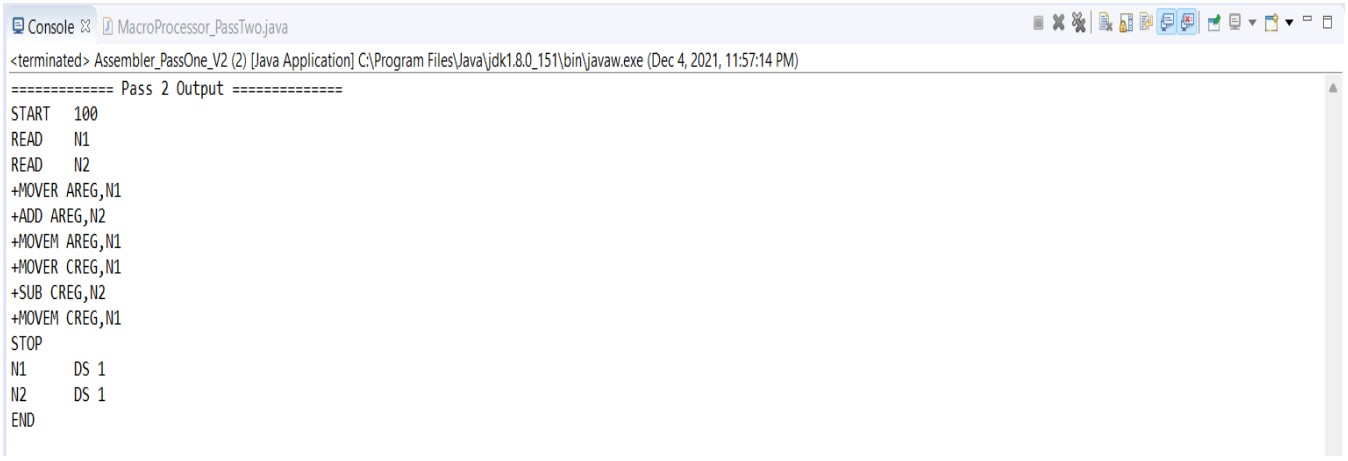
```
                }
                br.close();
        }

        static String[] tokenizeString(String str, String separator){
                StringTokenizer st = new StringTokenizer(str, separator, false);
                //Construct an array of the separated tokens
                String s_arr[] = new String[st.countTokens()];
                for(int i=0 ; i < s_arr.length ; i++) {
                        s_arr[i] = st.nextToken();
                }
                return s_arr;
        }
}
```

Output :

a) Pass-I =>



b) Pass-II =>



Assignment No. B2

Problem Statement :- Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) and Round Robin (Preemptive).

Solution :-

Program :

a) FCFS =>

```
/* FCFS */

package B2;

import java.io.*;
import java.util.Scanner;
public class FCFS
{
	public static void main(String args[])
	{
		int i,no_p,burst_time[],TT[],WT[];
		float avg_wait=0,avg_TT=0;
		burst_time=new int[50];
		TT=new int[50];
		WT=new int[50];
		WT[0]=0;
		Scanner s=new Scanner(System.in);
		System.out.println("Enter the number of process: ");
		no_p=s.nextInt();
		System.out.println("\nEnter Burst Time for processes:");
		for(i=0;i<no_p;i++)
		{
			System.out.print("\tP"+(i+1)+":  ");
			burst_time[i]=s.nextInt();
		}

		for(i=1;i<no_p;i++)
		{
			WT[i]=WT[i-1]+burst_time[i-1];
			avg_wait+=WT[i];
		}
		avg_wait/=no_p;
```

```
                for(i=0;i<no_p;i++)
                {
                        TT[i]=WT[i]+burst_time[i];
                        avg_TT+=TT[i];
                }
                avg_TT/=no_p;

        System.out.println("\n**********************************************************
**********");
                System.out.println("\tProcesses:");

        System.out.println("**********************************************************
*********");
                System.out.println("    Process\tBurst Time\tWaiting Time\tTurn Around Time");
                for(i=0;i<no_p;i++)
                {
                        System.out.println("\tP"+(i+1)+"\t    "+burst_time[i]+"\t\t    "+WT[i]+"\t\t
"+TT[i]);

                }
                System.out.println("\n-----------------------------------------------------------------");
                System.out.println("\nAverage waiting time : "+avg_wait);
                System.out.println("\nAverage Turn Around time : "+avg_TT+"\n");
        }
}
```

