Practical No.1

```
/*
Problem Statement 1: Two-Pass Assembler
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.

SOURCE.ASM COPY THIS AND SAVE FILE TO Source.asm


START 100
LOOP LDA ALPHA
STA BETA
ADD GAMMA
ALPHA DC 5
BETA DS 1
GAMMA DC 10
END

*/
import java.io.*;
import java.util.*;

class Symbol {
    String name;
    int address;
    boolean isDefined;

    Symbol(String name, int address, boolean defined) {
        this.name = name;
        this.address = address;
        this.isDefined = defined;
    }
}

public class TwoPassAssembler {

    private static final Map<String, String[]> OP_TABLE = new HashMap<>();
    static {
        OP_TABLE.put("LDA", new String[]{"IS", "01"});
        OP_TABLE.put("STA", new String[]{"IS", "02"});
        OP_TABLE.put("ADD", new String[]{"IS", "03"});
        OP_TABLE.put("SUB", new String[]{"IS", "04"});
        OP_TABLE.put("MOV", new String[]{"IS", "07"});
        OP_TABLE.put("START", new String[]{"AD", "01"});
        OP_TABLE.put("END", new String[]{"AD", "02"});
        OP_TABLE.put("DC", new String[]{"DL", "01"});
        OP_TABLE.put("DS", new String[]{"DL", "02"});
    }

    private static final Map<String, Symbol> SYMBOL_TABLE = new
LinkedHashMap<>();
    private static List<String> INTERMEDIATE = new ArrayList<>();
    private static int LOC_COUNTER = 0;

    private static int getOrCreateSymbolIndex(String name) {
        if (!SYMBOL_TABLE.containsKey(name)) {
            SYMBOL_TABLE.put(name, new Symbol(name, -1, false));
        }
        int idx = 1;
        for (String key : SYMBOL_TABLE.keySet()) {
            if (key.equals(name)) break;
```

```
            idx++;
        }
        return idx;
    }

    public static void pass1(String sourceFile, String interFile, String
symFile) throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader(sourceFile));
             BufferedWriter interWriter = new BufferedWriter(new
FileWriter(interFile));
             BufferedWriter symWriter = new BufferedWriter(new
FileWriter(symFile))) {

            String line;
            LOC_COUNTER = 0;

            while ((line = br.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty() || line.startsWith(";")) continue;

                String[] tokens = line.split("\\s+");
                String label = null, opcode = null, operand = null;

                int i = 0;

                if (tokens.length > 0 && tokens[0].endsWith(":")) {
                    label = tokens[0].substring(0, tokens[0].length() - 1);
                    i = 1;
                } else if (tokens.length > 1 && !
OP_TABLE.containsKey(tokens[0])) {
                    label = tokens[0];
                    i = 1;
                }

                if (i < tokens.length) opcode = tokens[i++];
                if (i < tokens.length) operand = tokens[i];


                if (label != null && !label.isEmpty()) {
                    if (SYMBOL_TABLE.containsKey(label)) {
                        Symbol existingSym = SYMBOL_TABLE.get(label);
                        if (existingSym.isDefined) {
                            System.err.println("Error: Duplicate definition of
symbol '" + label + "' at LC " + LOC_COUNTER);
                        } else {
                            existingSym.address = LOC_COUNTER;
                            existingSym.isDefined = true;
                        }
                    } else {
                        SYMBOL_TABLE.put(label, new Symbol(label, LOC_COUNTER,
true));
                    }
                }


                if (opcode == null || !OP_TABLE.containsKey(opcode)) {
                    System.err.println("Error: Invalid/missing opcode in line: "
+ line);
                    continue;
                }

                String[] opInfo = OP_TABLE.get(opcode);
                String cls = opInfo[0];
                String opc = opInfo[1];
```

```
                switch (cls) {
                    case "AD":
                        if ("START".equals(opcode)) {
                            LOC_COUNTER = (operand != null) ?
 Integer.parseInt(operand) : 0;
                            INTERMEDIATE.add(String.format("%03d (AD,01) (C,
%s)", LOC_COUNTER, operand));
                        } else if ("END".equals(opcode)) {
                            INTERMEDIATE.add("(AD,02)");
                        }
                        break;

                    case "IS":
                        String operandCode = "-";
                        if (operand != null) {
                            if (operand.matches("\\d+")) {
                                operandCode = "(C," + operand + ")";
                            } else {
                                int symIdx = getOrCreateSymbolIndex(operand);
                                operandCode = "(S," + symIdx + ")";
                            }
                        }
                        INTERMEDIATE.add(String.format("%03d (IS,%s) %s",
LOC_COUNTER, opc, operandCode));
                        LOC_COUNTER++;
                        break;

                    case "DL":
                        if ("DC".equals(opcode)) {
                            INTERMEDIATE.add(String.format("%03d (DL,01) (C,
%s)", LOC_COUNTER, operand));
                            LOC_COUNTER++;
                        } else if ("DS".equals(opcode)) {
                            int size = Integer.parseInt(operand);
                            INTERMEDIATE.add(String.format("%03d (DL,02) (C,
%s)", LOC_COUNTER, operand));
                            LOC_COUNTER += size;
                        }
                        break;
                }
            }

            for (String ic : INTERMEDIATE) {
                interWriter.write(ic);
                interWriter.newLine();
            }

            symWriter.write("Index\tSymbol\tAddress\tDefined\n");
            int idx = 1;
            for (Symbol sym : SYMBOL_TABLE.values()) {
                symWriter.write(String.format("%d\t%s\t%d\t%s\n",
                    idx++, sym.name, sym.address, sym.isDefined ? "YES" :
"NO"));
            }
        }
    }

    public static void pass2(String interFile, String symFile, String
machineFile) throws IOException {
        Map<Integer, Integer> symAddr = new HashMap<>();
        try (BufferedReader symReader = new BufferedReader(new
FileReader(symFile))) {
            symReader.readLine();
```

```
            String line;
            int idx = 1;
            while ((line = symReader.readLine()) != null) {
                String[] parts = line.trim().split("\\s+");
                if (parts.length >= 3) {
                    int addr = Integer.parseInt(parts[2]);
                    symAddr.put(idx++, addr);
                }
            }
        }

        try (BufferedReader interReader = new BufferedReader(new
FileReader(interFile));
             BufferedWriter machineWriter = new BufferedWriter(new
FileWriter(machineFile))) {

            String line;
            while ((line = interReader.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty() || (line.contains("(AD,") && !
line.contains("START"))) {
                    continue;
                }

                String[] parts = line.split("\\s+", 3);
                if (parts.length < 2) continue;

                String locStr = parts[0];
                String opPart = parts[1];
                String oprPart = (parts.length > 2) ? parts[2] : "-";

                String content = opPart.substring(1, opPart.length() - 1);
                String[] opInfo = content.split(",");
                String opClass = opInfo[0];
                int opcode = Integer.parseInt(opInfo[1]);

                if ("DL".equals(opClass)) {
                    if (opcode == 1) {
                        int constantValue = Integer.parseInt(
                            oprPart.substring(oprPart.indexOf(',') + 1,
oprPart.length() - 1));

                        machineWriter.write(String.format("%s %03d", locStr,
constantValue));
                        machineWriter.newLine();
                    }
                    continue;
                }

                int operandValue = 0;
                if (oprPart.equals("-")) {
                    operandValue = 0;
                } else if (oprPart.startsWith("(C,")) {
                    operandValue = Integer.parseInt(
                        oprPart.substring(oprPart.indexOf(',') + 1,
oprPart.length() - 1));
                } else if (oprPart.startsWith("(S,")) {
                    int symIdx = Integer.parseInt(
                        oprPart.substring(oprPart.indexOf(',') + 1,
oprPart.length() - 1));
                    operandValue = symAddr.getOrDefault(symIdx, 0);
                    if (operandValue == 0 && symIdx != 0) {
                        System.err.println("Warning: Undefined symbol address
retrieved for index " + symIdx);
```

```
                    }
                }

                machineWriter.write(String.format("%s %02d %03d", locStr,
opcode, operandValue));
                machineWriter.newLine();
            }
        }
    }

    public static void main(String[] args) {
        String source = "source.asm";
        String inter = "intermediate.ic";
        String symtab = "symtab.txt";
        String machine = "machinecode.mc";

        if (args.length >= 4) {
            source = args[0];
            inter = args[1];
            symtab = args[2];
            machine = args[3];
        }

        try {
            SYMBOL_TABLE.clear();
            INTERMEDIATE.clear();

            pass1(source, inter, symtab);
            pass2(inter, symtab, machine);

            System.out.println("Two-pass assembly completed successfully.");
            System.out.println("Intermediate: " + inter);
            System.out.println("Symbol Table: " + symtab);
            System.out.println("Machine Code: " + machine);

        } catch (Exception e) {
            System.err.println("Assembly Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Practical No.2

