

SPOS 1

```
package pass1;
```

```
import java.util.ArrayList;
```

```
import java.util.HashMap;
```

```
import java.util.List;
```

```
import java.util.Map;
```

```
public class Pass1 {
```

```
    //MOT
```

```
    static Map <String, String> IS = new HashMap<String, String>();
```

```
    static Map <String, String> AD = new HashMap<String, String>();
```

```
    static Map <String, String> DL = new HashMap<String, String>();
```

```
    static Map <String, String> RG = new HashMap<String, String>();
```

```
    //To hold addresses of the symbols
```

```
    static List<String> symtab = new ArrayList<String>();
```

```
    static Map <String, Integer> symtabAddr = new HashMap<String, Integer>();
```

```
    //To hold addresses of the literals
```

```
    static List<String> littab = new ArrayList<String>();
```

```
    static Map <String, Integer> littabAddr = new HashMap<String, Integer>();
```

```
    //To hold the intermediate code
```

```
    static List<String> IC = new ArrayList<String>();
```

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

```
        //Initialize MOT
```

```
        IS.put("ADD", "01");
```

```
        IS.put("SUB", "02");
```

```
IS.put("MULT", "03");  
IS.put("MOVER", "04");  
IS.put("MOVEM", "05");  
IS.put("DIV", "06");
```

```
AD.put("START", "01");  
AD.put("END", "02");
```

```
DL.put("DS", "01");  
DL.put("DC", "02"); // changed DC code so it's different from DS  
RG.put("AREG", "01");  
RG.put("BREG", "02");  
RG.put("CREG", "03");  
RG.put("DREG", "04");
```

```
//Input program - to store in array
```

```
String program[] = {  
    "START 200",  
    "MOVER AREG, ONE",  
    "ADD BREG, TWO",  
    "ADD BREG, ='5'",  
    "MOVEM CREG, RESULT",  
    "ONE DS 4",  
    "TWO DC 2",  
    "RESULT DS 1",  
    "END"
```

```
};
```

```
pass_1(program);
```

```
//Print Tables
```

```
System.out.println("Symbol Table: ");
```

```

for (int i=0; i<symtab.size(); i++) {
    String sym = symtab.get(i);
    System.out.println(""+ (i) + " " + sym + " " + symtabAddr.get(sym));
}

System.out.println("\nLiteral Table: ");
for (int i=0; i<littab.size(); i++) {
    String lit = littab.get(i);
    System.out.println(""+ (i) + " " + lit + " " + littabAddr.get(lit));
}

System.out.println("\nIntermediate Code: ");
for (String line: IC) {
    System.out.println(line);
}
}

static void pass_1 (String [] program) {

    int lc=0;

    for (String line: program) {
        String parts[] = line.split("[,]+");

        if (AD.containsKey(parts[0])) {
            if (parts[0].equals("START")) {
                lc = Integer.parseInt(parts[1]);
                IC.add("AD " + AD.get("START") + " C " +lc);
            }
            else if (parts[0].equals("END")) {
                IC.add("AD " + AD.get("END"));
            }
        }
    }
}

```

```

        //Assign the address to literals
        for (int i=0; i<littab.size(); i++) {
            String lit = littab.get(i);
            littabAddr.put(lit, lc); //='5'
            String value = lit.substring(2, lit.length()-1);
            IC.add("DL "+ DL.get("DC") + " C " + value);
            lc++;
        }
    }
}

else if(IS.containsKey(parts[0])) {
    String ic = "IS " + IS.get(parts[0]);

    if (parts.length >1 && RG.containsKey(parts[1])) {
        ic += " RG " + RG.get(parts[1]);
    }

    if (parts.length>2) {
        if (parts[2].startsWith("=")) {
            //literal
            if (!littab.contains(parts[2]))
                littab.add(parts[2]);
            int idx = littab.indexOf(parts[2]);
            ic += " L "+ idx;
        }
        else {
            //symbol
            if (!symtab.contains(parts[2]))
                symtab.add(parts[2]);

