

# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



## LAB REPORT

on

## OPERATING SYSTEMS

*Submitted by*

**Tejas S(1BM22CS308)**

*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**

**Apr-2024 to Aug-2024**

**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 – 23CS4PCOPS” carried out by **Tejas S (1BM22CS308)**, 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 year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

**Pramitha Pai**  
Assistant Professor  
Department of CSE  
BMSCE, Bengaluru

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

## Index Sheet

Sl. No.	Experiment Title	Page No.
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-preemptive)	1-7
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	8-14
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.	15-17
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling.	18-26
5.	Write a C program to simulate producer-consumer problem using semaphores..	27-39
6.	Write a C program to simulate the concept of Dining-Philosophers problem..	30-32
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance	33-36
8.	Write a C program to simulate deadlock detection	37-40
9.	Write a C program to simulate the following contiguous memory allocation techniques: a) Worst-fit b) Best-fit c) First-fit	41-44
10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal.	45-48

## 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-preemptive)

### FCFS Code:

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

int main(){
    int n;
    int process_id[n],at[n],bt[n],ct[n],tat[n],wt[n];
    printf("\nEnter number of processes: ");
    scanf("%d",&n);
    for(int i=0; i<n;i++){
        process_id[i] = i+1;
        printf("\nArrival Time for %d: ",(i+1));
        scanf("%d",&at[i]);
        printf("\nBurst Time for %d: ",(i+1));
        scanf("%d",&bt[i]);
    }
    int temp = 0;
    int temp2 = 0;
    for(int i = 0; i < n; i++){
        for(int j = i + 1; j < n; j++){
            if(at[i] > at[j]){
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                temp2 = bt[i];
                bt[i] = bt[j];
                bt[j] = temp2;
            }
        }
    }
}
```

```

int timePassed = 0;
for(int i = 0; i<n;i++){
    if(at[i] > timePassed){
        timePassed = timePassed + (at[i] - ct[i-1]);
    }
    timePassed += bt[i];
    ct[i] = timePassed;
}

for(int i = 0; i<n;i++){
    tat[i] = ct[i] = at[i];
    wt[i] = tat[i] = bt[i];
}
printf("\nPID\tAT\tBT\tCT\tTAT\tWT");

for(int i = 0; i<n;i++){
    printf("\n%d\t%d\t%d\t%d\t%d\t%d",process_id[i],at[i],bt[i],ct[i],tat[i],wt[i]);
}
}

```

## Result:

```

Enter the number of processes: 4
enter arrival time for process 1 to 4
0 1 3 5
enter the burst time for process 1 to 4
10 15 20 30
process_ID    AT    BT    CT    TAT    WT
1             0      10    10      10
2             1      15    25      24
3             3      20    45      42
2
4             5      30    75      70
0
avg turnaround time: 36.500000
avg waiting time: 17.750000

```

## SJF Non Pre-emptive Code:

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

```

#include <stdlib.h>
typedef struct {
    char process_name;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
} Process;

void sort_by_burst_time(Process *processes, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

void compute_completion_time(Process *processes, int n) {
    int current_time = 0;
    int index = 0;
    while (index < n) {
        int next_process = -1;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].completion_time == 0) {
                if (next_process == -1 || processes[i].burst_time < processes[next_process].burst_time)
                {
                    next_process = i;
                }
            }
        }
        if (next_process == -1) {
            current_time = processes[index].arrival_time;
        } else {
            processes[next_process].completion_time = current_time +
            processes[next_process].burst_time;
            current_time = processes[next_process].completion_time;
        }
    }
}

```

```

    }
    index++;
}
}
void compute_turnaround_waiting_time(Process *processes, int n) {
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
}
void display_table(Process *processes, int n) {
    printf("Process  Arrival Time  Burst Time  Completion Time  Turnaround Time  Waiting
Time\n");
    for (int i = 0; i < n; i++) {
        printf(" %c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_name,
                processes[i].arrival_time,
                processes[i].burst_time,
                processes[i].completion_time,
                processes[i].turnaround_time,
                processes[i].waiting_time);
    }
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process *processes = (Process *)malloc(n * sizeof(Process));
    for (int i = 0; i < n; i++) {
        printf("Enter details for process %d (Name Arrival Burst): ", i + 1);
        scanf(" %c %d %d", &processes[i].process_name, &processes[i].arrival_time,
&processes[i].burst_time);
        processes[i].completion_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
    sort_by_burst_time(processes, n);
    compute_completion_time(processes, n);
    compute_turnaround_waiting_time(processes, n);
    display_table(processes, n);
    free(processes);
}

```

```

    return 0;
}

```

## Result:

```

Enter number of processes: 4
Enter arrival times:
0 8 3 5
Enter burst times:
7 3 4 6
SJF scheduling:

```

PID	AT	BT	CT	TAT	WT
P1	0	7	7	7	0
P2	8	3	14	6	3
P3	3	4	11	8	4
P4	5	6	20	15	9

```

Average turnaround time:9.000000ms
Average waiting time:4.000000ms

```

## SJF Pre-emptive Code:

```

#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

int findShortestJob(struct Process processes[], int n, int current_time) {
    int shortest_job_index = -1;
    int shortest_job = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0 &&
            processes[i].remaining_time < shortest_job) {
            shortest_job_index = i;
            shortest_job = processes[i].remaining_time;
        }
    }
    return shortest_job_index;
}

```