```
/*
Problem Statement 2: Two-Pass Macro Processor
Design suitable data structures and implement Pass-I and Pass-II of a two-pass
macroprocessor.
The output of Pass-I (MNT, MDT and intermediate code file without any macro
definitions) should be input for Pass-II.


SOURCE.ASM COPY THIS AND SAVE FILE TO Source.asm


START 100
LOOP LDA ALPHA
STA BETA
ADD GAMMA
ALPHA DC 5
BETA DS 1
GAMMA DC 10
END

*/
```

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

class MNTEntry {
    String macroName;
    int mdtIndex;

    MNTEntry(String macroName, int mdtIndex) {
        this.macroName = macroName;
        this.mdtIndex = mdtIndex;
    }
}

public class TwoPassMacroProcessor {
    static List<MNTEntry> MNT = new ArrayList<>();
    static List<String> MDT = new ArrayList<>();
    static List<String> intermediateCode = new ArrayList<>();

    static void pass1(String sourceFile, String interFile, String mntFile,
String mdtFile) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(sourceFile));
        BufferedWriter interWriter = new BufferedWriter(new
FileWriter(interFile));
        BufferedWriter mntWriter = new BufferedWriter(new FileWriter(mntFile));
        BufferedWriter mdtWriter = new BufferedWriter(new FileWriter(mdtFile));

        String line;
        boolean inMacro = false;
        String currentMacro = null;
        int mdtIndex = 0;

        while ((line = br.readLine()) != null) {
            String trimmedLine = line.trim();
            if (trimmedLine.isEmpty()) continue;

            String[] parts = trimmedLine.split("\\s+");
            String firstWord = parts[0];

            if (firstWord.equalsIgnoreCase("MACRO")) {
                inMacro = true;
                continue; // Skip MACRO line
            }

            if (inMacro) {
                if (currentMacro == null) {

                    currentMacro = parts[0];


                    MNT.add(new MNTEntry(currentMacro, MDT.size()));


                    MDT.add(trimmedLine);

                } else if (firstWord.equalsIgnoreCase("MEND")) {

                    MDT.add("MEND");
                    inMacro = false;
                    currentMacro = null;
                } else {

                    MDT.add(trimmedLine);
                }
```

```
            } else {

                interWriter.write(trimmedLine);
                interWriter.newLine();
            }
        }


        for (MNTEntry entry : MNT) {
            mntWriter.write(entry.macroName + "\t" + entry.mdtIndex);
            mntWriter.newLine();
        }

        for (String mdtLine : MDT) {
            mdtWriter.write(mdtLine);
            mdtWriter.newLine();
        }

        br.close();
        interWriter.close();
        mntWriter.close();
        mdtWriter.close();
    }


    static void pass2(String interFile, String mntFile, String mdtFile, String
outputFile) throws IOException {
        BufferedReader interReader = new BufferedReader(new
FileReader(interFile));
        BufferedReader mntReader = new BufferedReader(new FileReader(mntFile));
        BufferedReader mdtReader = new BufferedReader(new FileReader(mdtFile));
        BufferedWriter outputWriter = new BufferedWriter(new
FileWriter(outputFile));


        Map<String, Integer> mntMap = new HashMap<>();
        String mntLine;
        while ((mntLine = mntReader.readLine()) != null) {
            String[] parts = mntLine.trim().split("\\s+");
            mntMap.put(parts[0], Integer.parseInt(parts[1]));
        }


        List<String> mdtList = new ArrayList<>();
        String mdtLine;
        while ((mdtLine = mdtReader.readLine()) != null) {
            mdtList.add(mdtLine);
        }

        String line;
        while ((line = interReader.readLine()) != null) {
            String trimmedLine = line.trim();
            if (trimmedLine.isEmpty()) {
                outputWriter.newLine();
                continue;
            }
            String[] parts = trimmedLine.split("\\s+");
            String possibleMacro = parts[0];

            if (mntMap.containsKey(possibleMacro)) {

                int mdtIndex = mntMap.get(possibleMacro);

                while (!mdtList.get(mdtIndex).equalsIgnoreCase("MEND")) {
```

```
                    outputWriter.write(mdtList.get(mdtIndex));
                    outputWriter.newLine();
                    mdtIndex++;
                }
            } else {

                outputWriter.write(trimmedLine);
                outputWriter.newLine();
            }
        }

        interReader.close();
        mntReader.close();
        mdtReader.close();
        outputWriter.close();
    }

    public static void main(String[] args) {
        try {
            pass1("source.asm", "intermediate.asm", "mnt.txt", "mdt.txt");
            pass2("intermediate.asm", "mnt.txt", "mdt.txt", "expanded.asm");

            System.out.println("Two-pass Macro processing completed.");
            System.out.println("Intermediate code: intermediate.asm");
            System.out.println("Macro Name Table: mnt.txt");
            System.out.println("Macro Definition Table: mdt.txt");
            System.out.println("Expanded code after macro expansion:
expanded.asm");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Practical No.3