```

```

        int idx = symtab.indexOf(parts[2]);
        ic += " S "+ idx;
    }
}
IC.add(ic);
lc++;
}
// LOOP ADD AREG Y
else {
    String label= parts[0];
    if (!symtab.contains(label))
        symtab.add(label);
    int idx= symtab.indexOf(label);

    //X DC 2
    if (DL.containsKey(parts[1])) {
        if (parts[1].equals("DS")) {
            symtabAddr.put(label, lc);
            IC.add("DL "+ DL.get("DS") + " C " + parts[2]);
            lc+= Integer.parseInt(parts[2]); //DS reserves N

wrods

        }
        else if (parts[1].equals("DC")) {
            symtabAddr.put(label, lc);
            IC.add("DL "+ DL.get("DC") + " C " + parts[2]);
            lc++;
        }
    }
}
}
}

```

```
}
```

CPP

```
#include <iostream>
#include <vector>
#include <sstream>
#include <algorithm>
using namespace std;

// Tables
vector<string> symtab;
vector<int> symaddr;
vector<string> littab;
vector<int> litaddr;
vector<string> IC;

// -----
// Function: Pass 1
// -----
void pass1(vector<string> program) {
    int lc = 0; // location counter

    for (string line : program) {
        if (line.empty()) continue;

        vector<string> parts;
        string word;
        stringstream ss(line);
        while (ss >> word) parts.push_back(word);

        string op = parts[0];

        // --- START ---
        if (op == "START") {
            lc = stoi(parts[1]);
            IC.push_back("AD 01 C " + to_string(lc));
        }

        // --- END ---
        else if (op == "END") {
            IC.push_back("AD 02");
            for (auto lit : littab) {
                litaddr.push_back(lc);
                string val = lit.substr(2, lit.size() - 3);
                IC.push_back("DL 02 C " + val);
                lc++;
            }
        }

        // --- Declarative (DS / DC) ---
        else if (op == "DS" || op == "DC") {
            string label = parts[0];
            symtab.push_back(label);
            symaddr.push_back(lc);

            if (op == "DS") {
```

```

        IC.push_back("DL 01 C " + parts[1]);
        lc += stoi(parts[1]);
    } else {
        IC.push_back("DL 02 C " + parts[1]);
        lc++;
    }
}

// --- Imperative (ADD, MOVER, etc.) ---
else {
    string ic = "IS ";

    // Assign opcode numbers manually (simple)
    if (op == "MOVER") ic += "04";
    else if (op == "MOVEM") ic += "05";
    else if (op == "ADD") ic += "01";
    else if (op == "SUB") ic += "02";
    else if (op == "MULT") ic += "03";
    else if (op == "DIV") ic += "06";

    // Register
    if (parts.size() > 1) {
        string reg = parts[1];
        if (reg == "AREG,") ic += " RG 01";
        else if (reg == "BREG,") ic += " RG 02";
        else if (reg == "CREG,") ic += " RG 03";
        else if (reg == "DREG,") ic += " RG 04";
    }

    // Operand (symbol or literal)
    if (parts.size() > 2) {
        string opr = parts[2];
        if (opr[0] == '=') { // literal
            if (find(littab.begin(), littab.end(), opr) ==
littab.end())

                littab.push_back(opr);
            int idx = find(littab.begin(), littab.end(), opr) -
littab.begin();
            ic += " L " + to_string(idx);
        } else { // symbol
            if (find(symtab.begin(), symtab.end(), opr) ==
symtab.end()) {
                symtab.push_back(opr);
                symaddr.push_back(0);
            }
            int idx = find(symtab.begin(), symtab.end(), opr) -
symtab.begin();
            ic += " S " + to_string(idx);
        }
    }

    IC.push_back(ic);
    lc++;
}
}

// -----
// MAIN
// -----
int main() {

```

```

vector<string> program = {
    "START 200",
    "MOVER AREG, ONE",
    "ADD BREG, TWO",
    "ADD BREG, ='5'",
    "MOVEM CREG, RESULT",
    "ONE DS 4",
    "TWO DC 2",
    "RESULT DS 1",
    "END"
};

pass1(program);

cout << "\n--- SYMBOL TABLE ---\n";
for (int i = 0; i < symtab.size(); i++)
    cout << i << " " << symtab[i] << " " << symaddr[i] << "\n";

cout << "\n--- LITERAL TABLE ---\n";
for (int i = 0; i < littab.size(); i++)
    cout << i << " " << littab[i] << " " << litaddr[i] << "\n";

cout << "\n--- INTERMEDIATE CODE ---\n";
for (auto &line : IC)
    cout << line << "\n";

return 0;
}