b) SJF (Preemptive) =>

```
/* SJF (Preemptive) */

package B2;

import java.util.*;

public class SJF {
public static void main (String args[])
{
Scanner sc=new Scanner(System.in);
System.out.println ("enter no of process:");
int n= sc.nextInt();
int pid[] = new int[n]; // it takes pid of process
int at[] = new int[n]; // at means arrival time
int bt[] = new int[n]; // bt means burst time
int ct[] = new int[n]; // ct means complete time
```

```
int ta[] = new int[n];// ta means turn around time
int wt[] = new int[n];  // wt means waiting time
int f[] = new int[n];  // f means it is flag it checks process is completed or not
int k[]= new int[n];   // it is also stores brust time
    int i, st=0, tot=0;
    float avgwt=0, avgta=0;

    for (i=0;i<n;i++)
    {
     pid[i]= i+1;
     System.out.println ("enter process " +(i+1)+ " arrival time:");
     at[i]= sc.nextInt();
     System.out.println("enter process " +(i+1)+ " burst time:");
     bt[i]= sc.nextInt();
     k[i]= bt[i];
     f[i]= 0;
    }

    while(true){
     int min=99,c=n;
     if (tot==n)
     break;

     for ( i=0;i<n;i++)
     {
     if ((at[i]<=st) && (f[i]==0) && (bt[i]<min))
     {
     min=bt[i];
     c=i;
     }
     }

     if (c==n)
     st++;
     else
     {
     bt[c]--;
     st++;
     if (bt[c]==0)
     {
     ct[c]= st;
     f[c]=1;
     tot++;
     }
     }
    }
```

```
    for(i=0;i<n;i++)
    {
     ta[i] = ct[i] - at[i];
     wt[i] = ta[i] - k[i];
     avgwt+= wt[i];
     avgta+= ta[i];
    }

    System.out.println("pid  arrival  burst  complete turn waiting");
    for(i=0;i<n;i++)
    {
     System.out.println(pid[i] +"\t"+ at[i]+"\t"+ k[i] +"\t"+ ct[i] +"\t"+ ta[i] +"\t"+ wt[i]);
    }

    System.out.println("\naverage tat is "+ (float)(avgta/n));
    System.out.println("average wt is "+ (float)(avgwt/n));
    sc.close();
}
}
```

c) Priority (Non-Preemptive) =>

```
/* Priority (Non-Preemptive) */

package B2;

import java.util.Scanner;

public class Priority
{

    int burstTime[];
    int priority[];
    int arrivalTime[];
    String[] processId;
    int numberOfProcess;

    void getProcessData(Scanner input)
    {
        System.out.print("Enter the number of Process for Scheduling          : ");
        int inputNumberOfProcess = input.nextInt();
        numberOfProcess = inputNumberOfProcess;
        burstTime = new int[numberOfProcess];
        priority = new int[numberOfProcess];
```

```
    arrivalTime = new int[numberOfProcess];
    processId = new String[numberOfProcess];
    String st = "P";
    for (int i = 0; i < numberOfProcess; i++)
    {
        processId[i] = st.concat(Integer.toString(i));
        System.out.print("Enter the burst time   for Process - " + (i) + " : ");
        burstTime[i] = input.nextInt();
        System.out.print("Enter the arrival time for Process - " + (i) + " : ");
        arrivalTime[i] = input.nextInt();
        System.out.print("Enter the priority     for Process - " + (i) + " : ");
        priority[i] = input.nextInt();
    }
}

void sortAccordingArrivalTimeAndPriority(int[] at, int[] bt, int[] prt, String[] pid)
{

    int temp;
    String stemp;
    for (int i = 0; i < numberOfProcess; i++)
    {

        for (int j = 0; j < numberOfProcess - i - 1; j++)
        {
            if (at[j] > at[j + 1])
            {
                //swapping arrival time
                temp = at[j];
                at[j] = at[j + 1];
                at[j + 1] = temp;

                //swapping burst time
                temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;

                //swapping priority
                temp = prt[j];
                prt[j] = prt[j + 1];
                prt[j + 1] = temp;

                //swapping process identity
                stemp = pid[j];
                pid[j] = pid[j + 1];
                pid[j + 1] = stemp;
```

```
            }
            //sorting according to priority when arrival timings are same
            if (at[j] == at[j + 1])
            {
               if (prt[j] > prt[j + 1])
               {
                  //swapping arrival time
                  temp = at[j];
                  at[j] = at[j + 1];
                  at[j + 1] = temp;

                  //swapping burst time
                  temp = bt[j];
                  bt[j] = bt[j + 1];
                  bt[j + 1] = temp;

                  //swapping priority
                  temp = prt[j];
                  prt[j] = prt[j + 1];
                  prt[j + 1] = temp;

                  //swapping process identity
                  stemp = pid[j];
                  pid[j] = pid[j + 1];
                  pid[j + 1] = stemp;

               }
            }
         }

      }
   }

   void priorityNonPreemptiveAlgorithm()
   {
      int finishTime[] = new int[numberOfProcess];
      int bt[] = burstTime.clone();
      int at[] = arrivalTime.clone();
      int prt[] = priority.clone();
      String pid[] = processId.clone();
      int waitingTime[] = new int[numberOfProcess];
      int turnAroundTime[] = new int[numberOfProcess];

      sortAccordingArrivalTimeAndPriority(at, bt, prt, pid);
```

```
        //calculating waiting & turn-around time for each process
        finishTime[0] = at[0] + bt[0];
        turnAroundTime[0] = finishTime[0] - at[0];
        waitingTime[0] = turnAroundTime[0] - bt[0];

        for (int i = 1; i < numberOfProcess; i++)
        {
           finishTime[i] = bt[i] + finishTime[i - 1];
           turnAroundTime[i] = finishTime[i] - at[i];
           waitingTime[i] = turnAroundTime[i] - bt[i];
        }
        float sum = 0;
        for (int n : waitingTime)
        {
           sum += n;
        }
        float averageWaitingTime = sum / numberOfProcess;

        sum = 0;
        for (int n : turnAroundTime)
        {
           sum += n;
        }
        float averageTurnAroundTime = sum / numberOfProcess;

        //print on console the order of processes along with their finish time & turn around time
        System.out.println("Priority Scheduling Algorithm : ");
        System.out.format("%20s%20s%20s%20s%20s%20s%20s\n",  "ProcessId",  "BurstTime",
"ArrivalTime", "Priority", "FinishTime", "WaitingTime", "TurnAroundTime");
        for (int i = 0; i < numberOfProcess; i++) {
           System.out.format("%20s%20d%20d%20d%20d%20d%20d\n", pid[i], bt[i], at[i], prt[i],
finishTime[i], waitingTime[i], turnAroundTime[i]);
        }

        System.out.format("%100s%20f%20f\n",          "Average",          averageWaitingTime,
averageTurnAroundTime);
    }

    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        Priority obj = new Priority();
        obj.getProcessData(input);
        obj.priorityNonPreemptiveAlgorithm();
    }
}
```