```
/*
Problem Statement 3: First Come First Serve (FCFS) CPU Scheduling
Write a program to simulate the First Come First Serve (FCFS) CPU scheduling
algorithm.
The program should accept process details such as Process ID, Arrival Time, and
Burst Time and compute the Waiting Time and Turnaround Time for each process.
Display the Gantt chart, average waiting time, and average turnaround time.

Description:
This program simulates the FCFS CPU scheduling algorithm. It takes input for the
number of processes and their respective burst times. Although the problem
statement mentions arrival time, this implementation assumes all processes
arrive at time zero (or in order) since arrival time handling is not fully
included here. The program calculates waiting time and turnaround time for each
process and prints a Gantt chart along with average waiting time and turnaround
time.
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSES 10
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
};
```

```
void initialize_processes(struct Process *procs, int *num_processes_ptr);
int compare_arrival_time(const void *a, const void *b);
void calculate_fcfs(struct Process *procs, int num_processes);
void display_gantt_chart(struct Process *procs, int num_processes);
void display_results(struct Process *procs, int num_processes);

int main() {
    struct Process processes[MAX_PROCESSES];
    int num_processes = 0;

    initialize_processes(processes, &num_processes);
    qsort(processes, num_processes, sizeof(struct Process),
compare_arrival_time);

    calculate_fcfs(processes, num_processes);
    display_gantt_chart(processes, num_processes);
    display_results(processes, num_processes);

    return 0;
}

void initialize_processes(struct Process *procs, int *num_processes_ptr) {
    printf("=== First-Come, First-Served (FCFS) Scheduling ===\n\n");
    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", num_processes_ptr);

    if (*num_processes_ptr <= 0 || *num_processes_ptr > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        exit(1);
    }

    printf("\n");
    for (int i = 0; i < *num_processes_ptr; i++) {
        struct Process *current_proc = procs + i;
        current_proc->id = i + 1;
        printf("Enter arrival time for Process P%d: ", current_proc->id);
        scanf("%d", &current_proc->arrival_time);
        printf("Enter burst time for Process P%d: ", current_proc->id);
        scanf("%d", &current_proc->burst_time);
    }
}

int compare_arrival_time(const void *a, const void *b) {
    struct Process *p1 = (struct Process *)a;
    struct Process *p2 = (struct Process *)b;
    return p1->arrival_time - p2->arrival_time;
}

void calculate_fcfs(struct Process *procs, int num_processes) {
    int current_time = 0;

    for (int i = 0; i < num_processes; i++) {
        struct Process *p = procs + i;

        if (current_time < p->arrival_time) {
            current_time = p->arrival_time;
        }

        p->waiting_time = current_time - p->arrival_time;
        p->completion_time = current_time + p->burst_time;
        p->turnaround_time = p->completion_time - p->arrival_time;

        current_time = p->completion_time;
```

```
    }
}

void display_gantt_chart(struct Process *procs, int num_processes) {
    printf("\n--- Gantt Chart ---\n");
    int last_completion_time = 0;
    struct Process *p;

    printf("|");
    for (int i = 0; i < num_processes; i++) {
        p = procs + i;
        if (p->arrival_time > last_completion_time) {
            printf(" IDLE |");
        }
        printf("  P%d  |", p->id);
        last_completion_time = p->completion_time;
    }
    printf("\n");

    last_completion_time = 0;
    printf("0");
    for (int i = 0; i < num_processes; i++) {
        p = procs + i;
        if (p->arrival_time > last_completion_time) {
            printf("     %d", p->arrival_time);
        }
        printf("      %d", p->completion_time);
        last_completion_time = p->completion_time;
    }
    printf("\n");
}

void display_results(struct Process *procs, int num_processes) {
    printf("\n--- FCFS Scheduling Results ---\n");
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");

printf("-------------------------------------------------------------------
\n");

    float total_wt = 0, total_tat = 0;

    for (int i = 0; i < num_processes; i++) {
        struct Process *p = procs + i;
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n",
               p->id,
               p->arrival_time,
               p->burst_time,
               p->waiting_time,
               p->turnaround_time);

        total_wt += p->waiting_time;
        total_tat += p->turnaround_time;
    }

printf("-------------------------------------------------------------------
\n");
    printf("Average Waiting Time: %.2f\n", total_wt / num_processes);
    printf("Average Turnaround Time: %.2f\n", total_tat / num_processes);
}