```

SPOS 2

```
package prac;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
public class SPOS2 {
```

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

```
        // Symbol table
```

```
        Map <Integer, ST> symtab = new HashMap<>();
```

```
        symtab.put(0, new ST (0, "X", 103));
```

```
        symtab.put(1, new ST (1, "Z", 104));
```

```

// Literal table

Map <Integer, LT> littab = new HashMap<>();

littab.put(0, new LT (0, "'5'", 110));

littab.put(0, new LT (1, "'2'", 111));


String IC [] = {

    "AD 01 c 100",

    "IS 04 RG 01 S 0",

    "IS 01 RG 01 S 0",

    "IS 05 RG 01 S 1",

    "DL 02 C 2",

    "DL 01 C 2",

    "AD 02",

    "DL 02 C 1"

};


System.out.println("Final Machine Code:- ");

System.out.println("=====");


for (String line: IC) {

    String tokens[] = line.split(" ,");

    String token_opcodeClass = tokens[0];

    String token_opcode = tokens.length>1? tokens[1] : null;

    String token_operandType1 = tokens.length>2? tokens[2] : null;

    String token_operand1 = tokens.length>3? tokens[3]:null;

    String token_operandType2 = tokens.length>4? tokens[4] : null;

    String token_operand2 = tokens.length > 5? tokens[5]:null;

    if ("IS".equals(token_opcodeClass)) {

        System.out.print(token_opcode + " ");
    }
}

```

```

        if ("RG".equals(token_operandType1)) {
            System.out.print(token_operand1 + " ");
        }

        if ("S".equals(token_operandType2)) {
            ST sym = symtab.get(Integer.parseInt(token_operand2));
            System.out.print(sym.addr);
        }

        if ("L".equals(token_operandType2)) {
            LT lit = littab.get(Integer.parseInt(token_operand2));
            System.out.print(lit.addr);
        }
        System.out.println();
    }

    else if ("DL".equals(token_opcodeClass)) {
        System.out.println("00 00 " + token_operand1);
    }

    else if ("AD".equals(token_opcodeClass)) {
        continue;
    }
}

}

}

ST

package prac;

public class LT {

```

```

    int index;
    String literal;
    int addr;

    public LT(int index, String literal, int addr) {
        super();
        this.index = index;
        this.literal = literal;
        this.addr = addr;
    }
}

```

LT

```

package prac;

public class LT {
    int index;
    String literal;
    int addr;

    public LT(int index, String literal, int addr) {
        super();
        this.index = index;
        this.literal = literal;
        this.addr = addr;
    }
}

```

SPOS 3

```
import java.util.*;
```

```

public class pass1 {

    static List<MNT>MNT = new ArrayList<> ();

    static List<String>MDT = new ArrayList<>();

    static Map <String , List<String>>ALA = new LinkedHashMap<>();

    public static void main(String args[])
    {
        String program [] =

            {

                "MACRO",

```

```
"INCR &A, &B",  
"ADD AREG, &A",  
"SUB BREG, &B",  
"MEND",  
"MACRO",  
"DECR &X, &Y",  
"SUB AREG, &X",  
"ADD BREG, &Y",  
"MEND",  
"START 100",  
"INCR A, B",  
"MOVER CREG, ='2'",  
"DECR 5, 10",  
"END"
```

```
};
```

```
pass1(program);
```

```
printMNT ();
```

```
printMDT ();
```

```
printALA ();
```

```
}
```

```
static void pass1(String[] program)
```

```
{
```

```
    boolean inMacroDef = false;
```

```
    String macroName = " ";
```

```
    int mntIndex =0;
```

```
    List<String> formals = new ArrayList<>();
```

```
    for(int lineNo=0; lineNo < program.length; lineNo++)
```

```
    {
```

```
        String line = program[lineNo].trim();
```

```

if(line.isEmpty()) continue;

String parts[] = line.split("[,]+" );

if(line.equals("MACRO"))
{
    inMacroDef=true;
    macroName = ""; //reset macro name and formals array
    formals.clear();
    continue;
}

if(inMacroDef)
{
    if(line.equals("MEND")) //for MEND
    {

        MDT.add("MEND");
        inMacroDef=false;
        continue;
    }

    if(macroName.equals("")) //for macro name
    {
        macroName = parts[0];
        int mdtStart=MDT.size();
        MNT.add(new MNT (mntIndex++,macroName, mdtStart ));
        for(int i=1; i<parts.length;i++)
        {
            formals.add(parts[i]);
        }
    }
}

