VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

Varun Raj S (1BM21CS264)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

June-2023 to September-2023

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “OPERATING SYSTEMS” carried out by **Varun Raj S (1BM21CS264),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge:
Prof. Basavaraj Jakkali
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Lab Program No.	**Program Details**	**Page No.**
1	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. a)FCFS b) SJF (pre-emptive & Non-pre-emptive)	
2	Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. Priority (pre-emptive & Non-pre-emptive) Round Robin (Experiment with different quantum sizes for RR algorithm)	
3	Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue	
4	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling	
5	Write a C program to simulate producer-consumer problem using semaphores	
6	Write a C program to simulate the concept of Dining-Philosophers problem.	
7	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	
8	Write a C program to simulate deadlock detection	
9	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	

10	Write a C program to simulate paging technique of memory management	
11	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	
12	Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN	
13	Write a C program to simulate disk scheduling algorithms a) SSTF b) LOOK c) c-LOOK	

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System.
CO2	Analyse various Operating system strategies and techniques
CO3	Demonstrate the different functionalities of Operating System.
CO4	Conduct practical experiments to implement the functionalities of Operating system.

Lab Program 1

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

- FCFS
- SJF (pre-emptive & Non-pre-emptive)

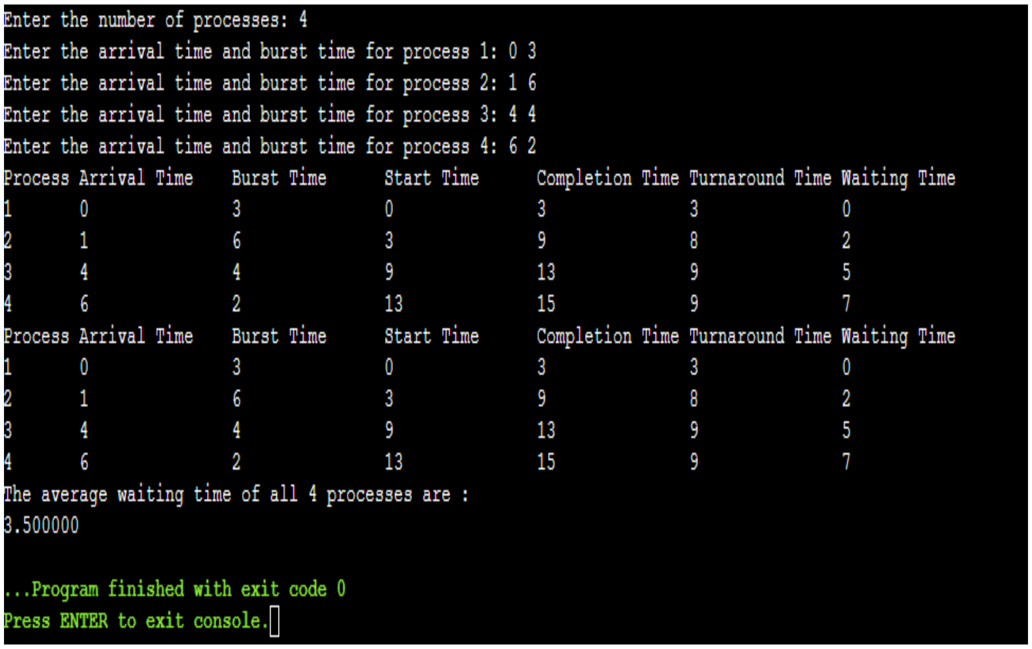
Source code:

First Come First Serve(FCFS):

```
#include<stdio.h>
typedef struct
{
   int pID,aT,bT,sT,cT,taT,wT;
} Process;

void calculateTimes(Process p[], int n)
{
   int currT = 0;
   for (int i = 0; i < n; i++)
   {
      p[i].sT = currT;
      p[i].cT = currT + p[i].bT;
      p[i].taT = p[i].cT - p[i].aT;
      p[i].wT = p[i].taT - p[i].bT;
      currT = p[i].cT;
   }
}

void displayp(Process p[], int n)
```

```
{
    printf("Process\tArrival Time\tBurst Time\tStart Time\tCompletion
Time\tTurnaround Time\tWaiting Time\n");

    for (int i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pID, p[i].aT,
               p[i].bT, p[i].sT, p[i].cT,
               p[i].taT, p[i].wT);
    }
}
void averageWaitingTime(Process p[], int n){
    printf("The average waiting time of all %d processes are :\n",n);
    float sum=0.0;
    int k;
    for(k=0;k<n;k++){
        sum+=p[k].wT;
    }
    float avg = (sum/n);
    printf("%f",avg);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process p[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the arrival time and burst time for process %d: ", i +
1);
        scanf("%d %d", &p[i].aT, &p[i].bT);
        p[i].pID = i + 1;
    }
    calculateTimes(p, n);
    displayp(p, n);

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].aT > p[j + 1].aT) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    calculateTimes(p, n);
    displayp(p, n);
    averageWaitingTime(p, n);
    return 0;
}
```

Output:

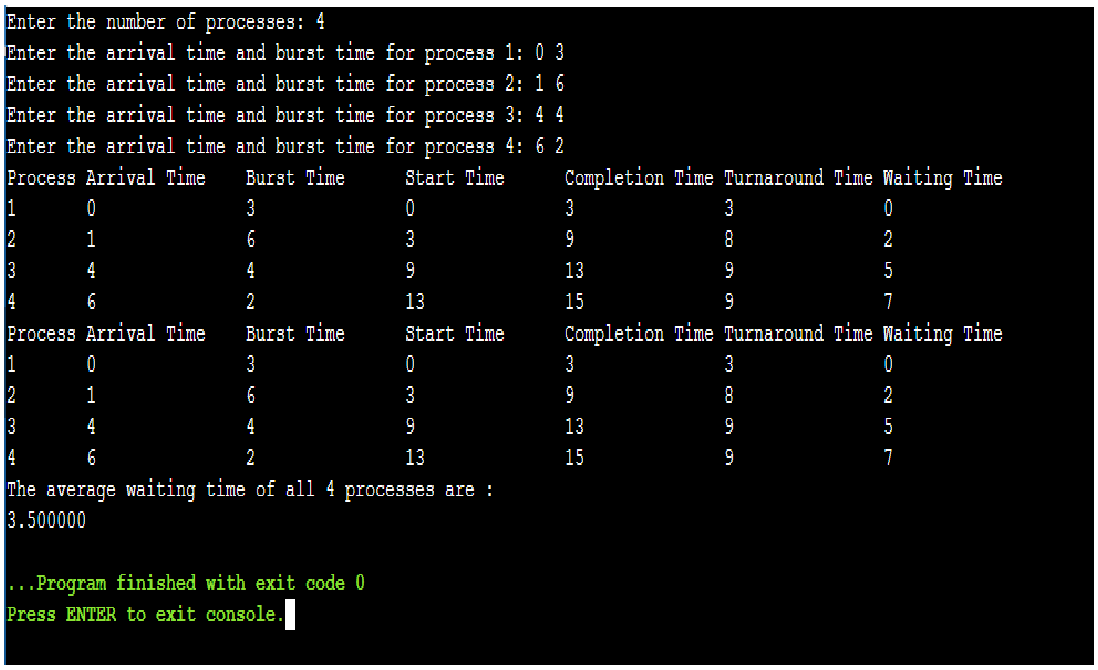


```
Enter the number of processes: 4
Enter the arrival time and burst time for process 1: 0 3
Enter the arrival time and burst time for process 2: 1 6
Enter the arrival time and burst time for process 3: 4 4
Enter the arrival time and burst time for process 4: 6 2
Process Arrival Time    Burst Time      Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7
Process Arrival Time    Burst Time      Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7
The average waiting time of all 4 processes are :
3.500000

...Program finished with exit code 0
Press ENTER to exit console.
```

Shortest Job First(SJF):

```
#include<stdio.h>
typedef struct
{
    int pID,aT,bT,sT,cT,taT,wT;
} Process;

void calculateTimes(Process p[], int n)
{
    int i,j,t, s,f;
    for(i=0;i<n-1;i++){
        for(j=0;j<(n-i-1);j++){
            s = (p[j].bT)+(p[j].aT);
            f = (p[j+1].bT)+(p[j+1].aT);
            if(s > f){
                t=f;
                f= s;
                s = t;
            }
        }
    }
    int currT = 0;
    for (int i = 0; i < n; i++)
    {
        p[i].sT = currT;
        p[i].cT = currT + p[i].bT;
        p[i].taT = p[i].cT - p[i].aT;
        p[i].wT = p[i].taT - p[i].bT;
        currT = p[i].cT;
    }
}

void displayp(Process p[], int n)
{
    printf("Process\tArrival Time\tBurst Time\tStart Time\tCompletion
Time\tTurnaround Time\tWaiting Time\n");

    for (int i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pID, p[i].aT,
               p[i].bT, p[i].sT, p[i].cT,
               p[i].taT, p[i].wT);
    }
}
void averageWaitingTime(Process p[], int n){
    printf("The average waiting time of all %d processes are :\n",n);
    float sum=0.0;
    int k;
    for(k=0;k<n;k++){
        sum+=p[k].wT;
    }
    float avg = (sum/n);
    printf("%f",avg);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process p[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the arrival time and burst time for process %d: ", i +
1);
        scanf("%d %d", &p[i].aT, &p[i].bT);
        p[i].pID = i + 1;
    }
    calculateTimes(p, n);
    displayp(p, n);
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].aT > p[j + 1].aT) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
    calculateTimes(p, n);
    displayp(p, n);
    averageWaitingTime(p, n);
    return 0;
}
```

Output:



```
Enter the number of processes: 4
Enter the arrival time and burst time for process 1: 0 3
Enter the arrival time and burst time for process 2: 1 6
Enter the arrival time and burst time for process 3: 4 4
Enter the arrival time and burst time for process 4: 6 2
Process Arrival Time    Burst Time      Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7
Process Arrival Time    Burst Time      Start Time      Completion Time Turnaround Time Waiting Time
1       0               3               0               3               3               0
2       1               6               3               9               8               2
3       4               4               9               13              9               5
4       6               2               13              15              9               7
The average waiting time of all 4 processes are :
3.500000

...Program finished with exit code 0
Press ENTER to exit console.
```

Lab Program 2:

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

- Priority (pre-emptive & Non-pre-emptive)
- Round Robin (Experiment with different quantum sizes for RR algorithm)

Source Code:

```
//Menu driven program for all three algorithms
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 10

struct Process
{
    int pid;
    int arr_time;
    int burst_time;
    int priority;
    int rem_time;
    int tat;
    int wt;
};


void priority_nonpreemptive(struct Process p[], int n)
{
    int i, j, count = 0, m;
    for (i = 0; i < n; i++)
    {
        if (p[i].arr_time == 0)
            count++;
    }
    if (count == n || count == 1)
    {
        if (count == n)
        {
            for (i = 0; i < n - 1; i++)
            {
                for (j = 0; j < n - i - 1; j++)
                {
                    if (p[j].priority > p[j + 1].priority)
                    {
                        struct Process temp = p[j];
                        p[j] = p[j + 1];
                        p[j + 1] = temp;
                    }
                }
            }
        }
        else
        {
            for (i = 1; i < n - 1; i++)
            {
                for (j = 1; j <= n - i - 1; j++)
                {
                    if (p[j].priority > p[j + 1].priority)
                    {
                        struct Process temp = p[j];
                        p[j] = p[j + 1];
                        p[j + 1] = temp;
                    }
                }
            }
        }
    }

    int total_time = 0;
    double total_tat = 0;
    double total_wt = 0;

    for (i = 0; i < n; i++)
    {
        total_time += p[i].burst_time;
        p[i].tat = total_time - p[i].arr_time;
        p[i].wt = p[i].tat - p[i].burst_time;

        total_tat += p[i].tat;
        total_wt += p[i].wt;
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++)
```