/*
Problem Statement 4: Shortest Job First (SJF â Preemptive) Scheduling
Write a program to simulate the Shortest Job First (SJF â Preemptive) CPU
```

Practical No.4

scheduling algorithm. The program should calculate and display the order of execution, waiting time, turnaround time, and their averages for all processes.
Description:
This program simulates the SJF Preemptive scheduling algorithm where the CPU is assigned to the process with the shortest remaining burst time at every time unit. Processes with earlier arrival times and smaller burst times are prioritized. The program tracks remaining time for each process, calculates waiting and turnaround times after completion, and prints the results including average waiting and turnaround times.
*/

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSES 10

struct Process {
    int id;
    int burst_time;
    int arrival_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completed;
};

struct Process processes[MAX_PROCESSES];
int num_processes = 0;

void initialize_processes() {
    printf("=== Shortest Job First (Preemptive) Scheduling ===\n\n");
    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &num_processes);

    for (int i = 0; i < num_processes; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time for Process P%d: ", processes[i].id);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for Process P%d: ", processes[i].id);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].completed = 0;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }
}

void calculate_sjf() {
    int current_time = 0;
    int completed = 0;
    int shortest = -1;
    int min_burst = 9999;
    int check = 0;

    while (completed != num_processes) {
        shortest = -1;
        min_burst = 9999;
        check = 0;

        for (int i = 0; i < num_processes; i++) {
            if (processes[i].arrival_time <= current_time &&
                processes[i].remaining_time > 0 &&
                processes[i].remaining_time < min_burst)
            {
                min_burst = processes[i].remaining_time;
```

```
                shortest = i;
                check = 1;
            }
        }

        if (check == 0) {
            current_time++;
            continue;
        }

        processes[shortest].remaining_time--;
        current_time++;

        if (processes[shortest].remaining_time == 0) {
            completed++;
            processes[shortest].turnaround_time = current_time -
processes[shortest].arrival_time;
            processes[shortest].waiting_time =
processes[shortest].turnaround_time - processes[shortest].burst_time;
            if (processes[shortest].waiting_time < 0) {
                processes[shortest].waiting_time = 0;
            }
        }
    }
}

void display_results() {
    printf("\n--- SJF CPU Scheduling Algorithm ---\n");
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");
    printf("-----------------------------------------------------------------
\n");

    float total_wt = 0, total_tat = 0;

    for (int i = 0; i < num_processes; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].id,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].waiting_time,
               processes[i].turnaround_time);

        total_wt += processes[i].waiting_time;
        total_tat += processes[i].turnaround_time;
    }

    printf("-----------------------------------------------------------------
\n");
    printf("Average Waiting Time: %.2f\n", total_wt / num_processes);
    printf("Average Turnaround Time: %.2f\n", total_tat / num_processes);
}

int main() {
    initialize_processes();
    calculate_sjf();
    display_results();
    return 0;
}


/*
Problem Statement 5: Priority Scheduling (Non-Preemptive)
Write a program to simulate the Priority Scheduling (Non-Preemptive) algorithm.
```

Practical No.5

```
Each process should have an associated priority value, and the scheduler should
select the process with the highest priority for execution next.
Compute and display the waiting time, turnaround time, and average times for all
processes.
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSES 10
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
    int completed;
};

struct Process processes[MAX_PROCESSES];
int num_processes = 0;

void initialize_processes() {
    printf("=== Priority Scheduling (Non-Preemptive) ===\n\n");
    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &num_processes);

    if (num_processes <= 0 || num_processes > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        exit(1);
    }

    for (int i = 0; i < num_processes; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time for Process P%d: ", processes[i].id);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for Process P%d: ", processes[i].id);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for Process P%d (lower value = higher priority):
", processes[i].id);
        scanf("%d", &processes[i].priority);
        processes[i].completed = 0;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
    }
}

void calculate_priority() {
    int current_time = 0;
    int completed_processes = 0;

    while (completed_processes < num_processes) {
        int highest_priority = 9999;
        int idx = -1;
        for (int i = 0; i < num_processes; i++) {
            if (processes[i].arrival_time <= current_time &&
                !processes[i].completed &&
                processes[i].priority < highest_priority) {
                highest_priority = processes[i].priority;
                idx = i;
            }
        }
        if (idx == -1) {
            current_time++;
```

```
            continue;
        }

        current_time += processes[idx].burst_time;
        processes[idx].waiting_time = current_time - processes[idx].arrival_time
- processes[idx].burst_time;
        if (processes[idx].waiting_time < 0) processes[idx].waiting_time = 0;
        processes[idx].turnaround_time = processes[idx].waiting_time +
processes[idx].burst_time;
        processes[idx].completed = 1;
        completed_processes++;
    }
}

void display_results() {
    printf("\n--- Priority Scheduling (Non-Preemptive) Results ---\n");
    printf("Process\tArrival Time\tBurst Time\tPriority\tWaiting
Time\tTurnaround Time\n");

printf("------------------------------------------------------------------------
-----\n");

    float total_wt = 0, total_tat = 0;
    for (int i = 0; i < num_processes; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].id,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].priority,
               processes[i].waiting_time,
               processes[i].turnaround_time);
        total_wt += processes[i].waiting_time;
        total_tat += processes[i].turnaround_time;
    }

printf("------------------------------------------------------------------------
-----\n");
    printf("Average Waiting Time: %.2f\n", total_wt / num_processes);
    printf("Average Turnaround Time: %.2f\n", total_tat / num_processes);
}

int main() {
    initialize_processes();
    calculate_priority();
    display_results();
    return 0;
}
```