```

```

else //for macro body
{
    StringBuilder mdtLine = new StringBuilder();
    for(String word : parts)
    {
        if(formals.contains(word))
        {
            int index = formals.indexOf(word);

mdtLine.append("#").append(index).append(" ");

        }
        else
        {
            mdtLine.append(word).append(" ");
        }
    }
    MDT.add(mdtLine.toString().trim());
}
}

else //for macro calls
{
    MNT found=null;
    for(MNT e : MNT)
    {
        if(e.name.equals(parts[0]))
        {
            found=e;
            break;
        }
    }
    if(found!=null)

```

```

        {
            List<String> actuals = new ArrayList<>();
            for(int i=1;i<parts.length;i++)
            {
                actuals.add(parts[i]);
            }
            ALA.put(parts[0], actuals);
        }
    }
}

static void printMNT()
{
    System.out.println("MNT ");
    System.out.println("Index\tName\tMDT Index");
    for(MNT e :MNT)
    {
        System.out.println(e.index + "\t" +e.name + "\t" + e.mdt_index);
    }
    System.out.println();
}

static void printMDT()
{
    System.out.println("MDT");
    System.out.println("Index\tDefinition");
    for(int i=0;i<MDT.size();i++)
    {
        System.out.println(i+"\t"+MDT.get(i));
    }
    System.out.println();
}

```

```

static void printALA()
{
    System.out.println("ALA");
    for(Map.Entry<String, List<String>> entry : ALA.entrySet())
    {
        System.out.println("Macro: "+entry.getKey());
        List<String> args = entry.getValue();
        for(int i=0; i<args.size(); i++)
        {
            System.out.println(" #" + i + "->" + args.get(i));
        }
    }
}

```

SPOS 4

```

package pass2;
import java.util.*;
public class pass2macro {

    static List<MNT>MNT=Arrays.asList
    (
        new MNT(1,"INCR",0),
        new MNT(2,"DECR",3)
    );

    static List<String>MDT=Arrays.asList
    (
        "ADD AREG,#0",
        "SUB BREG,#1",
        "MEND",

```

```

        "SUB AREG,#0",
        "ADD BREG,#1",
        "MEND"
    );

    static Map<String , List<String>> ALA=new HashMap<>();

    static
    {
        ALA.put("INCR",Arrays.asList("A","B"));
        ALA.put("DECR",Arrays.asList("X","Y"));

    }

    public static void main(String args[])
    {
        String[] program=
            {
                "START 100",
                "INCR A,B",
                "MOVER CREG,=2",
                "DECR X,Y",
                "END"
            };

        List<String>expanded=pass2(program);
        System.out.println("-----Expanded Code-----");
        for(String line:expanded)
        {
            System.out.println(line);
        }
    }

    static List<String> pass2(String[] program)

```

```

{
    List <String>output=new ArrayList<>();
    for(String line:program)
    {
        line=line.trim();
        if(line.isEmpty()) continue;
        String parts[]=line.split("[,]+" );
        MNT found=null;
        for(MNT e:MNT)
        {
            if(e.name.equals(parts[0]))
            {
                found=e;
                break;
            }
        }
        if(found!=null)
        {
            List<String> actuals=ALA.get(parts[0]);
            int index=found.mdtindex;
            while(index<MDT.size() &&
!MDT.get(index).equals("MEND"))
            {
                String expLine=MDT.get(index);
                for(int i=0;i<actuals.size();i++)
                {
                    expLine=expLine.replace("#"+i,actuals.get(i));
                }
                output.add(expLine);
                index++;
            }
        }
    }
}