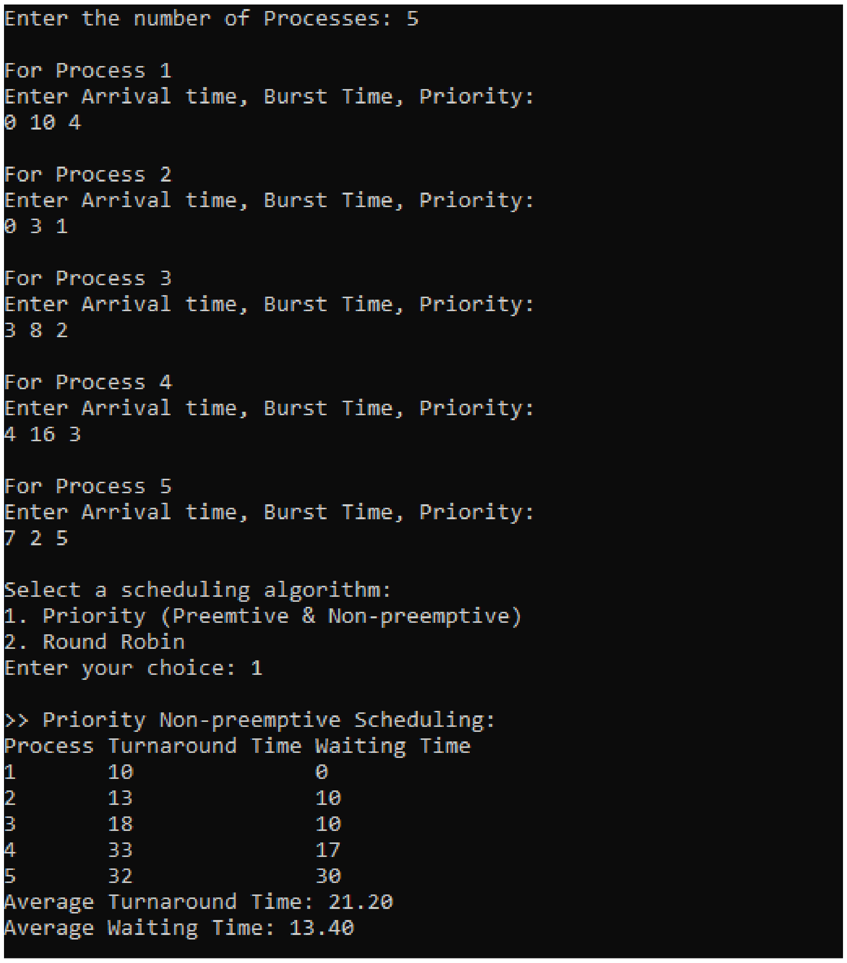
```
    {
        printf("%d\t%d\t\t%d\n", p[i].pid, p[i].tat, p[i].wt);
    }

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

void priority_preemptive(struct Process p[], int n)
{
    int total_time = 0, i;
    int completed = 0;

    while (completed < n)
    {
        int highest_priority = -1;
        int next_process = -1;

        for (i = 0; i < n; i++)
        {
            if (p[i].arr_time <= total_time && p[i].rem_time > 0)
            {
                if (highest_priority == -1 || p[i].priority < highest_priority)
                {
                    highest_priority = p[i].priority;
                    next_process = i;
                }
            }
        }

        if (next_process == -1)
        {
            total_time++;
            continue;
        }

        p[next_process].rem_time--;
        total_time++;

        if (p[next_process].rem_time == 0)
        {
            completed++;
            p[next_process].tat = total_time - p[next_process].arr_time;
            p[next_process].wt = p[next_process].tat -
p[next_process].burst_time;
        }
    }

    double total_tat = 0;
    double total_wt = 0;

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\n", p[i].pid, p[i].tat, p[i].wt);

        total_tat += p[i].tat;
        total_wt += p[i].wt;
    }

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

void round_robin(struct Process p[], int n, int quantum)
{
```

```
int total_time = 0, i;
int completed = 0;

printf("\nGantt Chart: \n");
while (completed < n)
{

    for (i = 0; i < n; i++)
    {
        if (p[i].arr_time <= total_time && p[i].rem_time > 0)
        {
            if (p[i].rem_time <= quantum)
            {
                printf("P%d ", p[i].pid);
                total_time += p[i].rem_time;
                p[i].rem_time = 0;
                p[i].tat = total_time - p[i].arr_time;
                p[i].wt = p[i].tat - p[i].burst_time;
                completed++;
            }
            else
            {
                printf("P%d ", p[i].pid);
                total_time += quantum;
                p[i].rem_time -= quantum;
            }
        }
    }
}

double total_tat = 0;
double total_wt = 0;
```

```
    printf("\n");
    printf("\nProcess\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\n", p[i].pid, p[i].tat, p[i].wt);

        total_tat += p[i].tat;
        total_wt += p[i].wt;
    }

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

int main()
{
    int n, quantum, i, choice;
    struct Process p[MAX_PROCESSES];
    printf("Enter the number of Processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("\nFor Process %d\n", i + 1);
        printf("Enter Arrival time, Burst Time, Priority:\n");
        scanf("%d%d%d",&p[i].arr_time,&p[i].burst_time,&p[i].priority);
        p[i].pid = i + 1;
        p[i].rem_time = p[i].burst_time;
        p[i].tat = 0;
        p[i].wt = 0;
    }
    printf("\nSelect a scheduling algorithm:\n");
    printf("1. Priority (Preemtive & Non-preemptive)\n");
    printf("2. Round Robin\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice)
    {
    case 1:
        printf("\n>> Priority Non-preemptive Scheduling:\n");
        priority_nonpreemptive(p, n);

        printf("\n>> Priority Preemptive Scheduling:\n");
        priority_preemptive(p, n);
        break;
    case 2:
        printf("\nEnter the quantum size for Round Robin: ");
        scanf("%d", &quantum);
        printf("\n>> Round Robin Scheduling (Quantum: %d):\n",
quantum);
        round_robin(p, n, quantum);
        break;
    default:
        printf("Invalid choice!\n");
        return 1;
    }
    return 0;
}
```

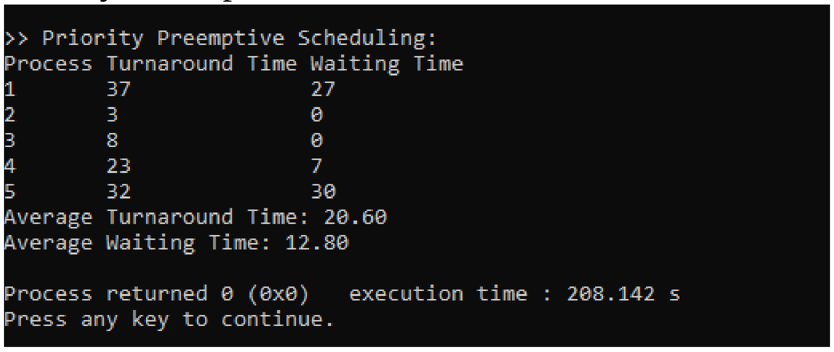
Output:

Priority Non-preemptive:



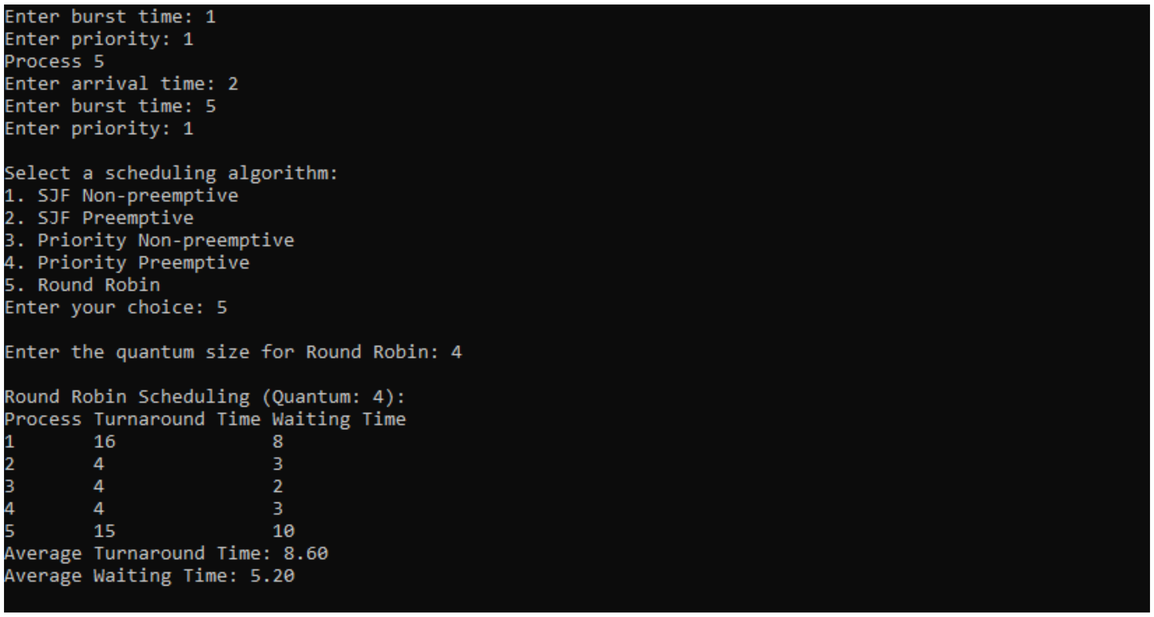
```
Enter the number of Processes: 5

For Process 1
Enter Arrival time, Burst Time, Priority:
0 10 4

For Process 2
Enter Arrival time, Burst Time, Priority:
0 3 1

For Process 3
Enter Arrival time, Burst Time, Priority:
3 8 2

For Process 4
Enter Arrival time, Burst Time, Priority:
4 16 3

For Process 5
Enter Arrival time, Burst Time, Priority:
7 2 5

Select a scheduling algorithm:
1. Priority (Preemtive & Non-preemptive)
2. Round Robin
Enter your choice: 1

>> Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1       10              0
2       13              10
3       18              10
4       33              17
5       32              30
Average Turnaround Time: 21.20
Average Waiting Time: 13.40
```

<u>Priority Preemptive:</u>



```
>> Priority Preemptive Scheduling:
Process Turnaround Time Waiting Time
1       37              27
2       3               0
3       8               0
4       23              7
5       32              30
Average Turnaround Time: 20.60
Average Waiting Time: 12.80

Process returned 0 (0x0)   execution time : 208.142 s
Press any key to continue.
```

<u>Round-Robin:</u>



```
Enter burst time: 1
Enter priority: 1
Process 5
Enter arrival time: 2
Enter burst time: 5
Enter priority: 1

Select a scheduling algorithm:
1. SJF Non-preemptive
2. SJF Preemptive
3. Priority Non-preemptive
4. Priority Preemptive
5. Round Robin
Enter your choice: 5

Enter the quantum size for Round Robin: 4

Round Robin Scheduling (Quantum: 4):
Process Turnaround Time Waiting Time
1       16              8
2       4               3
3       4               2
4       4               3
5       15              10
Average Turnaround Time: 8.60
Average Waiting Time: 5.20
```

<u>Lab Program 3</u>:

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

<u>Source Code</u>:

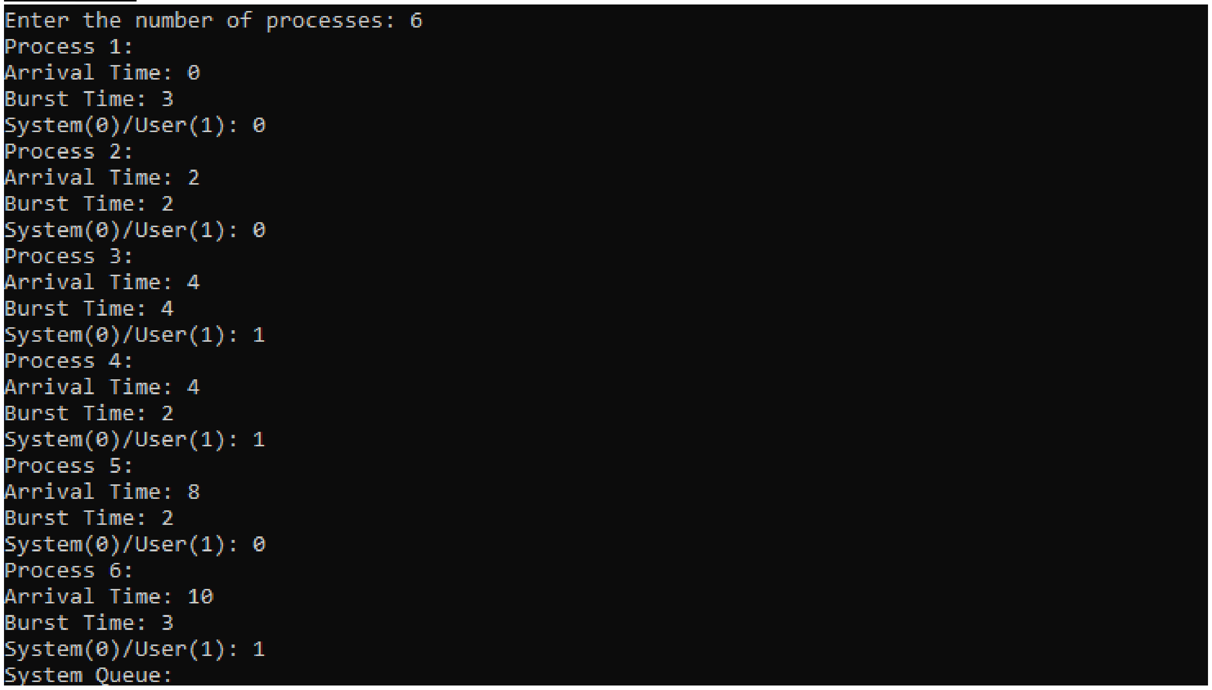
```
#include <stdio.h>
```

```
#define MAX_QUEUE_SIZE 100

// Structure to represent a process
typedef struct {
    int processID;
    int arrivalTime;
    int burstTime;
    int priority; // 0 for system process, 1 for user process
} Process;

// Function to execute a process
void executeProcess(Process process) {
    printf("Executing Process %d\n", process.processID);
    // Simulating the execution time of the process
    for (int i = 1; i <= process.burstTime; i++) {
        printf("Process %d: %d/%d\n", process.processID, i,
process.burstTime);
    }
    printf("Process %d executed\n", process.processID);
}

// Function to perform FCFS scheduling for a queue of processes
void scheduleFCFS(Process queue[], int size) {
    for (int i = 0; i < size; i++) {
        executeProcess(queue[i]);
    }
}

int main() {
    int numProcesses;
    Process processes[MAX_QUEUE_SIZE];

    // Reading the number of processes
```