Practical No.6

```
Problem Statement 6: Round Robin (RR) Scheduling
Write a program to simulate the Round Robin (Preemptive) CPU scheduling
algorithm.
The program should take time quantum as input and schedule processes in a cyclic
order.
Display the Gantt chart, waiting time, turnaround time, and average values for
all processes.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_PROCESSES 10
#define MAX_GANTT_SLOTS (MAX_PROCESSES * 20)
#define QUEUE_SIZE (MAX_PROCESSES * 10)
```

```
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
};

struct GanttSegment {
    int process_id;
    int start_time;
    int end_time;
};

void initialize_processes(struct Process *procs, int *num_processes_ptr, int
*quantum_ptr);
int compare_arrival_time(const void *a, const void *b);
static void enqueue_new_arrivals(int current_time, struct Process *procs, int
num_processes,
                                 int *queue, int *rear, int *is_in_system, int
*processes_arrived);
void calculate_rr(struct Process *procs, int num_processes, int time_quantum,
                  struct GanttSegment *gantt, int *gantt_idx_ptr);
void display_gantt_chart(struct GanttSegment *gantt, int gantt_idx);
void display_results(struct Process *procs, int num_processes);

int main() {
    struct Process processes[MAX_PROCESSES];
    struct GanttSegment gantt_chart[MAX_GANTT_SLOTS];
    int num_processes = 0;
    int time_quantum = 0;
    int gantt_index = 0;

    initialize_processes(processes, &num_processes, &time_quantum);
    qsort(processes, num_processes, sizeof(struct Process),
compare_arrival_time);
    calculate_rr(processes, num_processes, time_quantum, gantt_chart,
&gantt_index);
    display_gantt_chart(gantt_chart, gantt_index);
    display_results(processes, num_processes);

    return 0;
}

void initialize_processes(struct Process *procs, int *num_processes_ptr, int
*quantum_ptr) {
    printf("=== Round Robin Scheduling ===\n\n");
    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", num_processes_ptr);

    if (*num_processes_ptr <= 0 || *num_processes_ptr > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        exit(1);
    }

    for (int i = 0; i < *num_processes_ptr; i++) {
        struct Process *p = &procs[i];
        p->id = i + 1;
        printf("Enter arrival time for Process P%d: ", p->id);
        scanf("%d", &p->arrival_time);
        printf("Enter burst time for Process P%d: ", p->id);
```

```
        scanf("%d", &p->burst_time);

        p->remaining_time = p->burst_time;
        p->waiting_time = 0;
        p->turnaround_time = 0;
        p->completion_time = 0;
    }

    printf("Enter Time Quantum: ");
    scanf("%d", quantum_ptr);
}

int compare_arrival_time(const void *a, const void *b) {
    const struct Process *p1 = (const struct Process *)a;
    const struct Process *p2 = (const struct Process *)b;
    if (p1->arrival_time != p2->arrival_time) {
        return p1->arrival_time - p2->arrival_time;
    }
    return p1->id - p2->id;
}

static void enqueue_new_arrivals(int current_time, struct Process *procs, int
num_processes,
                                 int *queue, int *rear, int *is_in_system, int
*processes_arrived) {
    for (int i = 0; i < num_processes; i++) {
        if (!is_in_system[i] && procs[i].arrival_time <= current_time) {
            queue[*rear] = i;
            *rear = (*rear + 1) % QUEUE_SIZE;
            is_in_system[i] = 1;
            (*processes_arrived)++;
        }
    }
}

void calculate_rr(struct Process *procs, int num_processes, int time_quantum,
                  struct GanttSegment *gantt, int *gantt_idx_ptr) {

    if (num_processes <= 0) {
        return;
    }

    int current_time = procs[0].arrival_time;
    int completed = 0;
    int queue[QUEUE_SIZE];
    int front = 0, rear = 0;
    int is_in_system[MAX_PROCESSES] = {0};
    int processes_arrived = 0;

    enqueue_new_arrivals(current_time, procs, num_processes, queue, &rear,
is_in_system, &processes_arrived);

    while (completed < num_processes) {
        if (front == rear) {
            if (processes_arrived < num_processes) {
                int next_arrival_time = -1;
                for (int i = 0; i < num_processes; i++) {
                    if (!is_in_system[i]) {
                        if (next_arrival_time == -1 || procs[i].arrival_time <
next_arrival_time) {
                            next_arrival_time = procs[i].arrival_time;
                        }
                    }
                }
```

```
                if (next_arrival_time > current_time) {
                    gantt[*gantt_idx_ptr].process_id = -1;
                    gantt[*gantt_idx_ptr].start_time = current_time;
                    gantt[*gantt_idx_ptr].end_time = next_arrival_time;
                    (*gantt_idx_ptr)++;
                    current_time = next_arrival_time;
                }
                enqueue_new_arrivals(current_time, procs, num_processes, queue,
&rear, is_in_system, &processes_arrived);
            } else {
                break;
            }
            continue;
        }

        int idx = queue[front];
        front = (front + 1) % QUEUE_SIZE;
        struct Process *p = &procs[idx];

        int exec_time = (p->remaining_time < time_quantum) ? p->remaining_time :
time_quantum;

        gantt[*gantt_idx_ptr].process_id = p->id;
        gantt[*gantt_idx_ptr].start_time = current_time;
        gantt[*gantt_idx_ptr].end_time = current_time + exec_time;
        (*gantt_idx_ptr)++;

        current_time += exec_time;
        p->remaining_time -= exec_time;

        enqueue_new_arrivals(current_time, procs, num_processes, queue, &rear,
is_in_system, &processes_arrived);

        if (p->remaining_time > 0) {
            queue[rear] = idx;
            rear = (rear + 1) % QUEUE_SIZE;
        } else {
            completed++;
            p->completion_time = current_time;
            p->turnaround_time = p->completion_time - p->arrival_time;
            p->waiting_time = p->turnaround_time - p->burst_time;
        }
    }
}

void display_gantt_chart(struct GanttSegment *gantt, int gantt_idx) {
    if (gantt_idx <= 0) return;

    printf("\n--- Gantt Chart ---\n");

    printf("|");
    for (int i = 0; i < gantt_idx; i++) {
        if (gantt[i].process_id == -1) {
            printf(" IDLE |");
        } else {
            if (gantt[i].process_id < 10) {
                 printf("  P%d  |", gantt[i].process_id);
            } else {
                 printf(" P%d  |", gantt[i].process_id);
            }
        }
    }
    printf("\n");
```

```
    printf("%-7d", gantt[0].start_time);
    for (int i = 0; i < gantt_idx; i++) {
        printf("%-7d", gantt[i].end_time);
    }
    printf("\n");
}

void display_results(struct Process *procs, int num_processes) {
    printf("\n--- Round Robin Scheduling Results ---\n");
    if (num_processes <= 0) {
        printf("No processes were scheduled.\n");
        return;
    }
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround
Time\n");
    printf("----------------------------------------------------------------
\n");
    float total_wt = 0, total_tat = 0;

    for (int i = 0; i < num_processes; i++) {
        struct Process *p = &procs[i];
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n",
               p->id,
               p->arrival_time,
               p->burst_time,
               p->waiting_time,
               p->turnaround_time);

        total_wt += p->waiting_time;
        total_tat += p->turnaround_time;
    }

    printf("----------------------------------------------------------------
\n");
    printf("Average Waiting Time: %.2f\n", total_wt / num_processes);
    printf("Average Turnaround Time: %.2f\n", total_tat / num_processes);
}
```