```

```

        }
        else
        {
            output.add(line);
        }
    }
    return output;
}
}

```

MNT.java **package** pass2;

```

public class MNT {
    int index;
    String name;
    int mdtindex;
    public MNT(int index, String name, int mdtindex) {
        this.index = index;
        this.name = name;
        this.mdtindex = mdtindex;
    }
}

```

SPOS 5

FCFS

```
package scheduling_algo;
```

```
import java.util.Scanner;
```

```
public class fcfs {
```

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

```
    {
```

```
        Scanner sc= new Scanner(System.in);
```

```
        System.out.print("Enter the no of processes: ");
```

```
        int n= sc.nextInt();
```

```
        int [] processid = new int[n];
```

```
        int [] arrivalTime = new int[n];
```

```
        int [] burstTime = new int[n];
```

```
        int [] completionTime = new int[n];
```

```
        int [] turnaroundTime = new int[n];
```

```
        int [] waitingTime = new int[n];
```

```
        for(int i=0;i<n;i++)
```

```
        {
```

```
            processid[i]=i+1;
```

```
            System.out.print("Enter the arrival time for P"+processid[i]+" ");
```

```
            arrivalTime[i]=sc.nextInt();
```

```
            System.out.print("Enter the burst time for P"+processid[i]+" ");
```

```
            burstTime[i]=sc.nextInt();
```

```
        }
```

```
        for(int i=0;i<n-1;i++)
```

```
        {
```

```
            for(int j=i+1;j<n;j++)
```

```
            {
```

```
                if(arrivalTime[i]>arrivalTime[j])
```

```

        {

            int temp = arrivalTime[i];
            arrivalTime[i]=arrivalTime[j];
            arrivalTime[j]=temp;

            temp=burstTime[i];
            burstTime[i]=burstTime[j];
            burstTime[j]=temp;

            temp=processid[i];
            processid[i]=processid[j];
            processid[j]=temp;

        }
    }

}

int currentTime=0;
double totalWT=0, totalTAT=0;
for(int i=0;i<n;i++)
{
    if(currentTime<arrivalTime[i])
    {
        currentTime=arrivalTime[i];
    }
    completionTime[i] = currentTime + burstTime[i];
    currentTime=completionTime[i];
    turnaroundTime[i]=completionTime[i]-arrivalTime[i];
    waitingTime[i]=turnaroundTime[i]-burstTime[i];
    totalTAT+=turnaroundTime[i];
    totalWT+=waitingTime[i];
}

```

```

    }

    System.out.println("Process id\tArrival Time\tBurst Time\tCompletionTime\tTurn
Around Time\tWaiting Time");

    for(int i=0;i<n;i++)
    {
        System.out.println(processid[i]
+"\\t\\t"+arrivalTime[i]+"\\t\\t"+burstTime[i]+"\\t\\t"+completionTime[i]+"\\t\\t"
+"waitingTime[i]);
    }

    System.out.printf("Average TAT: %.2f\\n",totalTAT/n);
    System.out.printf("Average WT: %.2f\\n",totalWT/n);

    sc.close();
}

```

```

}

```

PRIORITY SCHEDULING

```

package scheduling_algo;

```

```

import java.util.Scanner;

```

```

public class priority {

    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the no of processes: ");

        int n = sc.nextInt();

        int[] processid = new int[n];
        int[] arrivalTime = new int[n];
        int[] burstTime = new int[n];
    }
}

```

```

int[] priority = new int[n];
int[] completionTime = new int[n];
int[] turnaroundTime = new int[n];
int[] waitingTime = new int[n];
boolean[] isCompleted=new boolean[n];

for(int i=0;i<n;i++)
{
    processid[i]=i+1;
    System.out.print("Enter the arrival time of P"+processid[i]+" : ");
    arrivalTime[i]=sc.nextInt();
    System.out.print("Enter the burst time of P"+processid[i]+" : ");
    burstTime[i]=sc.nextInt();
    System.out.print("Enter the priority of P"+processid[i]+" : ");
    priority[i]=sc.nextInt();
}

int completed=0;
int currentTime=0;
float totalTAT=0, totalWT=0;
while(completed!=n)
{
    int currentprocess = -1;
    int highest_priority = Integer.MAX_VALUE;

    for(int i=0;i<n;i++)
    {
        if(arrivalTime[i]<=currentTime && !isCompleted[i])
        {
            if(priority[i]<highest_priority)
            {