d) Round Robin (Preemptive) =>

```
/* Round Robin (Preemptive) */

package B2;

import java.util.*;

public class RoundRobin{
	private static Scanner inp = new Scanner(System.in);
	//Driver Code
	public static void main(String[] args){
		int n,tq, timer = 0, maxProccessIndex = 0;
		float avgWait = 0, avgTT = 0;
		System.out.print("\nEnter the time quanta : ");
		tq = inp.nextInt();
		System.out.print("\nEnter the number of processess : ");
		n = inp.nextInt();
		int arrival[] = new int[n];
		int burst[] = new int[n];
		int wait[] = new int[n];
		int turn[] = new int[n];
		int queue[] = new int[n];
		int temp_burst[] = new int[n];
		boolean complete[] = new boolean[n];

		System.out.print("\nEnter the arrival time of the processess : ");
		for(int i = 0; i < n; i++)
			arrival[i] = inp.nextInt();

		System.out.print("\nEnter the burst time of the processess : ");
		for(int i = 0; i < n; i++){
			burst[i] = inp.nextInt();
			temp_burst[i] = burst[i];
		}

		for(int i = 0; i < n; i++){ //Initializing the queue and complete array
			complete[i] = false;
			queue[i] = 0;
		}
		while(timer < arrival[0]) //Incrementing Timer until the first process arrives
			timer++;
		queue[0] = 1;

		while(true){
			boolean flag = true;
```