Practical No.7

```
Problem Statement 7: Memory Allocation â First Fit
Write a program to simulate First Fit memory allocation strategy.
The program should allocate each process to the first available memory block
that is large enough to accommodate it.
Display the memory allocation table and identify any unused or fragmented
memory.
*/

#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10

int main() {
    int block_sizes[MAX_BLOCKS], process_sizes[MAX_PROCESSES];
    int block_count, process_count;
    int allocation[MAX_PROCESSES];

    printf("Enter number of memory blocks (max %d): ", MAX_BLOCKS);
    scanf("%d", &block_count);
    if (block_count <= 0 || block_count > MAX_BLOCKS) {
        printf("Invalid number of blocks. Exiting.\n");
        return 1;
    }
```

```
    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < block_count; i++) {
        printf("Block %d size: ", i+1);
        scanf("%d", &block_sizes[i]);
    }

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &process_count);
    if (process_count <= 0 || process_count > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of processes:\n");
    for (int i = 0; i < process_count; i++) {
        printf("Process %d size: ", i+1);
        scanf("%d", &process_sizes[i]);
        allocation[i] = -1;
    }

    for (int i = 0; i < process_count; i++) {
        for (int j = 0; j < block_count; j++) {
            if (block_sizes[j] >= process_sizes[i]) {
                allocation[i] = j;
                block_sizes[j] -= process_sizes[i];
                break;
            }
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Remaining\n");
    printf("-----------------------------------------------------\n");
    for (int i = 0; i < process_count; i++) {
        if (allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\n",
                   i + 1,
                   process_sizes[i],
                   allocation[i] + 1,
                   block_sizes[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\t%s\t\t%s\n",
                   i + 1,
                   process_sizes[i],
                   "Not Allocated",
                   "N/A");
        }
    }

    printf("\nUnused / Fragmented Memory in Blocks:\n");
    int total_fragmented = 0;
    for (int i = 0; i < block_count; i++) {
        if (block_sizes[i] > 0) {
            printf("Block %d: %d units\n", i + 1, block_sizes[i]);
            total_fragmented += block_sizes[i];
        }
    }
    if (total_fragmented == 0) {
        printf("No unused or fragmented memory remaining.\n");
    }

    return 0;
}
```

Practical No.9

```
Problem Statement 8: Memory Allocation â Best Fit
Write a program to simulate Best Fit memory allocation strategy.
The program should allocate each process to the smallest available block that
can hold it.
Display the final allocation and show internal fragmentation if any.
*/

#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10

int main() {
    int block_sizes[MAX_BLOCKS], original_block_sizes[MAX_BLOCKS],
process_sizes[MAX_PROCESSES];
    int block_count, process_count;
    int allocation[MAX_PROCESSES];

    printf("Enter number of memory blocks (max %d): ", MAX_BLOCKS);
    scanf("%d", &block_count);
    if (block_count <= 0 || block_count > MAX_BLOCKS) {
        printf("Invalid number of blocks. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < block_count; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &block_sizes[i]);
        original_block_sizes[i] = block_sizes[i];
    }

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &process_count);
    if (process_count <= 0 || process_count > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of processes:\n");
    for (int i = 0; i < process_count; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &process_sizes[i]);
        allocation[i] = -1; // Initialize all to not allocated
    }

    for (int i = 0; i < process_count; i++) {
        int best_idx = -1;
        for (int j = 0; j < block_count; j++) {
            if (block_sizes[j] >= process_sizes[i]) {
                if (best_idx == -1 || block_sizes[j] < block_sizes[best_idx]) {
                    best_idx = j;
                }
            }
        }
        if (best_idx != -1) {
            allocation[i] = best_idx;
            block_sizes[best_idx] -= process_sizes[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\tInternal Fragmentation\n");
    printf("-----------------------------------------------------------------
\n");
```

```
    int total_fragmentation = 0;
    for (int i = 0; i < process_count; i++) {
        if (allocation[i] != -1) {
            int frag = block_sizes[allocation[i]];
            printf("%d\t\t%d\t\t%d\t\t%d\n",
                   i + 1,
                   process_sizes[i],
                   allocation[i] + 1,
                   frag);
            total_fragmentation += frag;
        } else {
            printf("%d\t\t%d\t\t%s\t\t%s\n",
                   i + 1,
                   process_sizes[i],
                   "Not Allocated",
                   "N/A");
        }
    }


    printf("\nUnused blocks and their sizes:\n");
    int unused_total = 0;
    for (int i = 0; i < block_count; i++) {
        if (block_sizes[i] == original_block_sizes[i]) {
            printf("Block %d: %d units\n", i + 1, block_sizes[i]);
            unused_total += block_sizes[i];
        }
    }
    if (unused_total == 0) {
        printf("No unused blocks remaining.\n");
    }

    printf("\nTotal internal fragmentation: %d units\n", total_fragmentation);

    return 0;
}
```

Practical No.9

```
Problem Statement 9: Memory Allocation â Next Fit
Write a program to simulate Next Fit memory allocation strategy.
The program should continue searching for the next suitable memory block from
the last allocated position instead of starting from the beginning.
Display the memory allocation table and fragmentation details.
*/

#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10

int main() {
    int block_sizes[MAX_BLOCKS], original_block_sizes[MAX_BLOCKS],
process_sizes[MAX_PROCESSES];
    int block_count, process_count;
    int allocation[MAX_PROCESSES]; // allocation[i] holds index of block
allocated to process i, -1 if none

    printf("Enter number of memory blocks (max %d): ", MAX_BLOCKS);
    scanf("%d", &block_count);
    if (block_count <= 0 || block_count > MAX_BLOCKS) {
        printf("Invalid number of blocks. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of memory blocks:\n");
```

```
    for (int i = 0; i < block_count; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &block_sizes[i]);
        original_block_sizes[i] = block_sizes[i]; // Store original size for
fragmentation tracking
    }

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &process_count);
    if (process_count <= 0 || process_count > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of processes:\n");
    for (int i = 0; i < process_count; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &process_sizes[i]);
        allocation[i] = -1; // Initialize all as not allocated
    }

    int last_allocated = 0; // Last allocated block index

    // Next Fit allocation
    for (int i = 0; i < process_count; i++) {
        int allocated = 0;
        int j = last_allocated;
        int count = 0;
        while (count < block_count) {
            if (block_sizes[j] >= process_sizes[i]) {
                allocation[i] = j;
                block_sizes[j] -= process_sizes[i];
                last_allocated = j;
                allocated = 1;
                break;
            }
            j = (j + 1) % block_count;
            count++;
        }
        if (!allocated) {
            allocation[i] = -1; // Not allocated
        }
    }

    // Display allocation table
    printf("\nProcess No.\tProcess Size\tBlock No.\tFragmentation\n");
    printf("-------------------------------------------------------\n");
    int total_fragmentation = 0;
    for (int i = 0; i < process_count; i++) {
        if (allocation[i] != -1) {
            int frag = block_sizes[allocation[i]];
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, process_sizes[i],
allocation[i] + 1, frag);
            total_fragmentation += frag;
        } else {
            printf("%d\t\t%d\t\t%s\t\t%s\n", i + 1, process_sizes[i], "Not
Allocated", "N/A");
        }
    }

    // Display unused blocks
    printf("\nUnused blocks and their sizes:\n");
    int unused_total = 0;
    for (int i = 0; i < block_count; i++) {
```

```
        if (block_sizes[i] == original_block_sizes[i]) {
            printf("Block %d: %d units\n", i + 1, block_sizes[i]);
            unused_total += block_sizes[i];
        }
    }
    if (unused_total == 0) {
        printf("No unused blocks remaining.\n");
    }

    printf("\nTotal internal fragmentation: %d units\n", total_fragmentation);

    return 0;
}
```