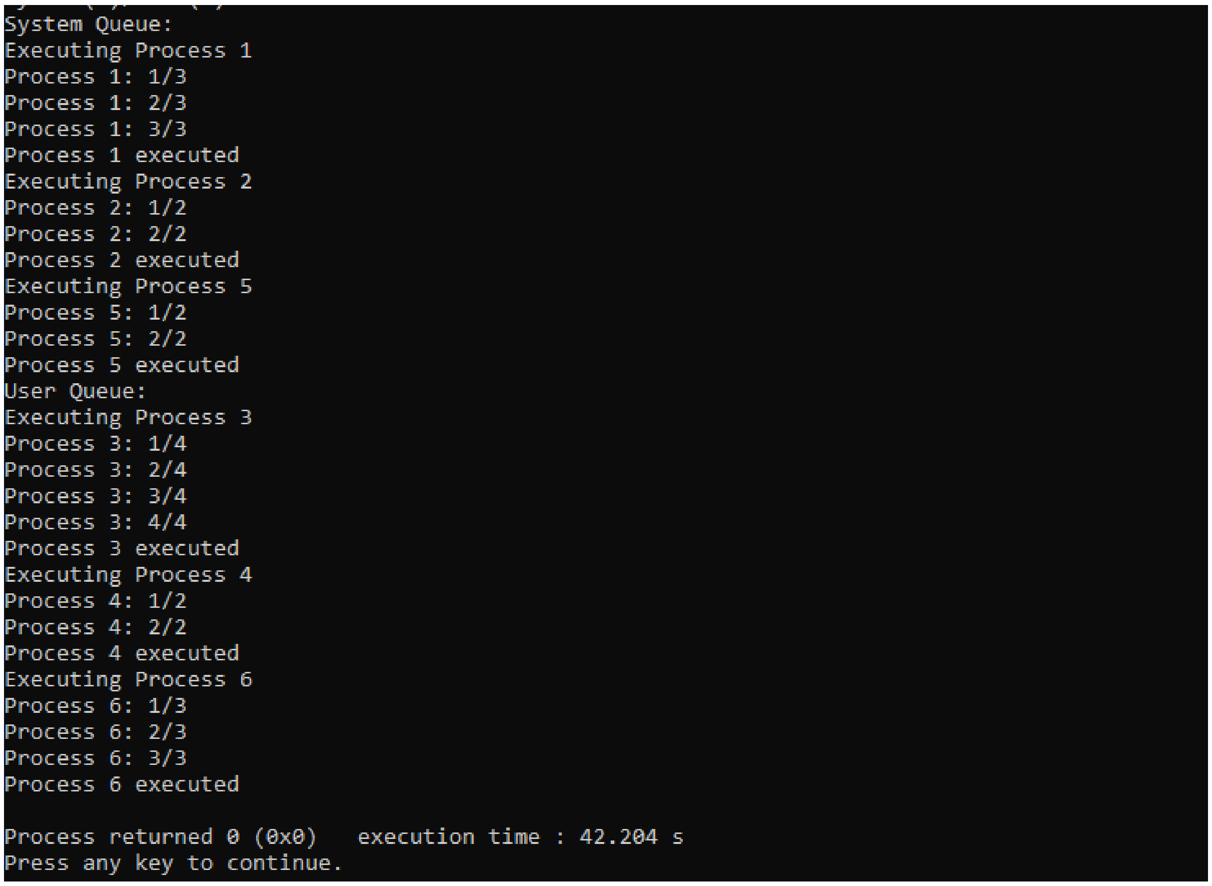
```
printf("Enter the number of processes: ");
scanf("%d", &numProcesses);

// Reading process details
for (int i = 0; i < numProcesses; i++) {
    printf("Process %d:\n", i + 1);
    printf("Arrival Time: ");
    scanf("%d", &processes[i].arrivalTime);
    printf("Burst Time: ");
    scanf("%d", &processes[i].burstTime);
    printf("System(0)/User(1): ");
    scanf("%d", &processes[i].priority);
    processes[i].processID = i + 1;
}

// Separate system and user processes into different queues
Process systemQueue[MAX_QUEUE_SIZE];
int systemQueueSize = 0;
Process userQueue[MAX_QUEUE_SIZE];
int userQueueSize = 0;

for (int i = 0; i < numProcesses; i++) {
    if (processes[i].priority == 0) {
        systemQueue[systemQueueSize++] = processes[i];
    } else {
        userQueue[userQueueSize++] = processes[i];
    }
}

// Execute system queue processes first
printf("System Queue:\n");
scheduleFCFS(systemQueue, systemQueueSize);
```

```
    // Execute user queue processes
    printf("User Queue:\n");
    scheduleFCFS(userQueue, userQueueSize);

    return 0;
}
```

Output:



```
Enter the number of processes: 6
Process 1:
Arrival Time: 0
Burst Time: 3
System(0)/User(1): 0
Process 2:
Arrival Time: 2
Burst Time: 2
System(0)/User(1): 0
Process 3:
Arrival Time: 4
Burst Time: 4
System(0)/User(1): 1
Process 4:
Arrival Time: 4
Burst Time: 2
System(0)/User(1): 1
Process 5:
Arrival Time: 8
Burst Time: 2
System(0)/User(1): 0
Process 6:
Arrival Time: 10
Burst Time: 3
System(0)/User(1): 1
System Queue:
```



```
System Queue:
Executing Process 1
Process 1: 1/3
Process 1: 2/3
Process 1: 3/3
Process 1 executed
Executing Process 2
Process 2: 1/2
Process 2: 2/2
Process 2 executed
Executing Process 5
Process 5: 1/2
Process 5: 2/2
Process 5 executed
User Queue:
Executing Process 3
Process 3: 1/4
Process 3: 2/4
Process 3: 3/4
Process 3: 4/4
Process 3 executed
Executing Process 4
Process 4: 1/2
Process 4: 2/2
Process 4 executed
Executing Process 6
Process 6: 1/3
Process 6: 2/3
Process 6: 3/3
Process 6 executed

Process returned 0 (0x0)   execution time : 42.204 s
Press any key to continue.
```

Lab Program 4:

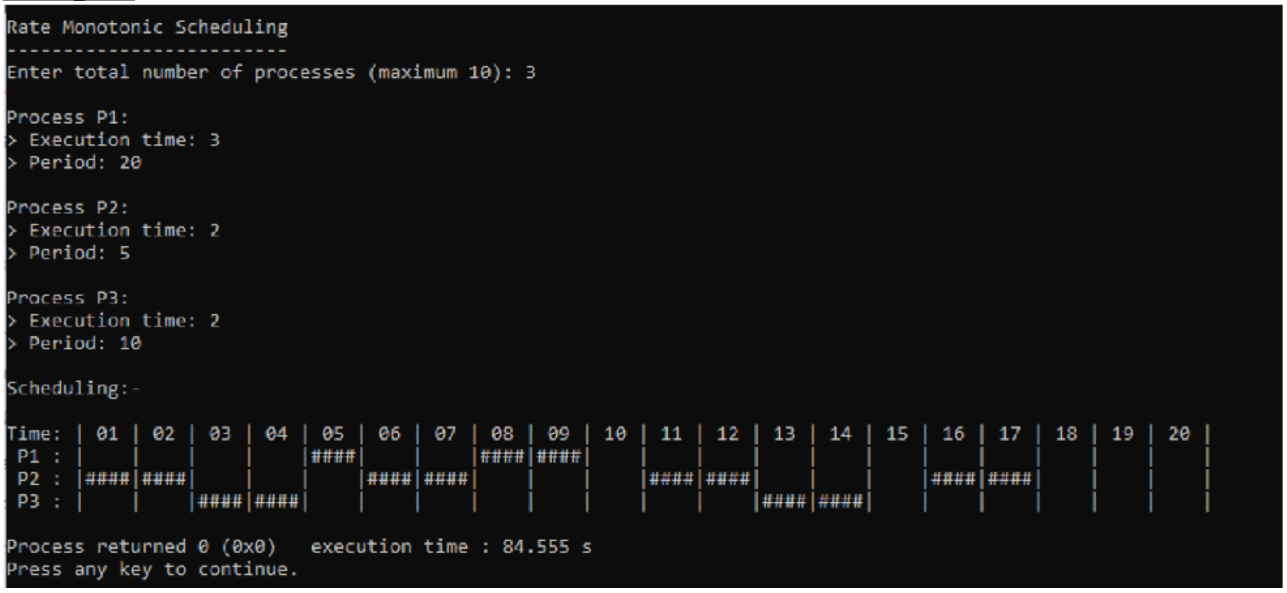
Write a C program to simulate Real-Time CPU Scheduling algorithms:
a) Rate- Monotonic
b) Earliest-deadline First

Source Code:
Rate – Monotonic:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAX_PROCESS 10

int num_of_process = 3;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
remain_time[MAX_PROCESS];

// collecting details of processes
void get_process_info()
{
    printf("Enter total number of processes (maximum %d): ",
MAX_PROCESS);
    scanf("%d", &num_of_process);
    if (num_of_process < 1)
    {
        printf("Do you really want to schedule %d processes? -_-\n",
num_of_process);
        exit(0);
    }
    for (int i = 0; i < num_of_process; i++)
    {
        printf("\nProcess P%d: \n", i + 1);
        printf("> Execution time: ");
        scanf("%d", &execution_time[i]);
        remain_time[i] = execution_time[i];

        printf("> Period: ");
        scanf("%d", &period[i]);
    }
}

// get maximum of three numbers
int max(int a, int b, int c)
{
    if (a >= b && a >= c)
        return a;
    else if (b >= a && b >= c)
        return b;
    else
        return c;
}

// calculating the observation time for scheduling timeline
int get_observation_time()
{
    return max(period[0], period[1], period[2]);
}

// print scheduling sequence
void print_schedule(int process_list[], int cycles)
{
    printf("\nScheduling:-\n\n");
    printf("Time: ");
    for (int i = 0; i < cycles; i++)
    {
        if (i < 9)
            printf("| 0%d ", i + 1);
        else
            printf("| %d ", i + 1);
    }
    printf("|\n");
    for (int i = 0; i < num_of_process; i++)
    {
        printf(" P%d : ", i + 1);
        for (int j = 0; j < cycles; j++)
        {
            if (process_list[j] == i + 1)
                printf("|####");
            else
                printf("|    ");
        }
        printf("|\n");
    }
}
void rate_monotonic(int time)
{
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    {
        utilization += (1.0 * execution_time[i]) / period[i];
    }
    int n = num_of_process;
    if (utilization > n * (pow(2, 1.0 / n) - 1))
    {
        printf("\nGiven problem is not schedulable under said scheduling
algorithm.\n");
        exit(0);
    }
    int process_list[time];
    int min = 999, next_process = 0;
    for (int i = 0; i < time; i++)
    {
        min = 1000;
        for (int j = 0; j < num_of_process; j++)
        {
            if (remain_time[j] > 0)
            {
                if (min > period[j])
                {
                    min = period[j];
                    next_process = j;
                }
            }
        }
        if (remain_time[next_process] > 0)
        {
            process_list[i] = next_process + 1; // +1 for catering 0 array
index.
            remain_time[next_process] -= 1;
        }
        for (int k = 0; k < num_of_process; k++)
        {
            if ((i + 1) % period[k] == 0)
            {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
```

```
}

int main(int argc, char *argv[])
{
    printf("\nRate Monotonic Scheduling\n");
    printf("------------------------\n");
    get_process_info(); // collecting processes detail
    int observation_time = get_observation_time();
    rate_monotonic(observation_time);

    return 0;
}
```

<u>Output</u>:

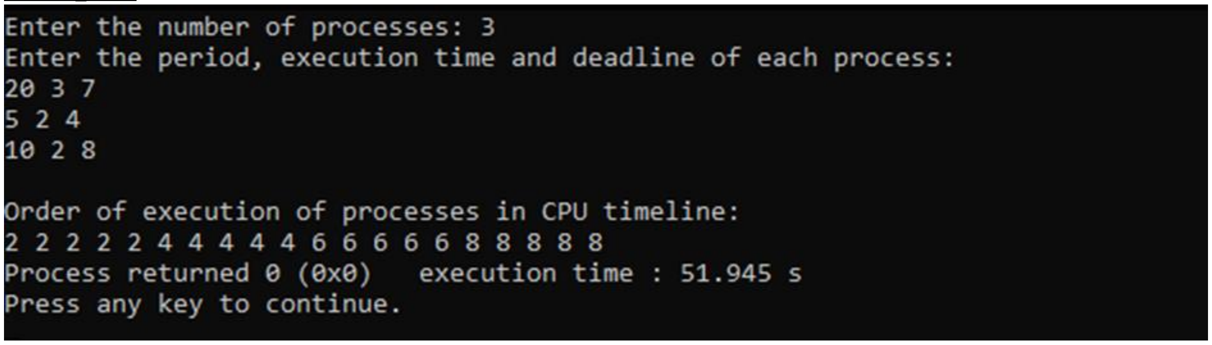


```
Rate Monotonic Scheduling
------------------------
Enter total number of processes (maximum 10): 3

Process P1:
> Execution time: 3
> Period: 20

Process P2:
> Execution time: 2
> Period: 5

Process P3:
> Execution time: 2
> Period: 10

Scheduling:-

Time: | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
 P1 : |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
 P2 : |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
 P3 : |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |

Process returned 0 (0x0)   execution time : 84.555 s
Press any key to continue.
```

Earliest Deadline First:

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

int n;
int period[MAX], execution[MAX], deadline[MAX];
int ready[MAX], task[MAX];
int time = 0;

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

void sort() {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (deadline[i] > deadline[j]) {
                swap(&period[i], &period[j]);
                swap(&execution[i], &execution[j]);
                swap(&deadline[i], &deadline[j]);
            }
        }
    }
}

int lcm(int arr[], int n)
{
    int ans = arr[0];
    for (int i = 1; i < n; i++)
        ans = (((arr[i] * ans)) / (gcd(arr[i], ans)));
    return ans;
}

void schedule() {
    int i, j;
    for (i = 0; i < n; i++) {
        if (time % period[i] == 0) {
            ready[i] = 1;
        }
    }
    for (i = 0; i < n; i++) {
        if (ready[i] == 1) {
            int min_deadline = 1000000000;
            int min_index = -1;
            for (j = 0; j < n; j++) {
                if (ready[j] == 1 && deadline[j] < min_deadline) {
                    min_deadline = deadline[j];
                    min_index = j;
                }
            }
            task[min_index] += execution[min_index];
            deadline[min_index] += period[min_index];
            ready[min_index] = 0;
        }
    }
}

int main() {
    int total_time;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the period, execution time and deadline of each
process:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d %d", &period[i], &execution[i], &deadline[i]);
        ready[i] = task[i] = 0;
    }
    sort();
    printf("\nOrder of execution of processes in CPU timeline:\n");
    total_time = lcm(period, n);
    while (time < total_time) { // assuming total time is 100
        schedule();
        printf("%d ", task[0]);
        time++;
    }
    return 0;
}
```

Output:



```
Enter the number of processes: 3
Enter the period, execution time and deadline of each process:
20 3 7
5 2 4
10 2 8

Order of execution of processes in CPU timeline:
2 2 2 2 2 4 4 4 4 4 6 6 6 6 6 8 8 8 8 8
Process returned 0 (0x0)   execution time : 51.945 s
Press any key to continue.
```

<u>Lab Program 5</u>:

Write a C program to simulate producer-consumer problem using semaphores.