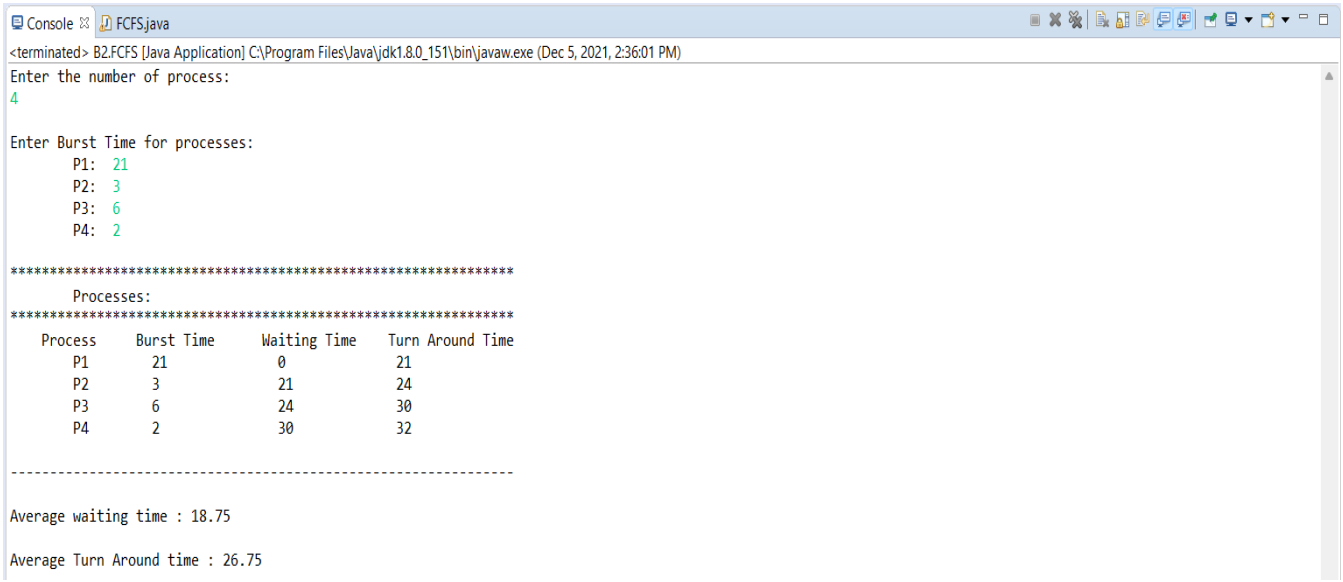
```
                for(int i = 0; i < n; i++){
                        if(temp_burst[i] != 0){
                                flag = false;
                                break;
                        }
                }
                if(flag)
                        break;

                for(int i = 0; (i < n) && (queue[i] != 0); i++){
                        int ctr = 0;
                        while((ctr < tq) && (temp_burst[queue[0]-1] > 0)){
                                temp_burst[queue[0]-1] -= 1;
                                timer += 1;
                                ctr++;

                                //Updating the ready queue until all the processes arrive
                                checkNewArrival(timer, arrival, n, maxProccessIndex,
queue);
                        }
                        if((temp_burst[queue[0]-1] == 0) && (complete[queue[0]-1] ==
false)){
                                turn[queue[0]-1] = timer;       //turn currently stores exit
times
                                complete[queue[0]-1] = true;
                        }

                        //checks whether or not CPU is idle
                        boolean idle = true;
                        if(queue[n-1] == 0){
                                for(int k = 0; k < n && queue[k] != 0; k++){
                                        if(complete[queue[k]-1] == false){
                                                idle = false;
                                        }
                                }
                        }
                        else
                                idle = false;

                        if(idle){
                                timer++;
                                checkNewArrival(timer, arrival, n, maxProccessIndex,
queue);
                        }
```

```
                        //Maintaining the entries of processes after each preemption in the
ready Queue
                        queueMaintainence(queue,n);
                    }
                }

                for(int i = 0; i < n; i++){
                    turn[i] = turn[i] - arrival[i];
                    wait[i] = turn[i] - burst[i];
                }

                System.out.print("\nProgram No.\tArrival Time\tBurst Time\tWait
Time\tTurnAround Time"
                                        + "\n");
                for(int i = 0; i < n; i++){
                    System.out.print(i+1+"\t\t"+arrival[i]+"\t\t"+burst[i]
                                            +"\t\t"+wait[i]+"\t\t"+turn[i]+ "\n");
                }
                for(int i =0; i< n; i++){
                    avgWait += wait[i];
                    avgTT += turn[i];
                }
                System.out.print("\nAverage wait time : "+(avgWait/n)
                                        +"\nAverage Turn Around Time : "+(avgTT/n));
        }
        public static void queueUpdation(int queue[],int timer,int arrival[],int n, int
maxProccessIndex){
                int zeroIndex = -1;
                for(int i = 0; i < n; i++){
                    if(queue[i] == 0){
                        zeroIndex = i;
                        break;
                    }
                }
                if(zeroIndex == -1)
                    return;
                queue[zeroIndex] = maxProccessIndex + 1;
        }

        public static void checkNewArrival(int timer, int arrival[], int n, int maxProccessIndex,int
queue[]){
                if(timer <= arrival[n-1]){
                    boolean newArrival = false;
                    for(int j = (maxProccessIndex+1); j < n; j++){
                        if(arrival[j] <= timer){
                            if(maxProccessIndex < j){
```