```

```

                                highest_priority=priority[i];
                                currentprocess=i;
                                }
                                }
                                }
                                if(currentprocess== -1)
                                {
                                    currentTime++;
                                    continue;
                                }
                                currentTime+=burstTime[currentprocess];
                                completionTime[currentprocess]=currentTime;
                                turnaroundTime[currentprocess]=completionTime[currentprocess]-
arrivalTime[currentprocess];
                                waitingTime[currentprocess]=turnaroundTime[currentprocess]-
burstTime[currentprocess];
                                totalTAT += turnaroundTime[currentprocess];
                                totalWT += waitingTime[currentprocess];
                                isCompleted[currentprocess]=true;
                                completed++;
                                }

                                System.out.println("ProcessID \tAT \tBT \tCT \tTAT \tWT");
                                for(int i=0;i<n;i++)
                                {

                                    System.out.println(processid[i]+"\\t\\t"+arrivalTime[i]+"\\t"+burstTime[i]+"\\t"+completionTime[i]+"\\t"+turnaroundTime[i]+"\\t"+waitingTime[i]);

                                }

                                System.out.println("Avg TAT: "+totalTAT/n);
                                System.out.println("Avg WT: "+totalWT/n);
                                sc.close();

```

```
}  
  
}
```

SJF

(Preemptive)

CODE:

package vee5;

```
import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;
```

```
public class SJF {  
    public static void main(String[] args) {  
  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter the total number of processes = ");  
        int n = sc.nextInt();  
  
        int[] pid = new int[n];  
        int[] at = new int[n];  
        int[] bt = new int[n];  
        int[] ct = new int[n];  
        int[] tat = new int[n];  
        int[] wt = new int[n];  
        boolean[] completed = new boolean[n];  
  
        for (int i = 0; i < n; i++)  
        {  
            pid[i] = i + 1;  
            System.out.println("Enter arrival time for " + pid[i] + " = ");  
            at[i] = sc.nextInt();  
            sc.nextLine();  
        }  
  
        for (int i = 0; i < n; i++)  
        {  
            System.out.println("Enter burst time for " + pid[i] + " = ");  
            bt[i] = sc.nextInt();  
            sc.nextLine();  
        }  
  
        // process with AT = 0  
        int firstProcessIndex = -1;  
        for (int i = 0; i < n; i++) {  
            if (at[i] == 0) {  
                firstProcessIndex = i;  
                break;  
            }  
        }  
  
        // process with AT = 0 first
```

```

int currentTime = 0;
currentTime += bt[firstProcessIndex];
ct[firstProcessIndex] = currentTime;
tat[firstProcessIndex] = ct[firstProcessIndex] - at[firstProcessIndex];
wt[firstProcessIndex] = tat[firstProcessIndex] - bt[firstProcessIndex];
completed[firstProcessIndex] = true;

// Track sums of TAT and WT
double sumTAT = tat[firstProcessIndex];
double sumWT = wt[firstProcessIndex];

// List to track the order in which processes are executed
List<Integer> executionOrder = new ArrayList<>();
executionOrder.add(pid[firstProcessIndex]);

// Now execute remaining processes based on the shortest burst time, considering arrival times.
int completedProcesses = 1;
while (completedProcesses < n) {
    int minBurstTime = Integer.MAX_VALUE;
    int minIndex = -1;

    // Look for the process with the shortest burst time that has arrived by current time
    for (int i = 0; i < n; i++) {
        if (!completed[i] && at[i] <= currentTime && bt[i] < minBurstTime)
        {
            minBurstTime = bt[i];
            minIndex = i;
        }
    }

    // If no process can be executed, increment the current time
    if (minIndex == -1)
        currentTime++;
    else
    {
        // Execute the selected process
        completed[minIndex] = true;
        currentTime += bt[minIndex];
        ct[minIndex] = currentTime;
        tat[minIndex] = ct[minIndex] - at[minIndex];
        wt[minIndex] = tat[minIndex] - bt[minIndex];

        sumTAT += tat[minIndex];
        sumWT += wt[minIndex];

        completedProcesses++;
        executionOrder.add(pid[minIndex]);
    }
}

System.out.println("\n\t\t\t...SJF...");
System.out.println("\tPID\tAT\tBT\tCT\tTAT\tWT");

for (int i = 0; i < n; i++) {
    int pidIndex = executionOrder.get(i) - 1;
    System.out.println("\tP" + pid[pidIndex] + "\t" + at[pidIndex] + "\t" + bt[pidIndex] + "\t"
        + ct[pidIndex] + "\t" + tat[pidIndex] + "\t" + wt[pidIndex]);
}

System.out.println("");