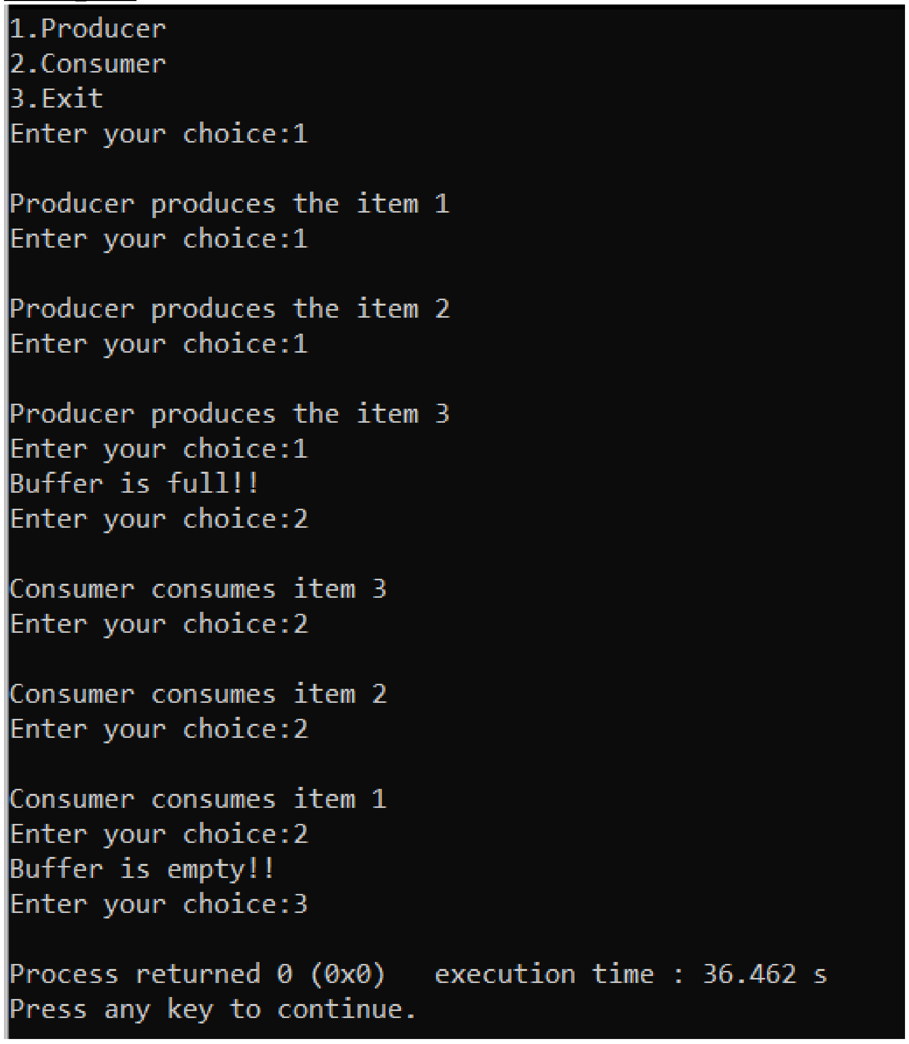
<u>Source Code</u>:

```
#include<stdio.h>
#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()
{
   int n;
   void producer ();
   void consumer ();
   int wait(int);
   int signal(int);
   printf("\n1.Producer\n2.Consumer\n3.Exit");
   while(1)
   {
      printf("\nEnter your choice:");
      scanf("%d",&n);
      switch(n)
      {
         case 1:   if((mutex==1)&&(empty!=0))
                      producer();
                   else
                      printf("Buffer is full!!");
                   break;
         case 2:   if((mutex==1)&&(full!=0))
                      consumer();
                   else
```

```
                    printf("Buffer is empty!!");
                break;
            case 3:
                exit(0);
                break;
        }
    }

    return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}

void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}
```

```
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}
```

Output:



```
1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3

Process returned 0 (0x0)   execution time : 36.462 s
Press any key to continue.
```

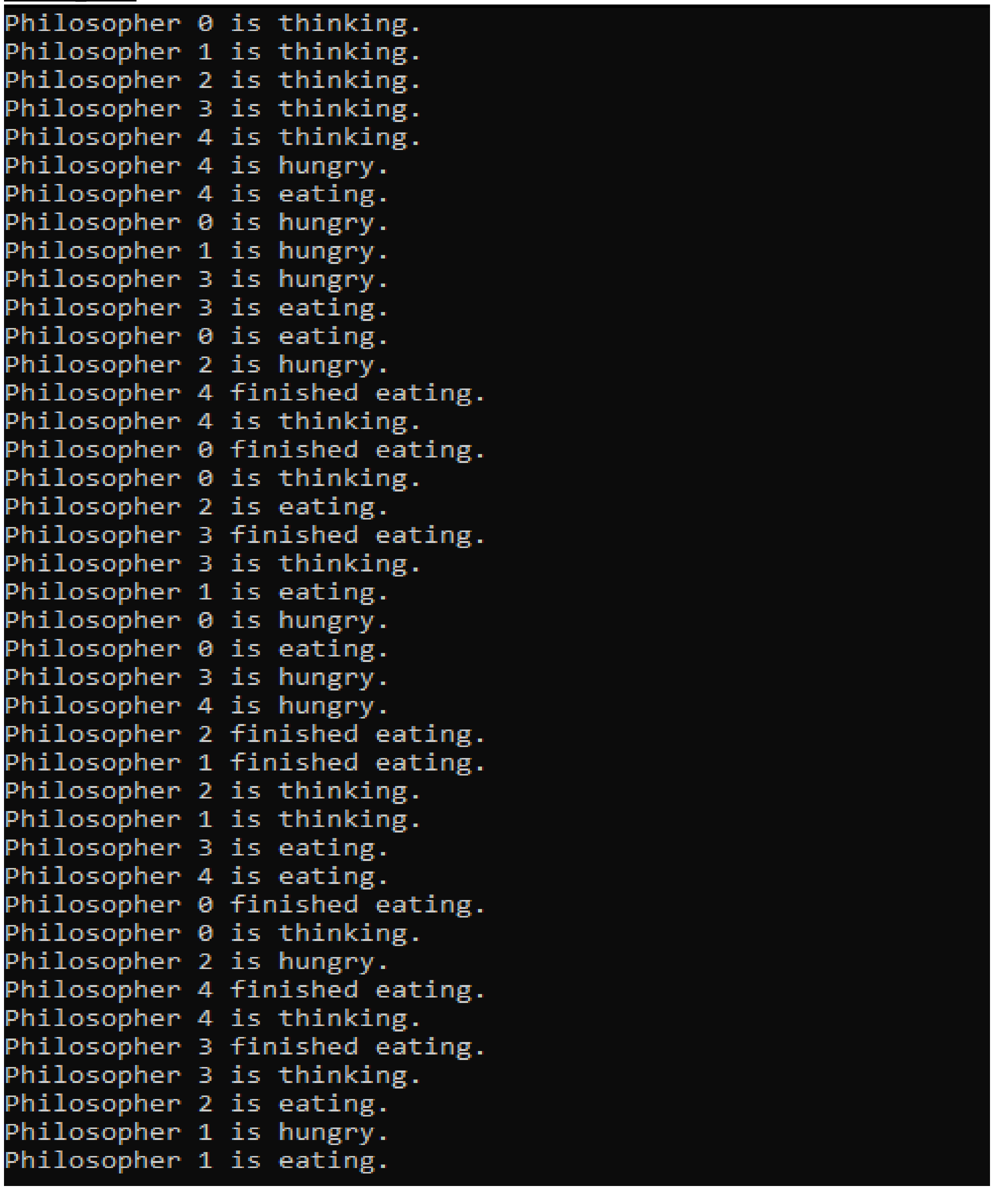
<u>Lab Program 5</u>:

Write a C program to simulate the concept of Dining-Philosophers problem.

<u>Source Code</u>:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include<conio.h>

#define NUM_PHILOSOPHERS 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT (philosopher_num + NUM_PHILOSOPHERS - 1) %
NUM_PHILOSOPHERS
#define RIGHT (philosopher_num + 1) % NUM_PHILOSOPHERS

sem_t chopsticks[NUM_PHILOSOPHERS];
int philosopher_state[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int philosopher_num = *((int *)arg);

    while (1) {
        printf("Philosopher %d is thinking.\n", philosopher_num);
        sleep(rand() % 3 + 1);
        printf("Philosopher %d is hungry.\n", philosopher_num);
        philosopher_state[philosopher_num] = HUNGRY;
        if (philosopher_num == 0) {
            sem_post(&chopsticks[RIGHT]);
            sem_post(&chopsticks[LEFT]);
        } else {
            sem_wait(&chopsticks[LEFT]);
            sem_wait(&chopsticks[RIGHT]);
        }

        printf("Philosopher %d is eating.\n", philosopher_num);
        sleep(rand() % 3 + 1);

        printf("Philosopher %d finished eating.\n", philosopher_num);
        philosopher_state[philosopher_num] = THINKING;
        sem_post(&chopsticks[LEFT]);
        sem_post(&chopsticks[RIGHT]);
    }
}

void main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_nums[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        sem_init(&chopsticks[i], 0, 1);
        philosopher_state[i] = THINKING;
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        philosopher_nums[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&philosopher_nums[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_join(philosophers[i], NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        sem_destroy(&chopsticks[i]);
    }

    getch();
}
```

Output:



```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 4 is hungry.
Philosopher 4 is eating.
Philosopher 0 is hungry.
Philosopher 1 is hungry.
Philosopher 3 is hungry.
Philosopher 3 is eating.
Philosopher 0 is eating.
Philosopher 2 is hungry.
Philosopher 4 finished eating.
Philosopher 4 is thinking.
Philosopher 0 finished eating.
Philosopher 0 is thinking.
Philosopher 2 is eating.
Philosopher 3 finished eating.
Philosopher 3 is thinking.
Philosopher 1 is eating.
Philosopher 0 is hungry.
Philosopher 0 is eating.
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 2 finished eating.
Philosopher 1 finished eating.
Philosopher 2 is thinking.
Philosopher 1 is thinking.
Philosopher 3 is eating.
Philosopher 4 is eating.
Philosopher 0 finished eating.
Philosopher 0 is thinking.
Philosopher 2 is hungry.
Philosopher 4 finished eating.
Philosopher 4 is thinking.
Philosopher 3 finished eating.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 1 is hungry.
Philosopher 1 is eating.
```

Lab Program 7:

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

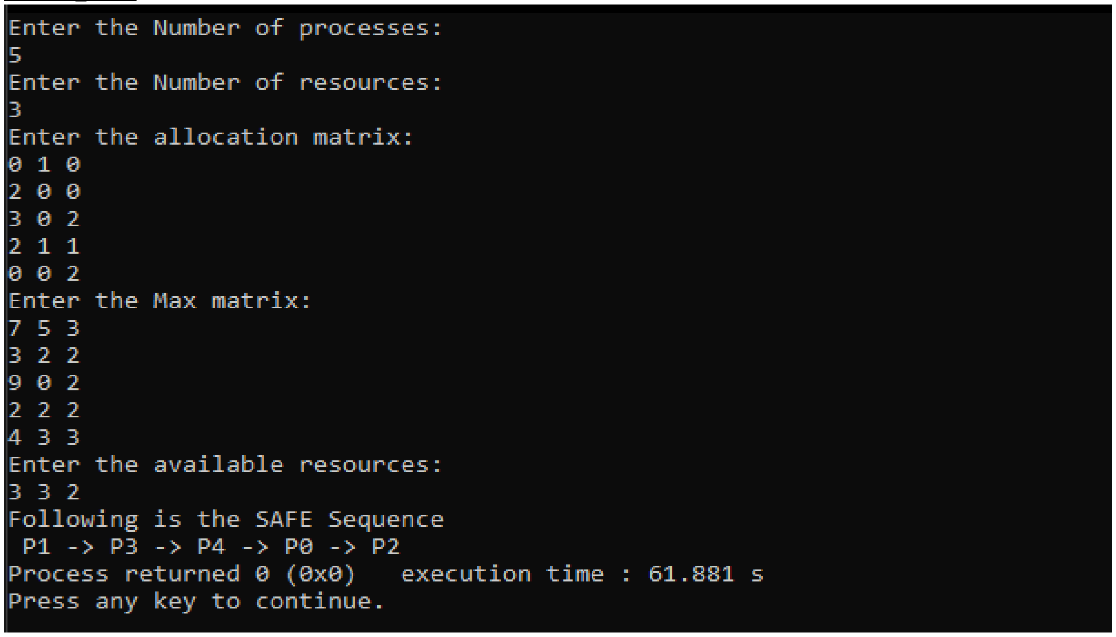
Source Code:

```
#include <stdio.h>
int main()
{
   int n, m, i, j, k;
   int alloc[50][50];
   int max[50][50];
   int avail[50];
   printf("Enter the Number of processes:\n");
   scanf("%d",&n);
   printf("Enter the Number of resources:\n");
   scanf("%d",&m);
   printf("Enter the allocation matrix:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&alloc[i][j]);
      }
   }
   printf("Enter the Max matrix:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&max[i][j]);
      }
   }
   printf("Enter the available resources:\n");
```