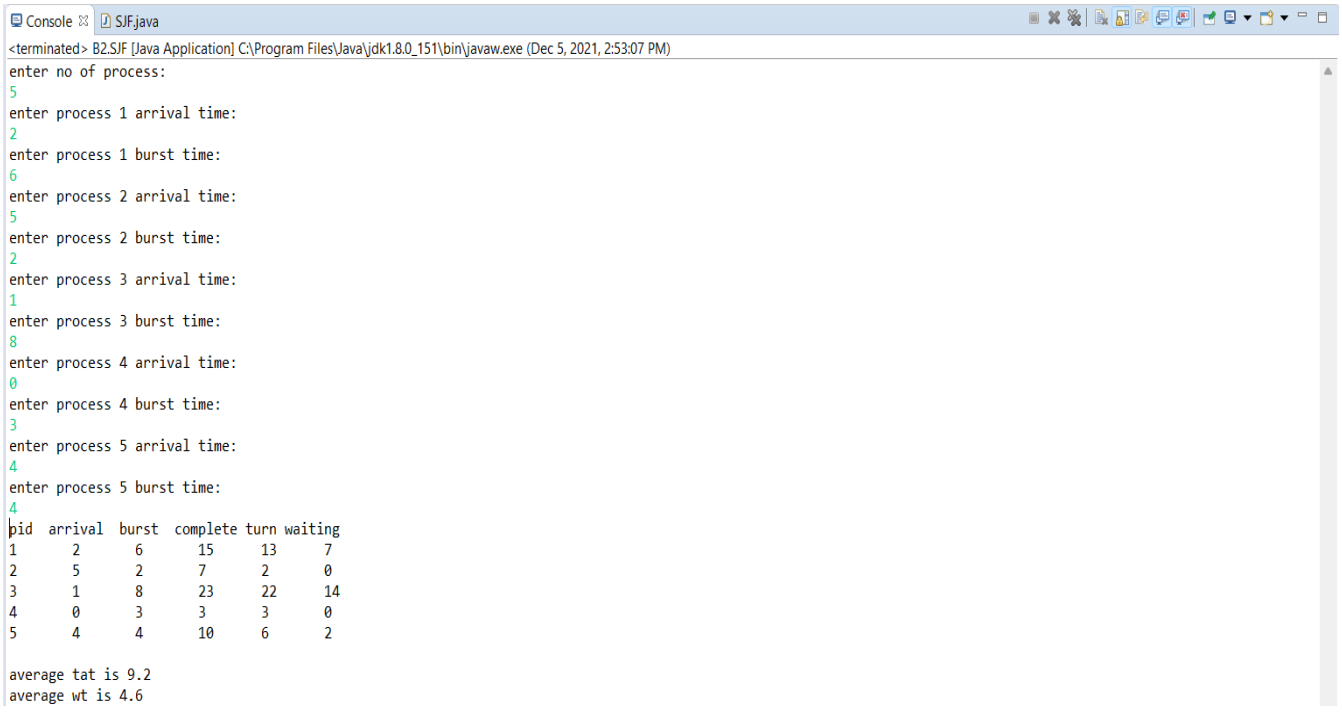
```
                                        maxProccessIndex = j;
                                        newArrival = true;
                                    }
                                }
                            }
                            if(newArrival) //adds the index of the arriving process(if any)
                                queueUpdation(queue,timer,arrival,n, maxProccessIndex);
                        }
                    }

                    public static void queueMaintainence(int queue[], int n){

                        for(int i = 0; (i < n-1) && (queue[i+1] != 0) ; i++){
                            int temp = queue[i];
                            queue[i] = queue[i+1];
                            queue[i+1] = temp;
                        }
                    }
                }
```

Output :