```

```

        System.out.println("TAT Avg = " + (sumTAT / n));
        System.out.println("WT Avg = " + (sumWT / n));

        sc.close();
    }
}

```

Round Robin (Preemptive)

CODE:

package vee5;

import java.util.*;

public class Round_Robin {

public static void main(String[] args) {
 Scanner sc = **new** Scanner(System.in);

 System.out.print("Enter total number of processes: ");
 int n = sc.nextInt();

int[] pid = **new int**[n];
 int[] at = **new int**[n];
 int[] bt = **new int**[n];
 int[] ct = **new int**[n];
 int[] tat = **new int**[n];
 int[] wt = **new int**[n];
 int[] rem_bt = **new int**[n];

for (**int** i = 0; i < n; i++) {
 pid[i] = i + 1;
 System.out.print("Enter arrival time of process " + pid[i] + ": ");
 at[i] = sc.nextInt();
 System.out.print("Enter burst time of process " + pid[i] + ": ");
 bt[i] = sc.nextInt();
 rem_bt[i] = bt[i];
 }

 System.out.print("Enter the time quantum: ");
 int quantum = sc.nextInt();

int completed = 0;
 int currentTime = 0;

 Queue<Integer> queue = **new** LinkedList<>();
 boolean[] inQueue = **new boolean**[n]; // To track which processes are in the queue

 // Add all processes that have arrived at time 0
 for (**int** i = 0; i < n; i++) {
 if (at[i] <= currentTime) {
 queue.add(i);
 inQueue[i] = **true**;
 }
 }

while (completed < n) {
 if (queue.isEmpty()) {

```

// No processes are ready; jump to next arrival time
int nextArrival = Integer.MAX_VALUE;
for (int i = 0; i < n; i++) {
    if (!inQueue[i] && rem_bt[i] > 0 && at[i] > currentTime) {
        nextArrival = Math.min(nextArrival, at[i]);
    }
}
currentTime = nextArrival;
for (int i = 0; i < n; i++) {
    if (!inQueue[i] && rem_bt[i] > 0 && at[i] <= currentTime) {
        queue.add(i);
        inQueue[i] = true;
    }
}
continue;
}

int i = queue.poll(); // get the process at the front

// Run for quantum or remaining burst time, whichever is smaller
int execTime = Math.min(quantum, rem_bt[i]);
rem_bt[i] -= execTime;
currentTime += execTime;

// Check if any other process arrived during execution time and add them to queue
for (int j = 0; j < n; j++) {
    if (!inQueue[j] && rem_bt[j] > 0 && at[j] <= currentTime) {
        queue.add(j);
        inQueue[j] = true;
    }
}

if (rem_bt[i] == 0) {
    completed++;
    ct[i] = currentTime;
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
} else {
    // Process not finished, add it back to the end of the queue
    queue.add(i);
}

}

double totalTat = 0, totalWt = 0;
for (int i = 0; i < n; i++) {
    totalTat += tat[i];
    totalWt += wt[i];
}

System.out.println("\nProcess\tAT\tBT\tCT\tTAT\tWT");
for (int i = 0; i < n; i++) {
    System.out.println(pid[i] + "\t" + at[i] + "\t" + bt[i] + "\t" + ct[i] + "\t" + tat[i] + "\t" + wt[i]);
}

System.out.println("\nAverage Turnaround Time (TAT): " + (totalTat / n));
System.out.println("Average Waiting Time (WT): " + (totalWt / n));


sc.close();
}
}