```
for(i=0;i<m;i++)
{
    scanf("%d",&avail[i]);
}
int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {
            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                     break;
                }
            }
            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
}
```

```
      int flag = 1;
      for(int i=0;i<n;i++)
    {
      if(f[i]==0)
      {
        flag=0;
         printf("The following system is not safe");
        break;
      }
    }
      if(flag==1)
    {
      printf("Following is the SAFE Sequence\n");
      for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
      printf(" P%d", ans[n-1]);
    }
return (0);
}
```

Output:



```
Enter the Number of processes:
5
Enter the Number of resources:
3
Enter the allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the Max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the available resources:
3 3 2
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
Process returned 0 (0x0)   execution time : 61.881 s
Press any key to continue.
```

<u>Lab Program 8</u>:

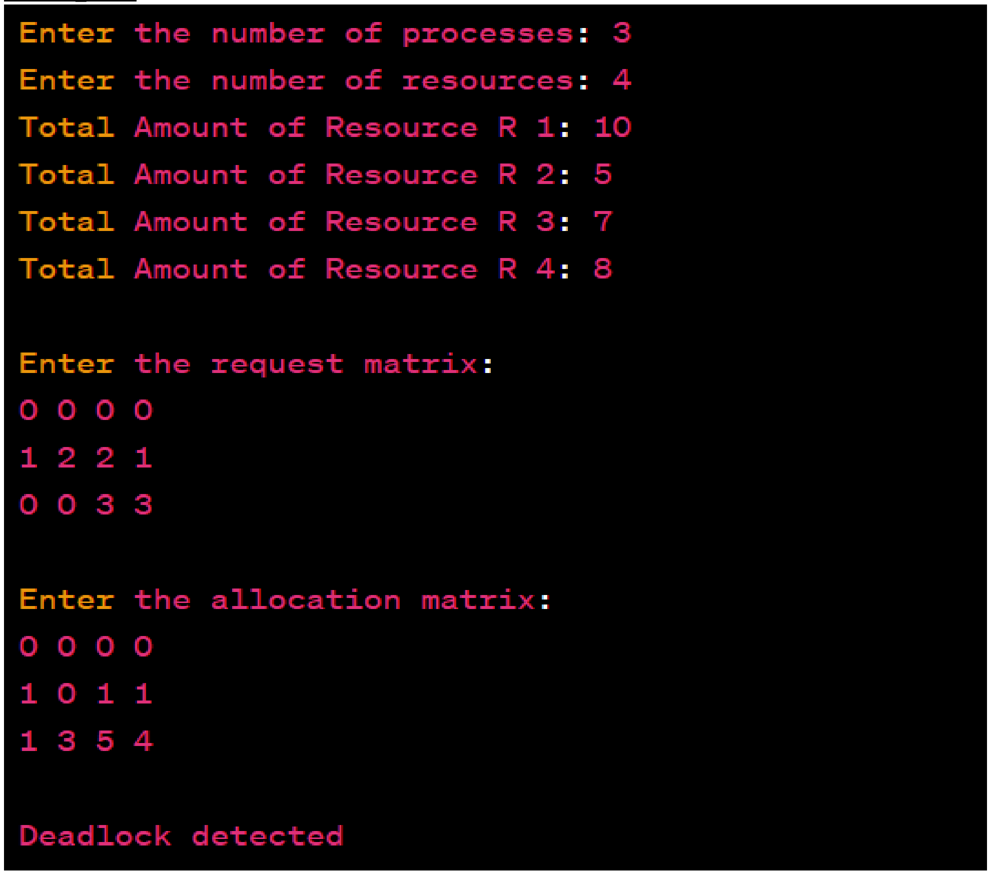
Write a C program to simulate deadlock detection

<u>Source Code</u>:

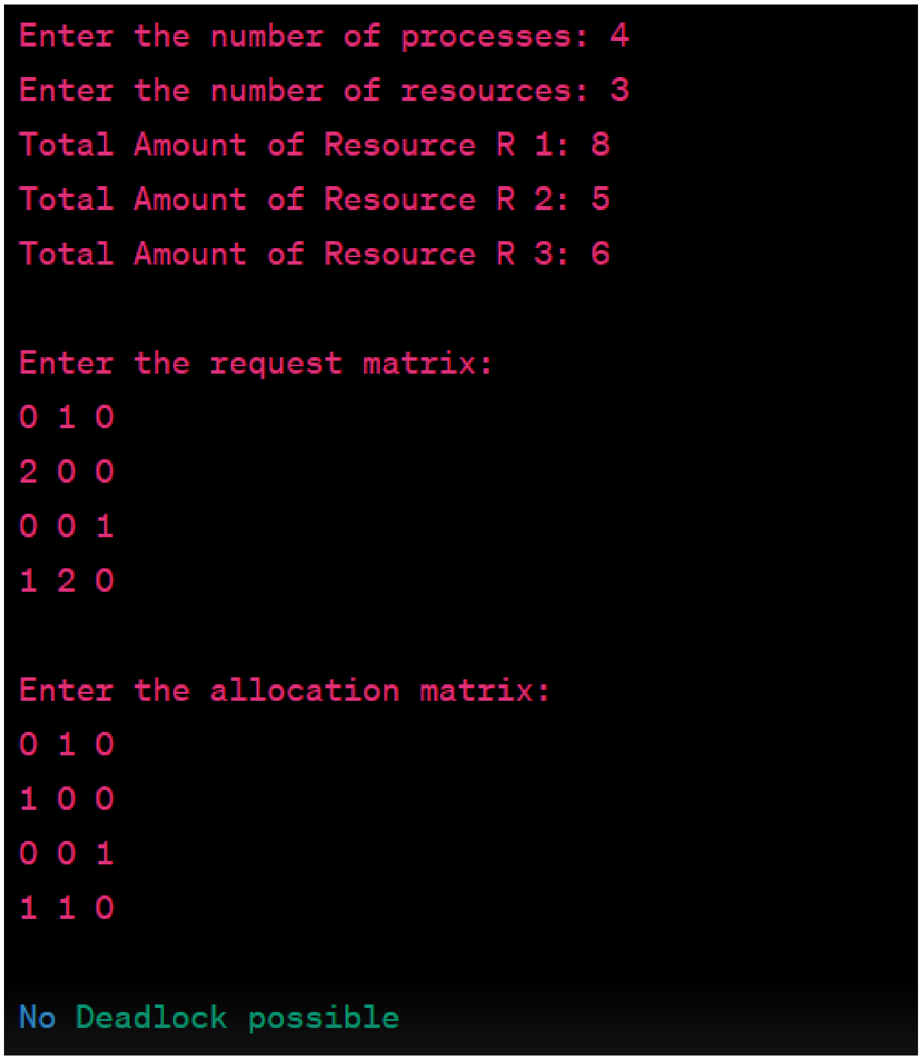
```
#include <stdio.h>
static int mark[20];
int i, j, np, nr;

int main() {
    int alloc[10][10], request[10][10], avail[10], r[10], w[10];

    printf("Enter the number of processes: ");
    scanf("%d", &np);
    printf("Enter the number of resources: ");
    scanf("%d", &nr);

    for (i = 0; i < nr; i++) {
        printf("Total Amount of Resource R %d: ", i + 1);
        scanf("%d", &r[i]);
    }

    printf("Enter the request matrix:\n");
    for (i = 0; i < np; i++)
        for (j = 0; j < nr; j++)
            scanf("%d", &request[i][j]);

    printf("Enter the allocation matrix:\n");
    for (i = 0; i < np; i++)
        for (j = 0; j < nr; j++)
            scanf("%d", &alloc[i][j]);

    // Calculate available resources
    for (j = 0; j < nr; j++) {
        avail[j] = r[j];
        for (i = 0; i < np; i++) {
            avail[j] -= alloc[i][j];
        }
    }

    // Mark processes with zero allocation
    for (i = 0; i < np; i++) {
        int count = 0;
        for (j = 0; j < nr; j++) {
            if (alloc[i][j] == 0)
                count++;
            else
                break;
        }
        if (count == nr)
            mark[i] = 1;
    }

    // Initialize W with avail
    for (j = 0; j < nr; j++)
        w[j] = avail[j];

    // Mark processes with request less than or equal to W
    for (i = 0; i < np; i++) {
        int canBeProcessed = 1;
        if (mark[i] != 1) {
            for (j = 0; j < nr; j++) {
                if (request[i][j] > w[j]) {
                    canBeProcessed = 0;
                    break;
                }
            }
            if (canBeProcessed) {
                mark[i] = 1;
                for (j = 0; j < nr; j++)
                    w[j] += alloc[i][j];
            }
        }
    }
```

```
    // Check for unmarked processes
    int deadlock = 0;
    for (i = 0; i < np; i++)
        if (mark[i] != 1)
            deadlock = 1;

    if (deadlock)
        printf("\nDeadlock detected\n");
    else
        printf("\nNo Deadlock possible\n");

    return 0;
}
```

Output:



```
Enter the number of processes: 3
Enter the number of resources: 4
Total Amount of Resource R 1: 10
Total Amount of Resource R 2: 5
Total Amount of Resource R 3: 7
Total Amount of Resource R 4: 8

Enter the request matrix:
0 0 0 0
1 2 2 1
0 0 3 3

Enter the allocation matrix:
0 0 0 0
1 0 1 1
1 3 5 4

Deadlock detected
```



```
Enter the number of processes: 4
Enter the number of resources: 3
Total Amount of Resource R 1: 8
Total Amount of Resource R 2: 5
Total Amount of Resource R 3: 6

Enter the request matrix:
0 1 0
2 0 0
0 0 1
1 2 0

Enter the allocation matrix:
0 1 0
1 0 0
0 0 1
1 1 0

No Deadlock possible
```

<u>Lab Program 9</u>:

Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit
b) Best-fit
c) First-fit

<u>Source Code</u>:

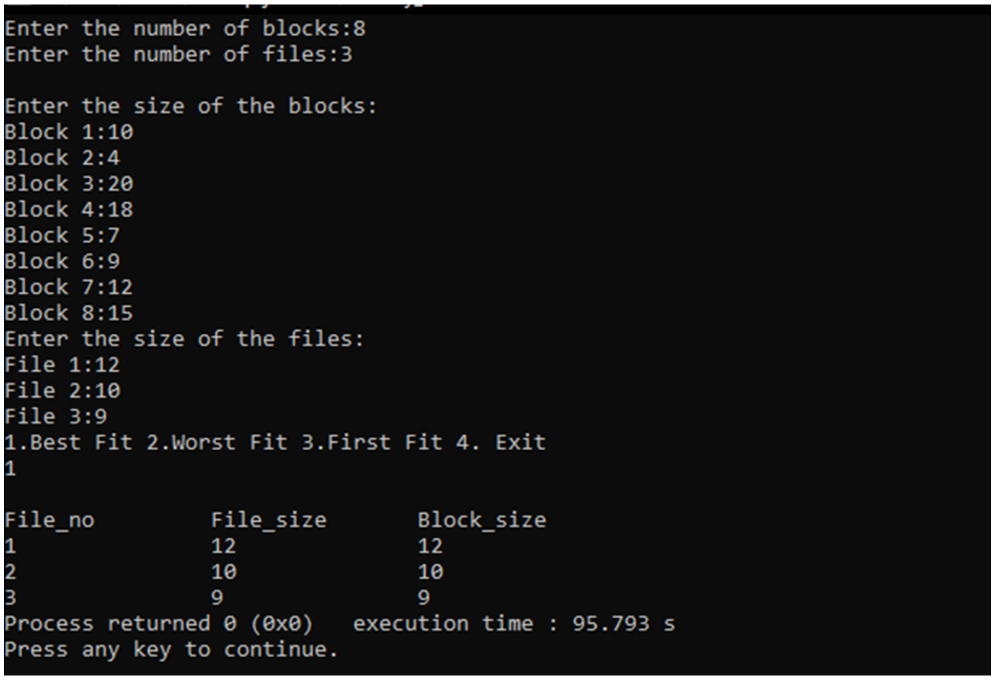
```
//menu driven code for all three algorithms
#include <stdio.h>
#include<stdlib.h>
#define max 25

void readInput(int *nb, int *nf, int b[], int f[]);
void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void displayResults(int nf, int f[], int b[], int ff[]);

int main()
{
    int nb, nf, ch;
    int b[max], f[max], bf[max] = {0}, ff[max] = {0}, frag[max] = {0};
    readInput(&nb, &nf, b, f);
    printf("1.Best Fit 2.Worst Fit 3.First Fit 4. Exit\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1: bestFit(nb, nf, b, f, bf, ff, frag);
                break;
        case 2: worstFit(nb, nf, b, f, bf, ff, frag);
                break;
        case 3: firstFit(nb, nf, b, f, bf, ff, frag);
```

```
                break;
            case 4: exit(0);
                break;
            default: printf("Inavlid choice\n");
                 break;
        }
        displayResults(nf, f, b,  ff);
        return 0;
}

void readInput(int *nb, int *nf, int b[], int f[])
{
    int i;
    printf("Enter the number of blocks:");
    scanf("%d", nb);

    printf("Enter the number of files:");
    scanf("%d", nf);

    printf("\nEnter the size of the blocks:\n");
    for (i = 1; i <= *nb; i++)
    {
        printf("Block %d:", i);
        scanf("%d", &b[i]);
    }

    printf("Enter the size of the files:\n");
    for (i = 1; i <= *nf; i++)
    {
        printf("File %d:", i);
        scanf("%d", &f[i]);
    }
}
```