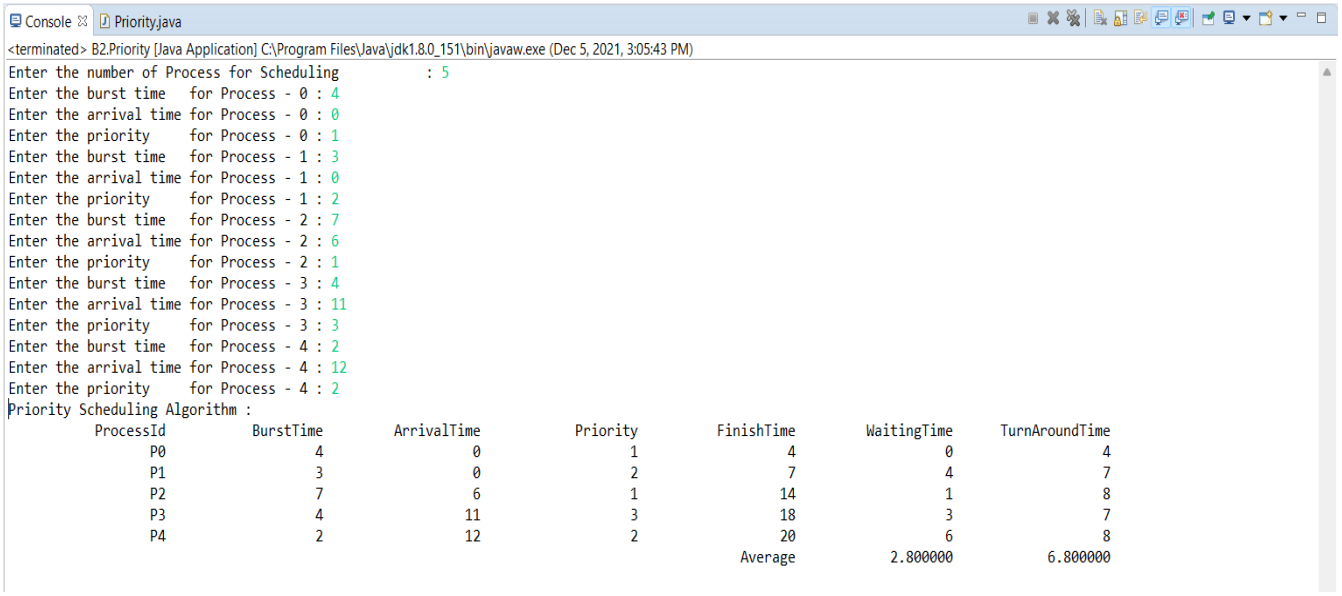
a) FCFS =>



b) SJF (Preemptive) =>

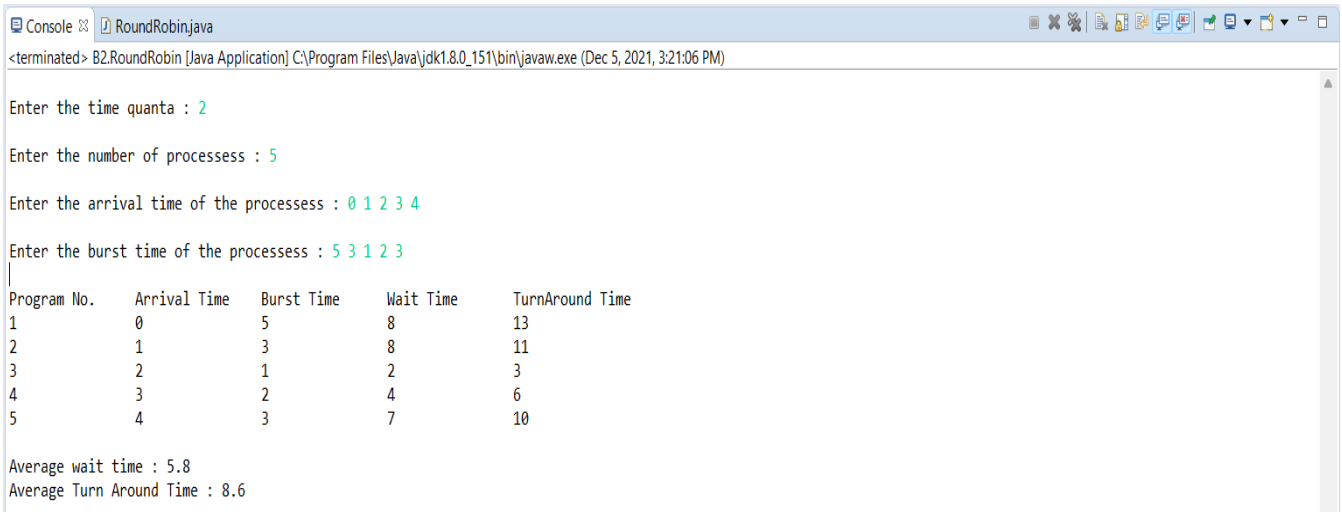


c) Priority (Non-Preemptive) =>



```
Console  Priority.java
<terminated> B2.Priority [Java Application] C:\Program Files\Java\jdk1.8.0_151\bin\javaw.exe (Dec 5, 2021, 3:05:43 PM)
Enter the number of Process for Scheduling          : 5
Enter the burst time   for Process - 0 : 4
Enter the arrival time for Process - 0 : 0
Enter the priority     for Process - 0 : 1
Enter the burst time   for Process - 1 : 3
Enter the arrival time for Process - 1 : 0
Enter the priority     for Process - 1 : 2
Enter the burst time   for Process - 2 : 7
Enter the arrival time for Process - 2 : 6
Enter the priority     for Process - 2 : 1
Enter the burst time   for Process - 3 : 4
Enter the arrival time for Process - 3 : 11
Enter the priority     for Process - 3 : 3
Enter the burst time   for Process - 4 : 2
Enter the arrival time for Process - 4 : 12
Enter the priority     for Process - 4 : 2
Priority Scheduling Algorithm :
          ProcessId           BurstTime         ArrivalTime           Priority         FinishTime        WaitingTime      TurnAroundTime
                 P0                   4                   0                   1                   4                   0                   4
                 P1                   3                   0                   2                   7                   4                   7
                 P2                   7                   6                   1                  14                   1                   8
                 P3                   4                  11                   3                  18                   3                   7
                 P4                   2                  12                   2                  20                   6                   8
                                                                                             Average          2.800000          6.800000
```

d) Round Robin (Preemptive) =>



```
Console  RoundRobin.java
<terminated> B2.RoundRobin [Java Application] C:\Program Files\Java\jdk1.8.0_151\bin\javaw.exe (Dec 5, 2021, 3:21:06 PM)

Enter the time quanta : 2

Enter the number of processess : 5

Enter the arrival time of the processess : 0 1 2 3 4

Enter the burst time of the processess : 5 3 1 2 3

Program No.     Arrival Time    Burst Time      Wait Time       TurnAround Time
1               0               5               8               13
2               1               3               8               11
3               2               1               2               3
4               3               2               4               6
5               4               3               7               10

Average wait time : 5.8
Average Turn Around Time : 8.6
```

Assignment No. B3

Problem Statement :- Write a program to simulate Memory placement strategies – best fit, first fit, next fit and worst fit.