Practical No.10

```
Problem Statement 10: Memory Allocation â Worst Fit
Write a program to simulate Worst Fit memory allocation strategy.
The program should allocate each process to the largest available memory block.
Display the memory allocation results and any unused space.
*/

#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10

int main() {
    int block_sizes[MAX_BLOCKS], original_block_sizes[MAX_BLOCKS],
process_sizes[MAX_PROCESSES];
    int block_count, process_count;
    int allocation[MAX_PROCESSES];

    printf("Enter number of memory blocks (max %d): ", MAX_BLOCKS);
    scanf("%d", &block_count);
    if (block_count <= 0 || block_count > MAX_BLOCKS) {
        printf("Invalid number of blocks. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < block_count; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &block_sizes[i]);
        original_block_sizes[i] = block_sizes[i];
    }

    printf("Enter number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &process_count);
    if (process_count <= 0 || process_count > MAX_PROCESSES) {
        printf("Invalid number of processes. Exiting.\n");
        return 1;
    }

    printf("Enter sizes of processes:\n");
    for (int i = 0; i < process_count; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &process_sizes[i]);
        allocation[i] = -1;
    }

    for (int i = 0; i < process_count; i++) {
        int worst_idx = -1;
        int max_size = -1;
        for (int j = 0; j < block_count; j++) {
```

```
            if (block_sizes[j] >= process_sizes[i] && block_sizes[j] > max_size)
{
                max_size = block_sizes[j];
                worst_idx = j;
            }
        }
        if (worst_idx != -1) {
            allocation[i] = worst_idx;
            block_sizes[worst_idx] -= process_sizes[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock No.\tFragmentation / Unused
Space\n");

printf("---------------------------------------------------------------------
\n");
    int total_fragmentation = 0;
    for (int i = 0; i < process_count; i++) {
        if (allocation[i] != -1) {
            int frag = block_sizes[allocation[i]];
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, process_sizes[i],
allocation[i] + 1, frag);
            total_fragmentation += frag;
        } else {
            printf("%d\t\t%d\t\t%s\t\t%s\n", i + 1, process_sizes[i], "Not
Allocated", "N/A");
        }
    }


    printf("\nUnused blocks and their sizes:\n");
    int unused_total = 0;
    for (int i = 0; i < block_count; i++) {
        if (block_sizes[i] == original_block_sizes[i]) {
            printf("Block %d: %d units\n", i + 1, block_sizes[i]);
            unused_total += block_sizes[i];
        }
    }
    if (unused_total == 0) {
        printf("No unused blocks remaining.\n");
    }

    printf("\nTotal internal fragmentation: %d units\n", total_fragmentation);

    return 0;
}
```

Practical No.11

```
/*
Problem Statement 11: Page Replacement â FIFO
Write a program to simulate the First In First Out (FIFO) page replacement
algorithm.
The program should accept a page reference string and number of frames as input,
simulate the process of page replacement, and display the total number of page
faults.
*/

#include <stdio.h>
#define MAX_FRAMES 10
#define MAX_PAGES 100

int main() {
```

```
int frames[MAX_FRAMES];
int page_string[MAX_PAGES];
int num_frames, num_pages;
int front = 0, rear = 0;
int page_faults = 0;

printf("Enter number of frames (max %d): ", MAX_FRAMES);
scanf("%d", &num_frames);
if (num_frames <= 0 || num_frames > MAX_FRAMES) {
    printf("Invalid number of frames. Exiting.\n");
    return 1;
}

printf("Enter number of pages in reference string (max %d): ", MAX_PAGES);
scanf("%d", &num_pages);
if (num_pages <= 0 || num_pages > MAX_PAGES) {
    printf("Invalid number of pages. Exiting.\n");
    return 1;
}

printf("Enter the page reference string:\n");
for (int i = 0; i < num_pages; i++) {
    printf("Page %d: ", i + 1);
    scanf("%d", &page_string[i]);
}


for (int i = 0; i < num_frames; i++) {
    frames[i] = -1;
}

printf("\nFIFO Page Replacement Process:\n");
for (int i = 0; i < num_pages; i++) {
    int page = page_string[i];
    int found = 0;
    for (int j = 0; j < num_frames; j++) {
        if (frames[j] == page) {
            found = 1;
            break;
        }
    }

    if (!found) {
        frames[rear] = page;
        rear = (rear + 1) % num_frames;
        page_faults++;
    }

    printf("After reference to page %d: ", page);
    for (int j = 0; j < num_frames; j++) {
        if (frames[j] != -1)
            printf("%d ", frames[j]);
        else
            printf("- ");
    }
    if (found)
        printf("[Hit]");
    else
        printf("[Page Fault]");
    printf("\n");
}

printf("\nTotal number of page faults: %d\n", page_faults);
```

```
    return 0;
}
```