```
void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
{
    int i, j, temp, lowest = 10000;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1) //if bf[j] is not allocated
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    if(lowest > temp)
                    {
                        ff[i] = j;
                        lowest = temp;
                    }
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }
}

void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
{
    int i, j, temp, lowest = 10000;
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    if (lowest == 10000 || temp > lowest)
                    {
                        ff[i] = j;
                        lowest = temp;
                    }
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }
}

void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
{
    int i, j, temp;
    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    ff[i] = j;
                    break;
                }
            }
        }
        frag[i] = temp;
        bf[ff[i]] = 1;
    }
}
void displayResults(int nf, int f[], int b[], int ff[])
{
    int i;
    printf("\nFile_no\t\tFile_size\tBlock_size");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d", i, f[i], b[ff[i]]);
    }
}
```

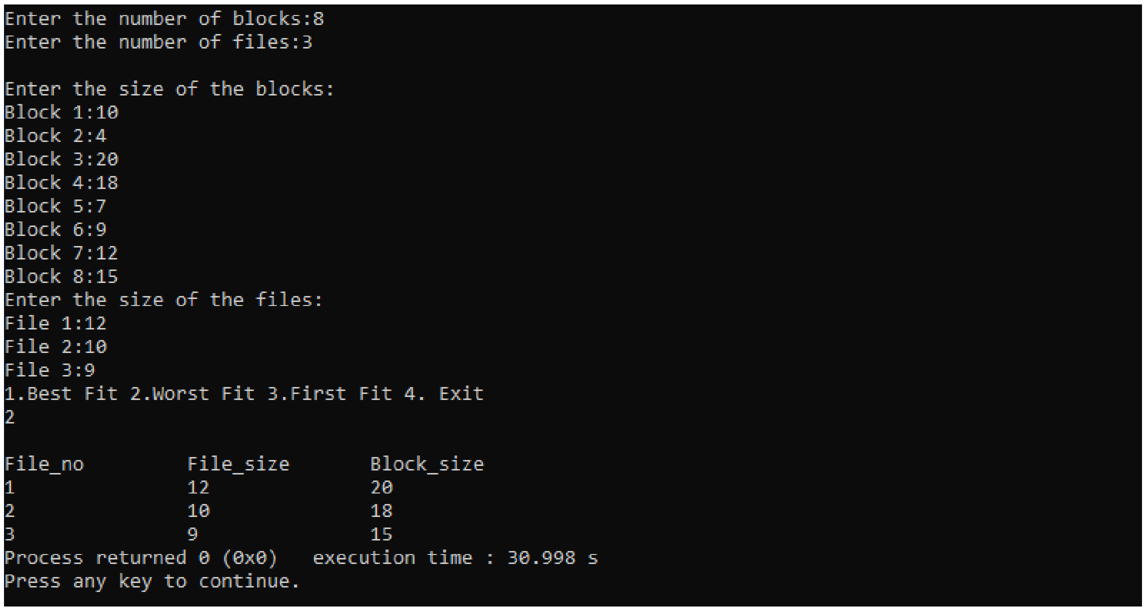
<u>Output</u>:

Best Fit:



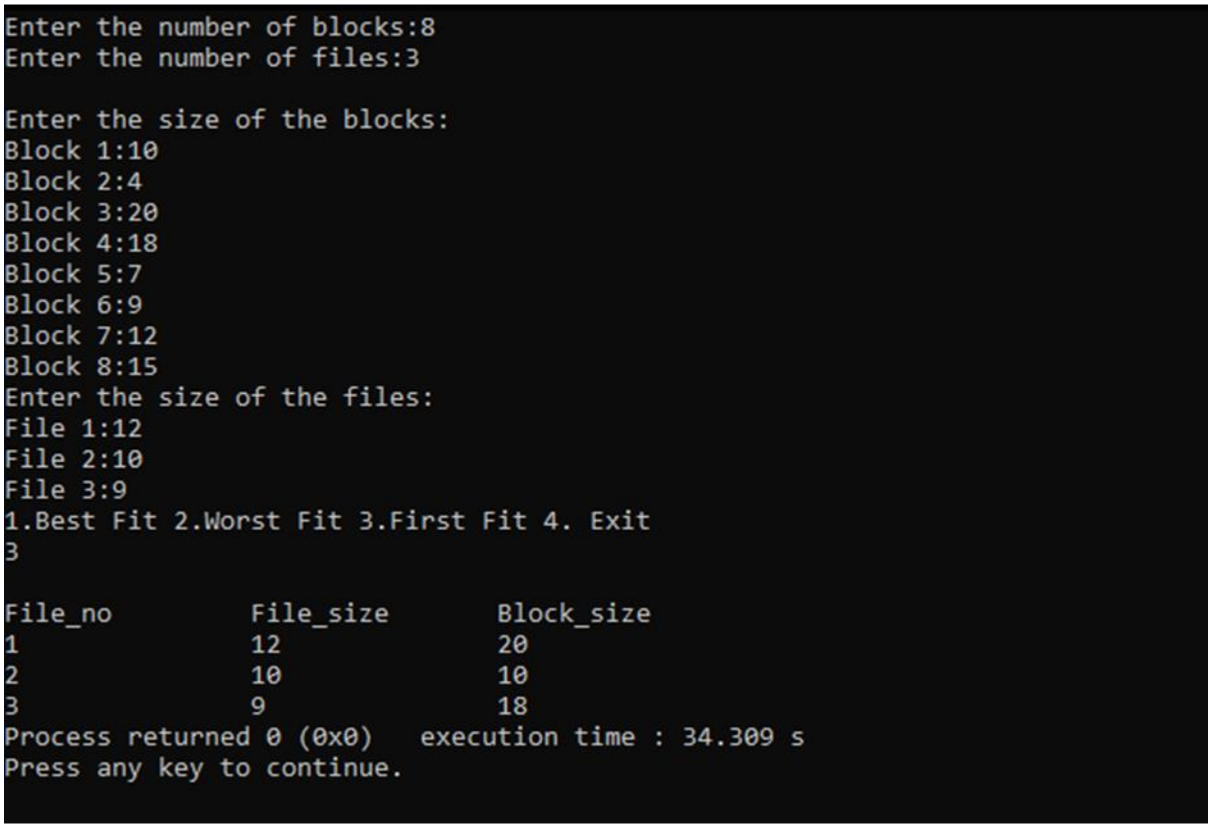
```
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
1

File_no         File_size       Block_size
1               12              12
2               10              10
3               9               9
Process returned 0 (0x0)   execution time : 95.793 s
Press any key to continue.
```

Worst Fit:



```
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
2

File_no         File_size       Block_size
1               12              20
2               10              18
3               9               15
Process returned 0 (0x0)   execution time : 30.998 s
Press any key to continue.
```

First Fit:



```
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
3

File_no         File_size       Block_size
1               12              20
2               10              10
3               9               18
Process returned 0 (0x0)   execution time : 34.309 s
Press any key to continue.
```

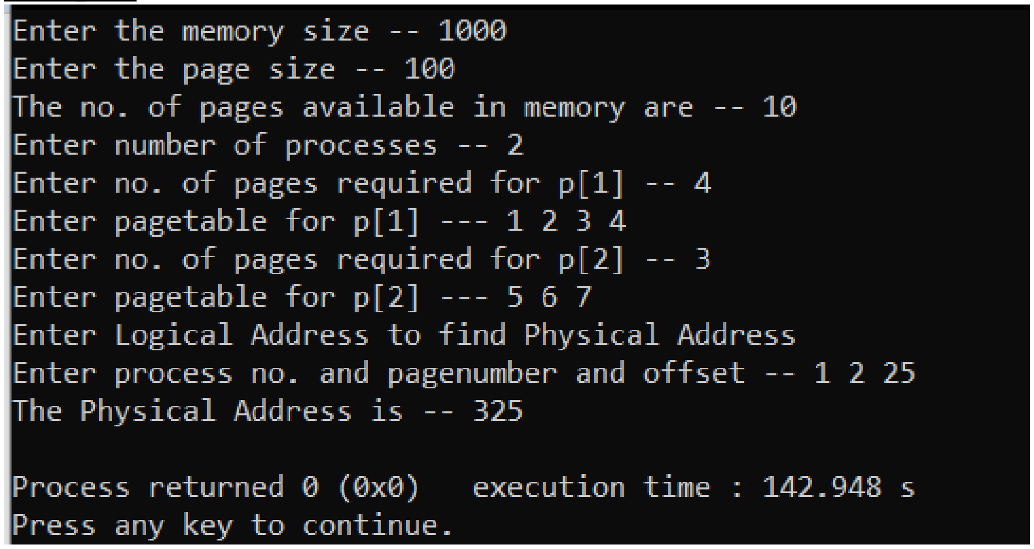
Lab Program 10:

Write a C program to simulate paging technique of memory management.

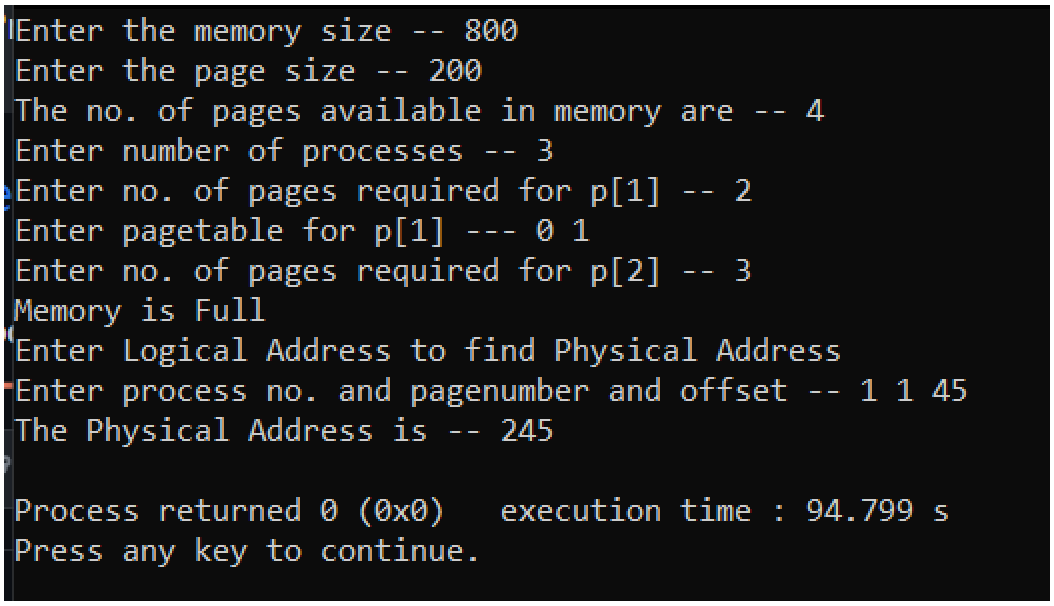
Source Code:

```
#include <stdio.h>
#include <conio.h>

int main() {
    int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
    int s[10], fno[10][20];

    clrscr();

    printf("Enter the memory size -- ");
    scanf("%d", &ms);

    printf("Enter the page size -- ");
    scanf("%d", &ps);

    nop = ms / ps;
    printf("The no. of pages available in memory are -- %d\n", nop);

    printf("Enter number of processes -- ");
    scanf("%d", &np);
    rempages = nop;

    for (i = 1; i <= np; i++) {
        printf("Enter no. of pages required for p[%d] -- ", i);
        scanf("%d", &s[i]);

        if (s[i] > rempages) {
            printf("Memory is Full\n");
            break;
        }

        rempages = rempages - s[i];

        printf("Enter pagetable for p[%d] --- ", i);
        for (j = 0; j < s[i]; j++)
            scanf("%d", &fno[i][j]);
    }

    printf("Enter Logical Address to find Physical Address\n");
    printf("Enter process no. and pagenumber and offset -- ");
    scanf("%d %d %d", &x, &y, &offset);

    if (x > np || y >= s[i] || offset >= ps)
        printf("Invalid Process or Page Number or offset\n");
    else {
        pa = fno[x][y] * ps + offset;
        printf("The Physical Address is -- %d\n", pa);
    }
    getch();
    return 0;
}
```

Output:



```
Enter the memory size -- 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 2
Enter no. of pages required for p[1] -- 4
Enter pagetable for p[1] --- 1 2 3 4
Enter no. of pages required for p[2] -- 3
Enter pagetable for p[2] --- 5 6 7
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 1 2 25
The Physical Address is -- 325

Process returned 0 (0x0)   execution time : 142.948 s
Press any key to continue.
```



```
Enter the memory size -- 800
Enter the page size -- 200
The no. of pages available in memory are -- 4
Enter number of processes -- 3
Enter no. of pages required for p[1] -- 2
Enter pagetable for p[1] --- 0 1
Enter no. of pages required for p[2] -- 3
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 1 1 45
The Physical Address is -- 245

Process returned 0 (0x0)   execution time : 94.799 s
Press any key to continue.
```

Lab Program 11:

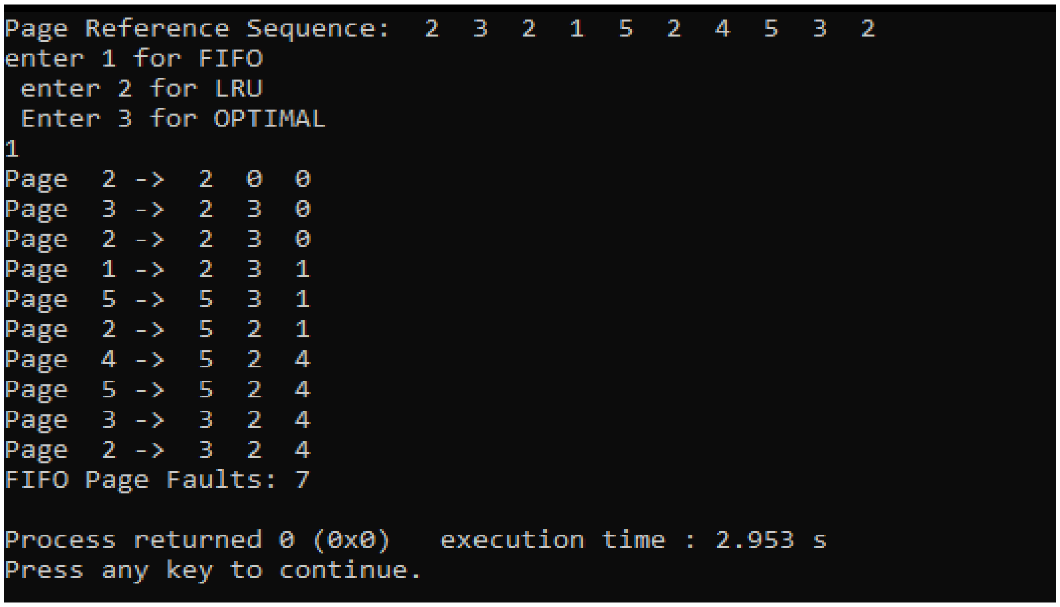
Write a C program to simulate page replacement algorithms
a) FIFO
b) LRU
c) Optimal

Source Code:

```
#include <stdio.h>
#define NUM_FRAMES 3
#define NUM_PAGES 10

void printFrames(int frames[]) {
    for (int i = 0; i < NUM_FRAMES; i++)
        printf("%2d ", frames[i]);
    printf("\n");
}

int findIndex(int arr[], int n, int element) {
    for (int i = 0; i < n; i++)
        if (arr[i] == element)
            return i;
    return -1;
}

void fifo(int pages[]) {
    int frames[NUM_FRAMES] = {0};
    int frameIndex = 0, pageFaults = 0;

    for (int i = 0; i < NUM_PAGES; i++) {
        int page = pages[i];
        if (findIndex(frames, NUM_FRAMES, page) == -1) {
            frames[frameIndex] = page;
            frameIndex = (frameIndex + 1) % NUM_FRAMES;
            pageFaults++;
        }
        printf("Page %2d -> ", page);
        printFrames(frames);
    }

    printf("FIFO Page Faults: %d\n", pageFaults);
}

void lru(int pages[]) {
    int frames[NUM_FRAMES] = {0};
    int pageFaults = 0;

    for (int i = 0; i < NUM_PAGES; i++) {
        int page = pages[i];
        int index = findIndex(frames, NUM_FRAMES, page);
        if (index == -1) {
            for (int j = 0; j < NUM_FRAMES; j++)
                if (frames[j] == 0 || findIndex(pages, i, frames[j]) == -1) {
                    frames[j] = page;
                    break;
                }
            pageFaults++;
        }
        printf("Page %2d -> ", page);
        printFrames(frames);
    }

    printf("LRU Page Faults: %d\n", pageFaults);
}

void optimal(int pages[]) {
    int frames[NUM_FRAMES] = {0};
    int pageFaults = 0;

    for (int i = 0; i < NUM_PAGES; i++) {
        int page = pages[i];
        int index = findIndex(frames, NUM_FRAMES, page);
        if (index == -1) {
            int optimalIndex = -1;
            for (int j = 0; j < NUM_FRAMES; j++) {
                int pageIndex = findIndex(pages, NUM_PAGES, frames[j]);
                if (pageIndex == -1) {
                    optimalIndex = j;
                    break;
                }
                if (optimalIndex == -1 || pageIndex > findIndex(pages,
NUM_PAGES, frames[optimalIndex]))
                    optimalIndex = j;
            }
            frames[optimalIndex] = page;
            pageFaults++;
        }
        printf("Page %2d -> ", page);
        printFrames(frames);
    }

    printf("Optimal Page Faults: %d\n", pageFaults);
}

int main() {
    int pages[NUM_PAGES] = {2, 3, 2, 1, 5, 2, 4, 5, 3, 2};

    printf("Page Reference Sequence: ");
    for (int i = 0; i < NUM_PAGES; i++)
        printf("%2d ", pages[i]);
    printf("\n");

    switch(ch)
    {
        case 1:
        fifo(pages); break;
        case 2:
        lru(pages); break;
        case 3:
        optimal(pages); break;
        case 4: exit(0); break;
        default: printf("Invalid\n"); break;
    }
    return 0;
}
```

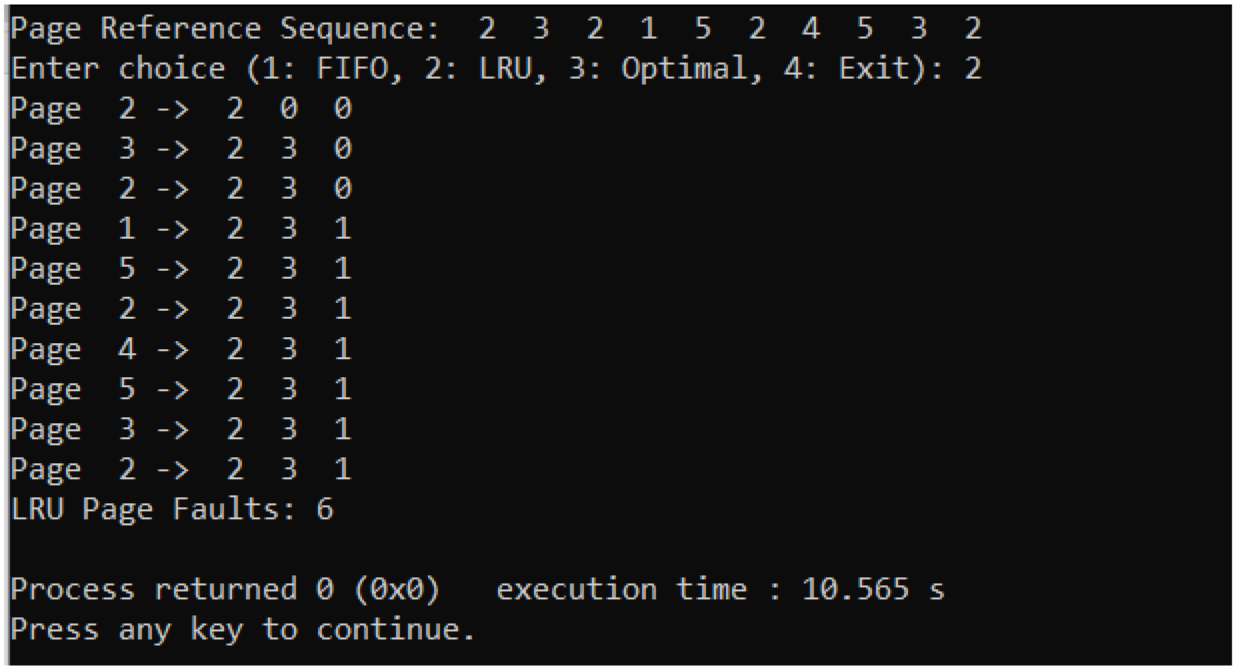
<u>Output</u>:

FIFO:



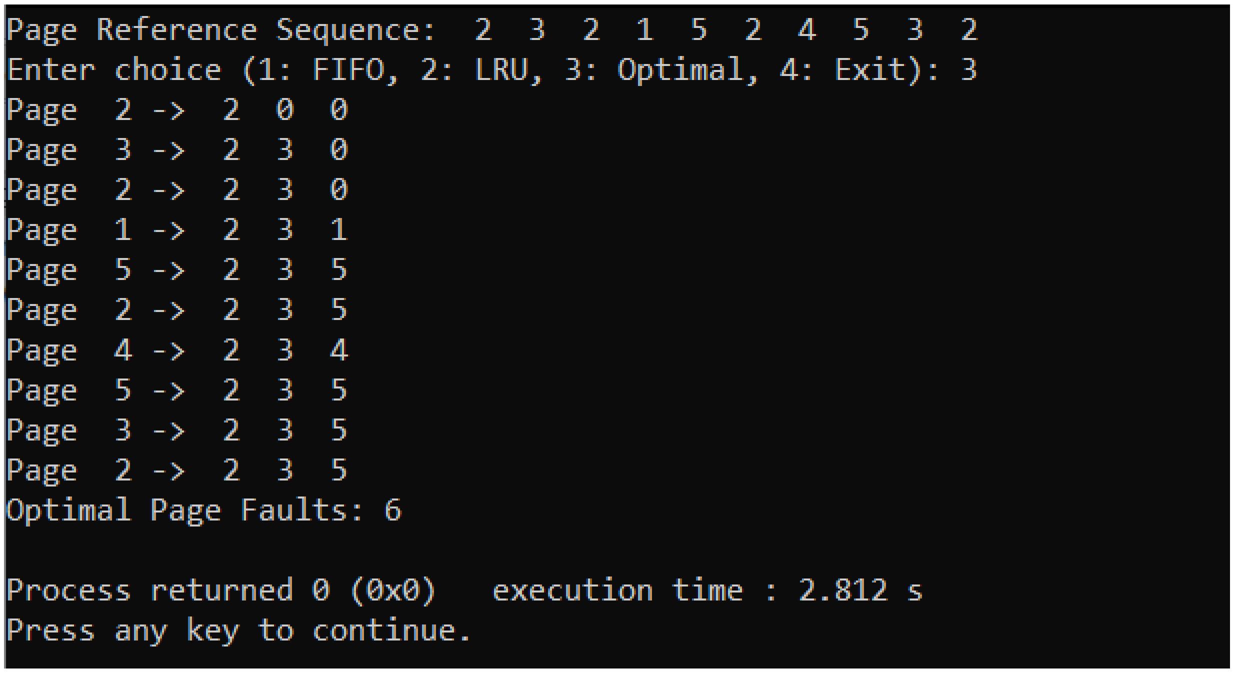
```
Page Reference Sequence:  2  3  2  1  5  2  4  5  3  2
enter 1 for FIFO
 enter 2 for LRU
 Enter 3 for OPTIMAL
1
Page  2 ->  2  0  0
Page  3 ->  2  3  0
Page  2 ->  2  3  0
Page  1 ->  2  3  1
Page  5 ->  5  3  1
Page  2 ->  5  2  1
Page  4 ->  5  2  4
Page  5 ->  5  2  4
Page  3 ->  3  2  4
Page  2 ->  3  2  4
FIFO Page Faults: 7

Process returned 0 (0x0)   execution time : 2.953 s
Press any key to continue.
```

LRU:



```
Page Reference Sequence:  2  3  2  1  5  2  4  5  3  2
Enter choice (1: FIFO, 2: LRU, 3: Optimal, 4: Exit): 2
Page  2 ->  2  0  0
Page  3 ->  2  3  0
Page  2 ->  2  3  0
Page  1 ->  2  3  1
Page  5 ->  2  3  1
Page  2 ->  2  3  1
Page  4 ->  2  3  1
Page  5 ->  2  3  1
Page  3 ->  2  3  1
Page  2 ->  2  3  1
LRU Page Faults: 6

Process returned 0 (0x0)   execution time : 10.565 s
Press any key to continue.
```

OPTIMAL:



```
Page Reference Sequence:  2  3  2  1  5  2  4  5  3  2
Enter choice (1: FIFO, 2: LRU, 3: Optimal, 4: Exit): 3
Page  2 ->  2  0  0
Page  3 ->  2  3  0
Page  2 ->  2  3  0
Page  1 ->  2  3  1
Page  5 ->  2  3  5
Page  2 ->  2  3  5
Page  4 ->  2  3  4
Page  5 ->  2  3  5
Page  3 ->  2  3  5
Page  2 ->  2  3  5
Optimal Page Faults: 6

Process returned 0 (0x0)   execution time : 2.812 s
Press any key to continue.
```

Lab Program 12:

Write a C program to simulate disk scheduling algorithms

a) FCFS

b) SCAN

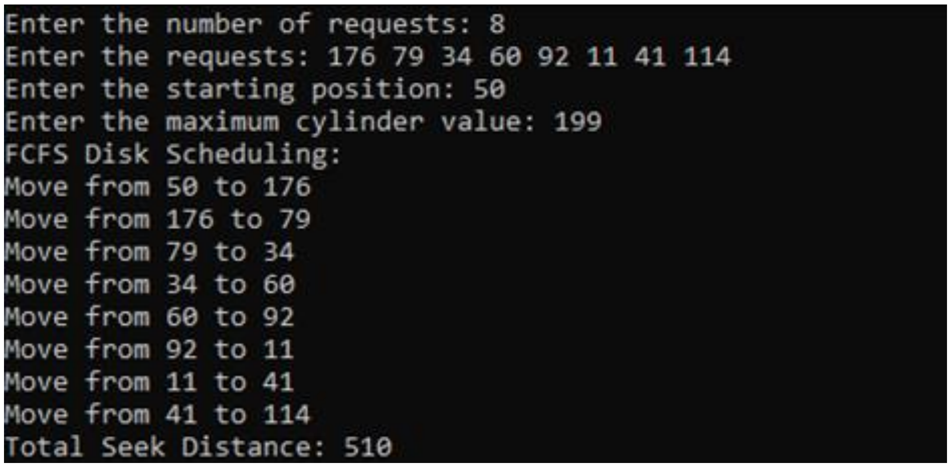
c) C-SCAN

Source Code:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_REQUESTS 100

void fcfs(int requests[], int n, int start) {
    int totalSeek = 0, current = start;

    printf("FCFS Disk Scheduling:\n");

    for (int i = 0; i < n; i++) {
        totalSeek += abs(current - requests[i]);
        printf("Move from %d to %d\n", current, requests[i]);
        current = requests[i];
    }

    printf("Total Seek Distance: %d\n", totalSeek);
}

void scan(int requests[], int n, int start, int maxCylinder) {
    int totalSeek = 0, current = start;

    printf("SCAN Disk Scheduling:\n");

    int direction = 1; // 1 for right, -1 for left
    int maxIndex = (direction == 1) ? maxCylinder : 0;
    for (int i = 0; i < n; i++) {
        totalSeek += abs(current - requests[i]);
        printf("Move from %d to %d\n", current, requests[i]);
        current = requests[i];
    }

    totalSeek += abs(current - maxIndex);
    printf("Move from %d to %d\n", current, maxIndex);

    for (int i = n - 1; i >= 0; i--) {
        totalSeek += abs(maxIndex - requests[i]);
        printf("Move from %d to %d\n", maxIndex, requests[i]);
        maxIndex = requests[i];
    }

    printf("Total Seek Distance: %d\n", totalSeek);
}

void cScan(int requests[], int n, int start, int maxCylinder) {
    int totalSeek = 0, current = start;

    printf("C-SCAN Disk Scheduling:\n");

    int maxIndex = maxCylinder;

    for (int i = 0; i < n; i++) {
        totalSeek += abs(current - requests[i]);
        printf("Move from %d to %d\n", current, requests[i]);
        current = requests[i];
    }

    totalSeek += abs(current - maxIndex);
    printf("Move from %d to %d\n", current, maxIndex);

    current = 0;
    for (int i = 0; i < n; i++) {
        totalSeek += abs(current - requests[i]);
        printf("Move from %d to %d\n", current, requests[i]);
        current = requests[i];
    }

    printf("Total Seek Distance: %d\n", totalSeek);
}

int main() {
    int requests[MAX_REQUESTS], n, start, maxCylinder;

    printf("Enter the number of requests: ");
    scanf("%d", &n);

    if (n > MAX_REQUESTS) {
        printf("Maximum number of requests exceeded.\n");
        return 1;
    }

    printf("Enter the requests: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &requests[i]);

    printf("Enter the starting position: ");
    scanf("%d", &start);

    printf("Enter the maximum cylinder value: ");
    scanf("%d", &maxCylinder);
    fcfs(requests, n, start);
    scan(requests, n, start, maxCylinder);
    cScan(requests, n, start, maxCylinder);

    return 0;
}
```