Solution :-

Program :

```
/* Memory Placement Strategies */

package B3;

import java.util.Arrays;
class First
{
   // Method to allocate memory to
   // blocks as per First fit algorithm
   static void firstFit(int blockSize[], int m,
                        int processSize[], int n)
   {
      // Stores block id of the
      // block allocated to a process
      int allocation[] = new int[n];

      // Initially no block is assigned to any process
      for (int i = 0; i < allocation.length; i++)
         allocation[i] = -1;

      // pick each process and find suitable blocks
      // according to its size ad assign to it
      for (int i = 0; i < n; i++)
      {
         for (int j = 0; j < m; j++)
         {
            if (blockSize[j] >= processSize[i])
            {
               // allocate block j to p[i] process
               allocation[i] = j;

               // Reduce available memory in this block.
               blockSize[j] -= processSize[i];
```

```
                    break;
                }
            }
        }



System.out.println("\nProcess No.\tProcess Size\tBlock no.");
        for (int i = 0; i < n; i++)
        {
            System.out.print(" " + (i+1) + "\t\t" +
                             processSize[i] + "\t\t");
            if (allocation[i] != -1)
                System.out.print(allocation[i] + 1);
            else
                System.out.print("Not Allocated");
            System.out.println();
        }
    }
    static void bestFit(int blockSize[], int m, int processSize[],
            int n)
{
// Stores block id of the block allocated to a
// process
int allocation[] = new int[n];

// Initially no block is assigned to any process
for (int i = 0; i < allocation.length; i++)
allocation[i] = -1;

// pick each process and find suitable blocks
// according to its size ad assign to it
for (int i=0; i<n; i++)
{
// Find the best fit block for current process
int bestIdx = -1;
for (int j=0; j<m; j++)
{
if (blockSize[j] >= processSize[i])
{
if (bestIdx == -1)
bestIdx = j;
else if (blockSize[bestIdx] > blockSize[j])
bestIdx = j;
}
```