Practical No.12

```
Problem Statement 12: Page Replacement â LRU
Write a program to simulate the Least Recently Used (LRU) page replacement
algorithm.
The program should display the frame contents after each page reference and the
total number of page faults.
*/
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 10
#define MAX_PAGES 100

void simulate_lru(int *frames, int num_frames, int *page_string, int num_pages);

int main() {
    int frames[MAX_FRAMES];
    int page_string[MAX_PAGES];
    int num_frames;
    int num_pages;

    printf("Enter number of frames (max %d): ", MAX_FRAMES);
    scanf("%d", &num_frames);
    if (num_frames <= 0 || num_frames > MAX_FRAMES) {
        printf("Invalid number of frames. Exiting.\n");
        return 1;
    }

    printf("Enter number of pages in reference string (max %d): ", MAX_PAGES);
    scanf("%d", &num_pages);
    if (num_pages <= 0 || num_pages > MAX_PAGES) {
        printf("Invalid number of pages. Exiting.\n");
        return 1;
    }

    printf("Enter the page reference string:\n");
    for (int i = 0; i < num_pages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &page_string[i]);
    }

    simulate_lru(frames, num_frames, page_string, num_pages);

    return 0;
}

void simulate_lru(int *frames, int num_frames, int *page_string, int num_pages)
{

    int page_faults = 0;
    int *last_used = (int *)calloc(num_frames, sizeof(int));

    for (int i = 0; i < num_frames; i++) {
        *(frames + i) = -1;
    }

    for (int i = 0; i < num_pages; i++) {
        int page = *(page_string + i);
        int found_index = -1;
```

```
        for (int j = 0; j < num_frames; j++) {
            if (*(frames + j) == page) {
                found_index = j;
                break;
            }
        }

        if (found_index != -1) {
            *(last_used + found_index) = i;
            printf("After reference to page %d: ", page);
            for (int k = 0; k < num_frames; k++) {
                if (*(frames + k) != -1)
                    printf("%d ", *(frames + k));
                else
                    printf("- ");
            }
            printf("[Hit]\n");
        } else {
            page_faults++;

            int lru_index = 0;
            int min_used_time = *(last_used + 0);

            for (int j = 0; j < num_frames; j++) {
                if (*(frames + j) == -1) {
                    lru_index = j;
                    goto replace;
                }
            }

            for (int j = 1; j < num_frames; j++) {
                if (*(last_used + j) < min_used_time) {
                    min_used_time = *(last_used + j);
                    lru_index = j;
                }
            }

        replace:
            *(frames + lru_index) = page;
            *(last_used + lru_index) = i;

            printf("After reference to page %d: ", page);
            for (int k = 0; k < num_frames; k++) {
                if (*(frames + k) != -1)
                    printf("%d ", *(frames + k));
                else
                    printf("- ");
            }
            printf("[Page Fault]\n");
        }
    }

    printf("\nTotal number of page faults: %d\n", page_faults);
    free(last_used);
}
```

Practical No.13

Problem Statement 13: Page Replacement â Optimal
Write a program to simulate the Optimal Page Replacement algorithm.
The program should replace the page that will not be used for the longest period of time in the future and display the page replacement steps and page fault count.
*/

```c
#include <stdio.h>
#define MAX_FRAMES 10
#define MAX_PAGES 100

int findOptimal(int frames[], int num_frames, int page_string[], int num_pages,
int current_index) {
    int farthest = current_index;
    int index = -1;

    for (int i = 0; i < num_frames; i++) {
        int j;
        for (j = current_index + 1; j < num_pages; j++) {
            if (frames[i] == page_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }

        if (j == num_pages)
            return i;
    }

    if (index == -1)
        return 0;
    else
        return index;
}

int main() {
    int frames[MAX_FRAMES];
    int page_string[MAX_PAGES];
    int num_frames, num_pages;
    int page_faults = 0;

    printf("Enter number of frames (max %d): ", MAX_FRAMES);
    scanf("%d", &num_frames);
    if (num_frames <= 0 || num_frames > MAX_FRAMES) {
        printf("Invalid number of frames. Exiting.\n");
        return 1;
    }

    printf("Enter number of pages in reference string (max %d): ", MAX_PAGES);
    scanf("%d", &num_pages);
    if (num_pages <= 0 || num_pages > MAX_PAGES) {
        printf("Invalid number of pages. Exiting.\n");
        return 1;
    }

    printf("Enter the page reference string:\n");
    for (int i = 0; i < num_pages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &page_string[i]);
    }

    for (int i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }

    int count = 0;
```

```
    printf("\nOptimal Page Replacement Process:\n");
    for (int i = 0; i < num_pages; i++) {
        int page = page_string[i];
        int found = 0;

        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            page_faults++;
            if (count < num_frames) {
                frames[count++] = page;
            } else {
                int replace_index = findOptimal(frames, num_frames, page_string,
num_pages, i);
                frames[replace_index] = page;
            }
        }

        printf("After reference to page %d: ", page);
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] != -1)
                printf("%d ", frames[j]);
            else
                printf("- ");
        }
        if (found)
            printf("[Hit]");
        else
            printf("[Page Fault]");
        printf("\n");
    }

    printf("\nTotal number of page faults: %d\n", page_faults);

    return 0;
}
```

```
// Extra codes
/*
```

Practical No.14(Extra)

```
MRU Page Replacement Algorithm
*/
#include <stdio.h>
#define MAX_FRAMES 10
#define MAX_PAGES 30

int frames[MAX_FRAMES];
int pages[MAX_PAGES];
int recent[MAX_FRAMES];
int num_frames, num_pages;

void initialize() {
    printf("=== MRU Page Replacement ===\n\n");
    printf("Enter number of frames: ");
    scanf("%d", &num_frames);
    printf("Enter number of pages: ");
    scanf("%d", &num_pages);
    printf("Enter page reference string: ");
    for(int i = 0; i < num_pages; i++) {
        scanf("%d", &pages[i]);
    }
    for(int i = 0; i < num_frames; i++) {
        frames[i] = -1;
        recent[i] = -1;
    }
}

void mru() {
    int page_faults = 0;
    int time = 0;
    printf("\nPage Replacement Process:\n");

    for(int i = 0; i < num_pages; i++) {
        int found = 0;
        for(int j = 0; j < num_frames; j++) {
            if(frames[j] == pages[i]) {
                found = 1;
                recent[j] = time++;
                break;
            }
        }

        if(!found) {
            int mru_index = 0;
            for(int j = 1; j < num_frames; j++) {
                if(recent[j] > recent[mru_index]) {
                    mru_index = j;
                }
            }
            frames[mru_index] = pages[i];
            recent[mru_index] = time++;
            page_faults++;
        }

        printf("Page %d -> [", pages[i]);
        for(int j = 0; j < num_frames; j++) {
            if(frames[j] == -1) printf(" - ");
            else printf(" %d ", frames[j]);
        }
        printf("] %s\n", found ? "Hit" : "Miss");
    }

    printf("\nTotal Page Faults: %d\n", page_faults);
    printf("Hit Ratio: %.2f%%\n", ((num_pages - page_faults) * 100.0) /
```

```
num_pages);
    printf("Miss Ratio: %.2f%%\n", (page_faults * 100.0) / num_pages);
}

int main() {
    initialize();
    mru();
    return 0;
}
```