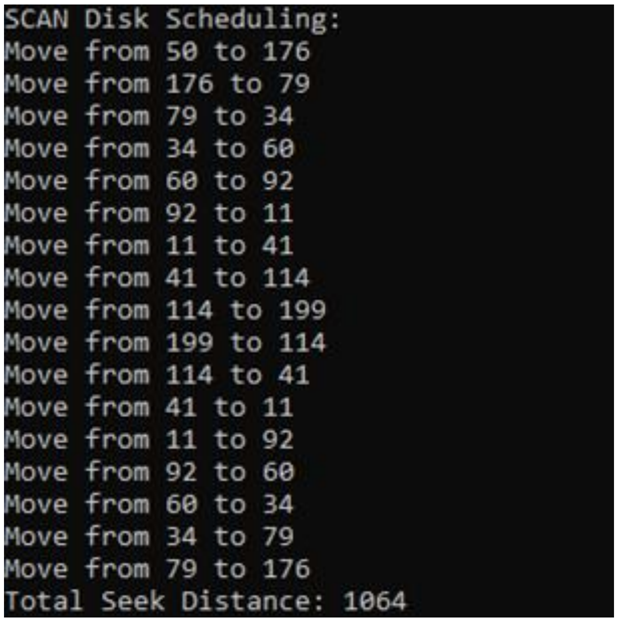
Output:

FCFS:



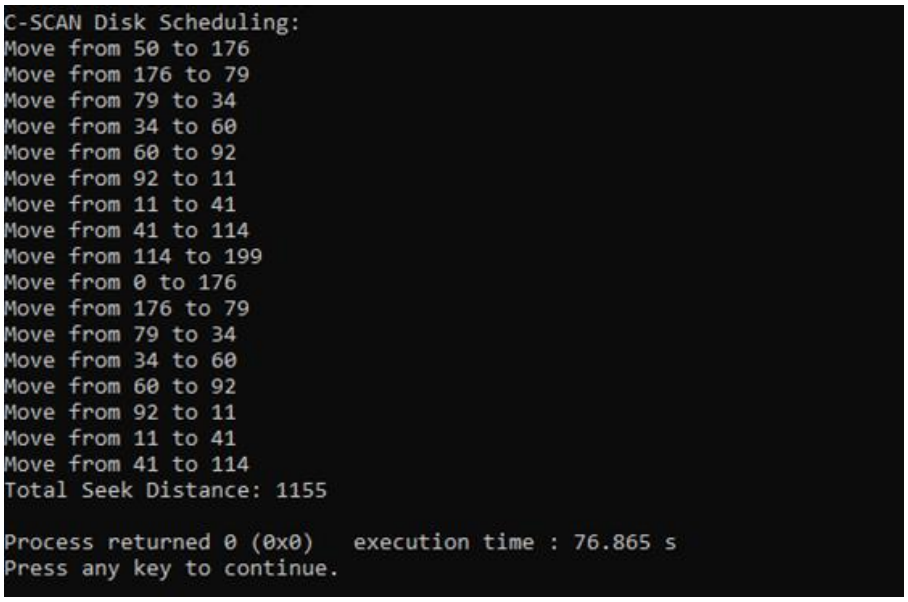
```
Enter the number of requests: 8
Enter the requests: 176 79 34 60 92 11 41 114
Enter the starting position: 50
Enter the maximum cylinder value: 199
FCFS Disk Scheduling:
Move from 50 to 176
Move from 176 to 79
Move from 79 to 34
Move from 34 to 60
Move from 60 to 92
Move from 92 to 11
Move from 11 to 41
Move from 41 to 114
Total Seek Distance: 510
```

SCAN:



```
SCAN Disk Scheduling:
Move from 50 to 176
Move from 176 to 79
Move from 79 to 34
Move from 34 to 60
Move from 60 to 92
Move from 92 to 11
Move from 11 to 41
Move from 41 to 114
Move from 114 to 199
Move from 199 to 114
Move from 114 to 41
Move from 41 to 11
Move from 11 to 92
Move from 92 to 60
Move from 60 to 34
Move from 34 to 79
Move from 79 to 176
Total Seek Distance: 1064
```

C-SCAN:



```
C-SCAN Disk Scheduling:
Move from 50 to 176
Move from 176 to 79
Move from 79 to 34
Move from 34 to 60
Move from 60 to 92
Move from 92 to 11
Move from 11 to 41
Move from 41 to 114
Move from 114 to 199
Move from 0 to 176
Move from 176 to 79
Move from 79 to 34
Move from 34 to 60
Move from 60 to 92
Move from 92 to 11
Move from 11 to 41
Move from 41 to 114
Total Seek Distance: 1155

Process returned 0 (0x0)   execution time : 76.865 s
Press any key to continue.
```

<u>Lab Program 13</u>:

Write a C program to simulate disk scheduling algorithms
a) SSTF
b) LOOK
c) c-LOOK

<u>Source Code</u>:

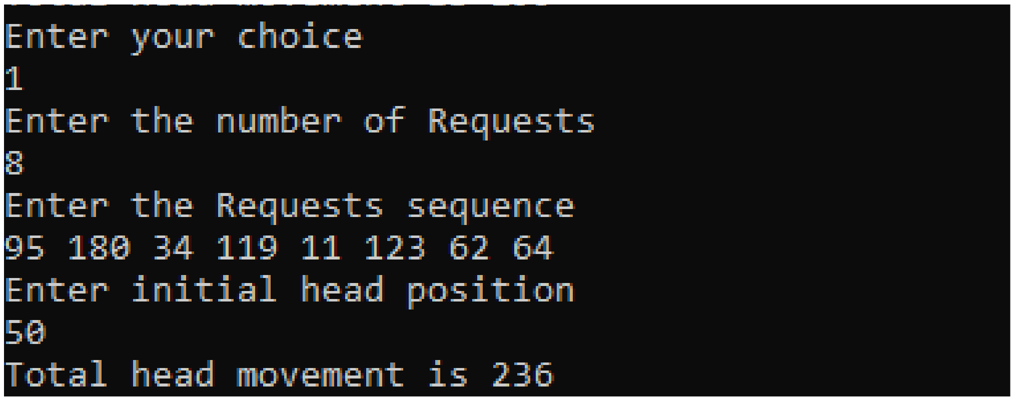
```
#include<stdio.h>
#include<stdlib.h>

void SSTF() {
    int RQ[100], i, n, TotalHeadMoment = 0, initial, count = 0;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);

    // logic for SSTF disk scheduling

    /* loop will execute until all processes are completed */
    while (count != n) {
        int min = 1000, d, index;
        for (i = 0; i < n; i++) {
            d = abs(RQ[i] - initial);
            if (min > d) {
                min = d;
                index = i;
            }
        }
        TotalHeadMoment = TotalHeadMoment + min;
        initial = RQ[index];
        // 1000 is used as a placeholder for completed requests
        RQ[index] = 1000;
        count++;
    }
    printf("Total head movement is %d\n", TotalHeadMoment);
}

void C_LOOK() {
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, move;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);
    printf("Enter the head movement direction for high (1) and for low
(0)\n");
    scanf("%d", &move);

    // logic for C-LOOK disk scheduling

    /* logic for sorting the request array */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (RQ[j] > RQ[j + 1]) {
                int temp;
                temp = RQ[j];
                RQ[j] = RQ[j + 1];
                RQ[j + 1] = temp;
            }
        }
```

```
}

int index;
for (i = 0; i < n; i++) {
    if (initial < RQ[i]) {
        index = i;
        break;
    }
}

// if movement is towards high value
if (move == 1) {
    for (i = index; i < n; i++) {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    for (i = 0; i < index; i++) {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}
// if movement is towards low value
else {
    for (i = index - 1; i >= 0; i--) {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    for (i = n - 1; i >= index; i--) {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}
```

```
    printf("Total head movement is %d\n", TotalHeadMoment);
}

void LOOK() {
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, move;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);
    printf("Enter the head movement direction for high (1) and for low
(0)\n");
    scanf("%d", &move);

    // logic for LOOK disk scheduling

    /* logic for sorting the request array */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (RQ[j] > RQ[j + 1]) {
                int temp;
                temp = RQ[j];
                RQ[j] = RQ[j + 1];
                RQ[j + 1] = temp;
            }
        }
    }

    int index;
    for (i = 0; i < n; i++) {
        if (initial < RQ[i]) {
            index = i;
            break;
        }
    }

    // if movement is towards high value
    if (move == 1) {
        for (i = index; i < n; i++) {
            TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
            initial = RQ[i];
        }
        for (i = index - 1; i >= 0; i--) {
            TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
            initial = RQ[i];
        }
    }
    // if movement is towards low value
    else {
        for (i = index - 1; i >= 0; i--) {
            TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
            initial = RQ[i];
        }
        for (i = index; i < n; i++) {
            TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
            initial = RQ[i];
        }
    }

    printf("Total head movement is %d\n", TotalHeadMoment);
}

int main() {
    int ch;
    printf("\n 1.SSTF\t 2.LOOK\t 3.C-LOOK\t 4.EXIT\n");
    while (1) {
        printf("\nEnter your choice\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                SSTF();
                break;
            case 2:
                LOOK();
                break;
            case 3:
                C_LOOK();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
}
```

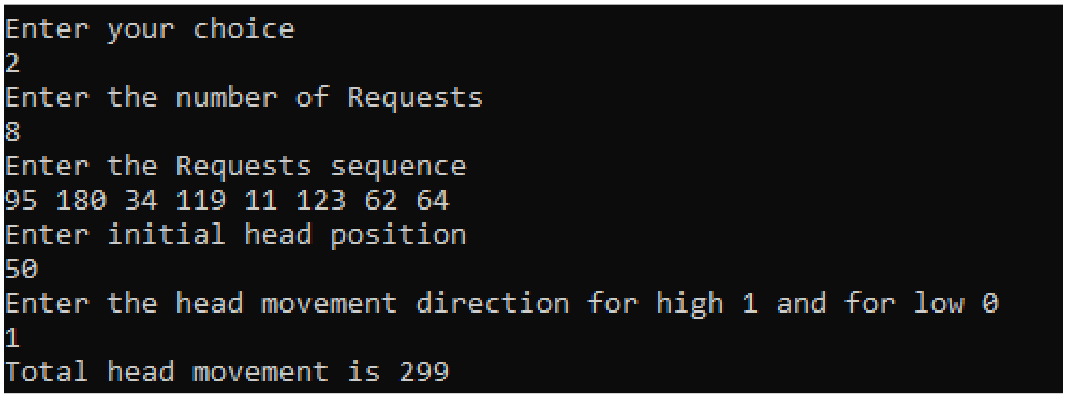
<u>Output</u>:

SSTF:



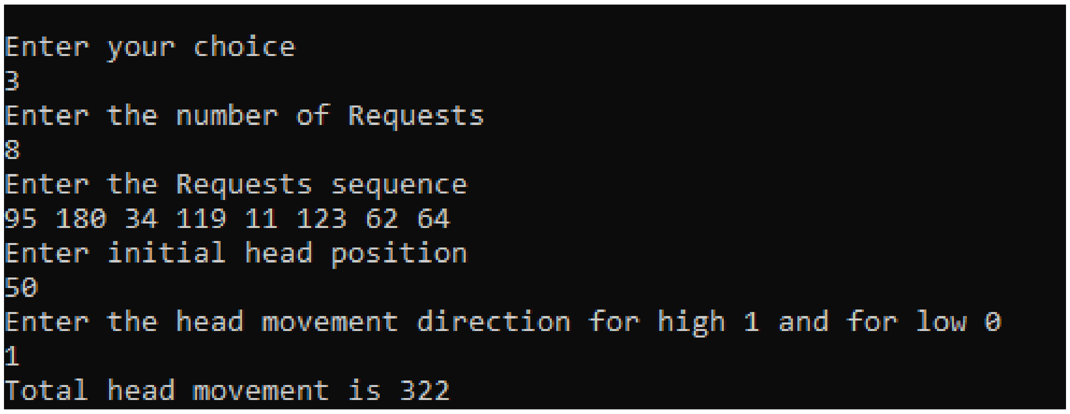
```
Enter your choice
1
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Total head movement is 236
```

LOOK:



```
Enter your choice
2
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 299
```

C-LOOK:



```
Enter your choice
3
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 322
```