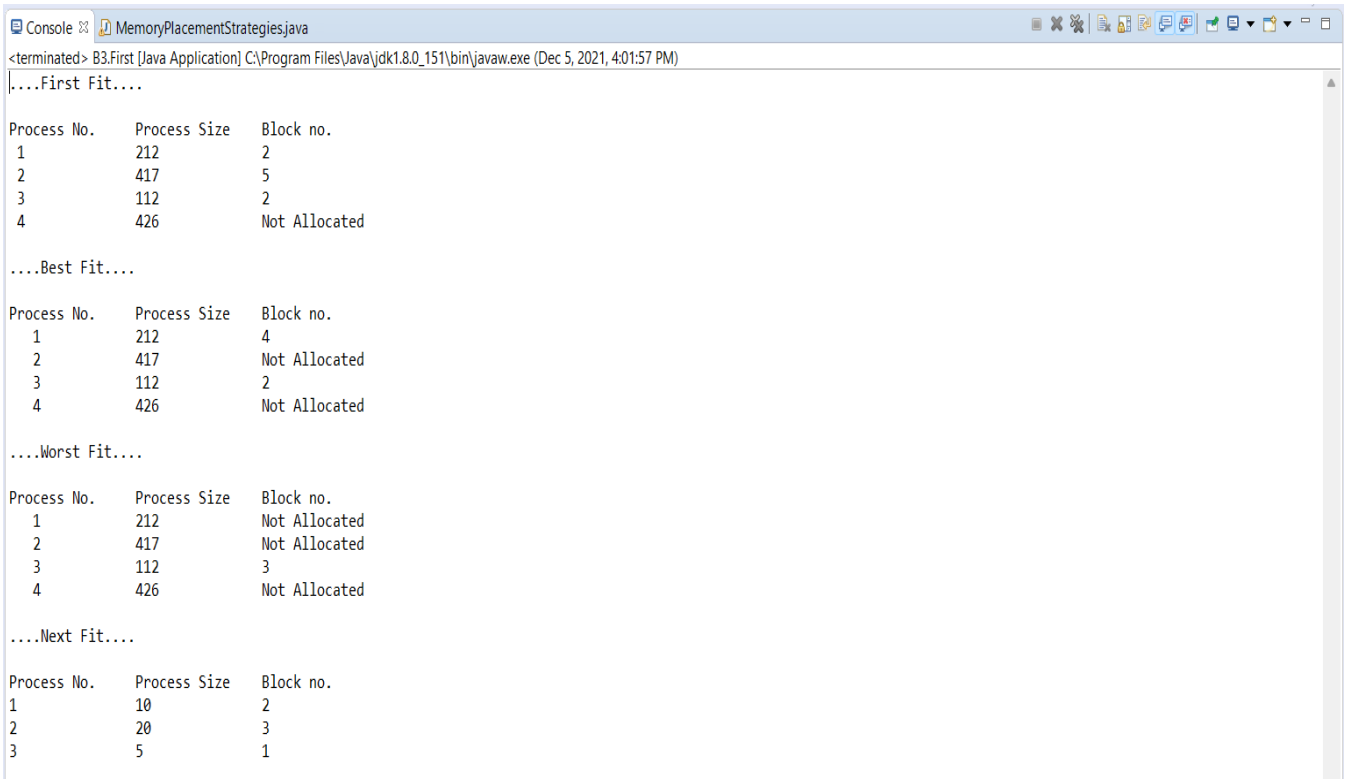
```
}

// If we could find a block for current process
if (bestIdx != -1)
{
// allocate block j to p[i] process
allocation[i] = bestIdx;

// Reduce available memory in this block.
blockSize[bestIdx] -= processSize[i];
}
}
System.out.println("\nProcess No.\tProcess Size\tBlock no.");
for (int i = 0; i < n; i++)
{
   System.out.print("   " + (i+1) + "\t\t" + processSize[i] + "\t\t");
   if (allocation[i] != -1)
      System.out.print(allocation[i] + 1);
   else
      System.out.print("Not Allocated");
   System.out.println();
}
}
   static void worstFit(int blockSize[], int m, int processSize[],
         int n)
{
// Stores block id of the block allocated to a
// process
int allocation[] = new int[n];

// Initially no block is assigned to any process
for (int i = 0; i < allocation.length; i++)
allocation[i] = -1;

// pick each process and find suitable blocks
// according to its size ad assign to it
for (int i=0; i<n; i++)
{
// Find the best fit block for current process
int wstIdx = -1;
for (int j=0; j<m; j++)
{
if (blockSize[j] >= processSize[i])
{
if (wstIdx == -1)
wstIdx = j;
```

```
else if (blockSize[wstIdx] < blockSize[j])
wstIdx = j;
}
}

// If we could find a block for current process
if (wstIdx != -1)
{
// allocate block j to p[i] process
allocation[i] = wstIdx;

// Reduce available memory in this block.
blockSize[wstIdx] -= processSize[i];
}
}

System.out.println("\nProcess No.\tProcess Size\tBlock no.");
for (int i = 0; i < n; i++)
{
System.out.print("   " + (i+1) + "\t\t" + processSize[i] + "\t\t");
if (allocation[i] != -1)
System.out.print(allocation[i] + 1);
else
System.out.print("Not Allocated");
System.out.println();
}
}

   static void NextFit(int blockSize1[], int m1, int processSize1[], int n1) {
      // Stores block id of the block allocated to a
      // process
      int allocation[] = new int[n1], j = 0;

      // Initially no block is assigned to any process
      Arrays.fill(allocation, -1);

      // pick each process and find suitable blocks
      // according to its size ad assign to it
      for (int i = 0; i < n1; i++) {

         // Do not start from beginning
         int count = 0;
         while (j < m1) {
            count++;    //makes sure that for every process we traverse through entire array
maximum once only.This avoids the problem of going into infinite loop if memory is not
available
```

```
            if (blockSize1[j] >= processSize1[i]) {

                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize1[j] -= processSize1[i];

                break;
            }

            // mod m will help in traversing the blocks from
            // starting block after we reach the end.
            j = (j + 1) % m1;
        }
    }

    System.out.print("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n1; i++) {
        System.out.print( i + 1 + "\t\t" + processSize1[i]
                + "\t\t");
        if (allocation[i] != -1) {
            System.out.print(allocation[i] + 1);
        } else {
            System.out.print("Not Allocated");
        }
        System.out.println("");
    }
}

// Driver Code
public static void main(String[] args)
{
        System.out.println("....First Fit....");
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = blockSize.length;
    int n = processSize.length;
    firstFit(blockSize, m, processSize, n);
    System.out.println(" ");
    System.out.println("....Best Fit....");
    bestFit(blockSize, m, processSize, n);
    System.out.println(" ");
    System.out.println("....Worst Fit....");
    worstFit(blockSize, m, processSize, n);
    System.out.println(" ");
```

```
        System.out.println("....Next Fit....");
        int blockSize1[] = {5, 10, 20};
        int processSize1[] = {10, 20, 5};
        int m1 = blockSize1.length;
        int n1 = processSize1.length;
        NextFit(blockSize1, m1, processSize1, n1);



    }
}
```

Output :


Console
MemoryPlacementStrategies.java
<terminated> B3.First [Java Application] C:\Program Files\Java\jdk1.8.0_151\bin\javaw.exe (Dec 5, 2021, 4:01:57 PM)
....First Fit....
Process No. Process Size Block no.
1 212 2
2 417 5
3 112 2
4 426 Not Allocated
....Best Fit....
Process No. Process Size Block no.
1 212 4
2 417 Not Allocated
3 112 2
4 426 Not Allocated
....Worst Fit....
Process No. Process Size Block no.
1 212 Not Allocated
2 417 Not Allocated
3 112 3
4 426 Not Allocated
....Next Fit....
Process No. Process Size Block no.
1 10 2
2 20 3
3 5 1