```

SPOS 6

```
#include <iostream>
#include <vector>
using namespace std;

// Utility: print result
void printAllocation(const vector<int>& blocks, const vector<int>&
processes, const vector<int>& allocation) {
    cout << "\nProcess No.\tProcess Size\tBlocks No.\n";
    for (size_t i = 0; i < processes.size(); i++) {
        cout << " " << i + 1 << "\t\t" << processes[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << "\n";
    }
}

/*-----
1  FIRST FIT (Fixed Partitioning)
-----*/
void firstFit(const vector<int>& blocks, const vector<int>& processes) {
    vector<int> allocation(processes.size(), -1);
    vector<bool> occupied(blocks.size(), false);

    for (size_t i = 0; i < processes.size(); i++) {
        for (size_t j = 0; j < blocks.size(); j++) {
            if (!occupied[j] && blocks[j] >= processes[i]) {
                allocation[i] = j;
                occupied[j] = true; // mark blocks as used
                break;
            }
        }
    }

    cout << "\n--- FIRST FIT (Fixed Partitioning) ---";
    printAllocation(blocks, processes, allocation);
}


/*-----
2  BEST FIT (Fixed Partitioning)
-----*/
void bestFit(const vector<int>& blocks, const vector<int>& processes) {
    vector<int> allocation(processes.size(), -1);
    vector<bool> occupied(blocks.size(), false);

    for (size_t i = 0; i < processes.size(); i++) {
        int bestIdx = -1;
        for (size_t j = 0; j < blocks.size(); j++) {
            if (!occupied[j] && blocks[j] >= processes[i]) {
                if (bestIdx == -1 || blocks[j] < blocks[bestIdx])
                    bestIdx = j;
            }
        }
    }
}
```

```


    }
}
if (bestIdx != -1) {
    allocation[i] = bestIdx;
    occupied[bestIdx] = true;
}
}

cout << "\n--- BEST FIT (Fixed Partitioning) ---";
printAllocation(blocks, processes, allocation);
}

/*-----
3  WORST FIT (Fixed Partitioning)
-----*/
void worstFit(const vector<int>& blocks, const vector<int>& processes) {
    vector<int> allocation(processes.size(), -1);
    vector<bool> occupied(blocks.size(), false);

    for (size_t i = 0; i < processes.size(); i++) {
        int worstIdx = -1;
        for (size_t j = 0; j < blocks.size(); j++) {
            if (!occupied[j] && blocks[j] >= processes[i]) {
                if (worstIdx == -1 || blocks[j] > blocks[worstIdx])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            occupied[worstIdx] = true;
        }
    }

    cout << "\n--- WORST FIT (Fixed Partitioning) ---";
    printAllocation(blocks, processes, allocation);
}

/*-----
4  NEXT FIT (Fixed Partitioning)
-----*/
void nextFit(const vector<int>& blocks, const vector<int>& processes) {
    vector<int> allocation(processes.size(), -1);
    vector<bool> occupied(blocks.size(), false);
    size_t j = 0; // start search index

    for (size_t i = 0; i < processes.size(); i++) {
        size_t count = 0;
        while (count < blocks.size()) {
            if (!occupied[j] && blocks[j] >= processes[i]) {
                allocation[i] = j;
                occupied[j] = true;
                break;
            }
            j = (j + 1) % blocks.size();
            count++;
        }
    }

    cout << "\n--- NEXT FIT (Fixed Partitioning) ---";
    printAllocation(blocks, processes, allocation);
}

```

```

/*-----
  MAIN PROGRAM
-----*/
int main() {
    vector<int> blocks = {100, 500, 200, 300, 600};
    vector<int> processes = {212, 417, 112, 426};

    cout << "Number of memory blockss: " << 5 << endl;
    cout << "blocks sizes: ";
    for (int b : blocks) cout << b << " ";
    cout << "\n\nNumber of processeses: " << 4 << endl;
    cout << "processes sizes: ";
    for (int p : processes) cout << p << " ";
    cout << "\n";

    firstFit(blocks, processes);
    bestFit(blocks, processes);
    worstFit(blocks, processes);
    nextFit(blocks, processes);

    return 0;
}

```