```

    }
}
return shortest_job_index;
}

void SJF(struct Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    while (completed < n) {
        int shortest_job_index = findShortestJob(processes, n, current_time);
        if (shortest_job_index == -1) {
            current_time++;
        } else {

            processes[shortest_job_index].remaining_time--;
            current_time++;
            if (processes[shortest_job_index].remaining_time == 0) {

                processes[shortest_job_index].completion_time = current_time;
                processes[shortest_job_index].turnaround_time =
processes[shortest_job_index].completion_time - processes[shortest_job_index].arrival_time;
                processes[shortest_job_index].waiting_time =
processes[shortest_job_index].turnaround_time - processes[shortest_job_index].burst_time;
                completed++;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
    }
}

```

```

scanf("%d", &processes[i].burst_time);
processes[i].remaining_time = processes[i].burst_time;
processes[i].pid = i + 1;
}
SJF(processes, n);
printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround
Time\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].arrival_time,
processes[i].burst_time, processes[i].completion_time, processes[i].waiting_time,
processes[i].turnaround_time);
}
return 0;
}

```

### Result:

```

Enter number of processes: 4
Enter arrival times:
0 8 3 5
Enter burst times:
7 3 4 6
SJF scheduling:

```

PID	AT	BT	CT	TAT	WT
P1	0	7	7	7	0
P2	8	3	14	6	3
P3	3	4	11	8	4
P4	5	6	20	15	9

```

Average turnaround time:9.000000ms
Average waiting time:4.000000ms

```

## Program - 2

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

→ Priority

→ Round Robin

### Priority Scheduling Code:

```
#include<stdio.h>

void sort (int proc_id[], int p[], int at[], int bt[], int n){
    int min = p[0], temp = 0;
    for (int i = 0; i < n; i++)
    {
        min = p[i];
        for (int j = i; j < n; j++)
        {
            if (p[j] < min)
            {
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                temp = bt[j];
                bt[j] = bt[i];
                bt[i] = temp;
                temp = p[j];
                p[j] = p[i];
                p[i] = temp;
                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}

int main (){
    int n, c = 0;
    printf ("Enter number of processes: ");
    scanf ("%d", &n);
```

```

int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], rt[n], p[n];
double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;
for (int i = 0; i < n; i++)
{
    proc_id[i] = i + 1;
    m[i] = 0;
}
printf ("Enter priorities:\n");
for (int i = 0; i < n; i++)
    scanf ("%d", &p[i]);
printf ("Enter arrival times:\n");
for (int i = 0; i < n; i++)
    scanf ("%d", &at[i]);
printf ("Enter burst times:\n");
for (int i = 0; i < n; i++)
{
    scanf ("%d", &bt[i]);
    m[i] = -1;
    rt[i] = -1;
}
sort (proc_id, p, at, bt, n);
int count = 0, pro = 0, priority = p[0];
int x = 0;
c = 0;
while (count < n)
{
    for (int i = 0; i < n; i++)
    {
        if (at[i] <= c && p[i] >= priority && m[i] != 1)
        {
            x = i;
            priority = p[i];
        }
    }
    if (rt[x] == -1)
        rt[x] = c - at[x];
    if (at[x] <= c)
        c += bt[x];
    else
        c += at[x] - c + bt[x];
}

```

```

    count++;
    ct[x] = c;
    m[x] = 1;
    while (x >= 1 && m[--x] != 1)
    {
        priority = p[x];
        break;
    }
    x++;
    if (count == n)
        break;
}
for (int i = 0; i < n; i++)
    tat[i] = ct[i] - at[i];
for (int i = 0; i < n; i++)
    wt[i] = tat[i] - bt[i];

printf ("\nPriority scheduling:\n");
printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++)
    printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
        bt[i], ct[i], tat[i], wt[i], rt[i]);
for (int i = 0; i < n; i++)
{
    ttat += tat[i];
    twt += wt[i];
}
avg_tat = ttat / (double) n;
avg_wt = twt / (double) n;
printf ("\nAverage turnaround time:%lfms\n", avg_tat);
printf ("\nAverage waiting time:%lfms\n", avg_wt);
}

```

**Result:**

```

Enter number of processes: 5
Enter priorities:
2 1 3 2 1
Enter arrival times:
2 4 5 1 7
Enter burst times:
4 7 2 4 6

Priority scheduling:

```

PID	Prior	AT	BT	CT	TAT	WT	RT	
P5	1		7	6	13	6	0	-7
P2	1		4	7	30	26	19	19
P1	2		2	4	23	21	17	17
P4	2		1	4	19	18	14	14
P3	3		5	2	15	10	8	8

```

Average turnaround time:16.200000ms
Average waiting time:11.600000ms

```

## Round Robin Code:

```

#include<stdio.h>
void sort (int proc_id[], int at[], int bt[], int b[], int n){
    int min = at[0], temp = 0;
    for (int i = 0; i < n; i++){
        min = at[i];
        for (int j = i; j < n; j++)
        {
            if (at[j] < min)
            {
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                temp = bt[j];
                bt[j] = bt[i];
                bt[i] = temp;
                temp = b[j];
                b[j] = b[i];
                b[i] = temp;
                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}

int main (){
    int n, c = 0, t = 0;

```

```

printf ("Enter number of processes: ");
scanf ("%d", &n);
printf ("Enter Time Quantum: ");
scanf ("%d", &t);
int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], b[n], rt[n], m[n];
int f = -1, r = -1;
int q[100];
int count = 0;
double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;
for (int i = 0; i < n; i++)
    proc_id[i] = i + 1;
printf ("Enter arrival times:\n");
for (int i = 0; i < n; i++)
    scanf ("%d", &at[i]);
printf ("Enter burst times:\n");
for (int i = 0; i < n; i++){
    scanf ("%d", &bt[i]);
    b[i] = bt[i];
    m[i] = 0;
    rt[i] = -1;
}
sort (proc_id, at, bt, b, n);
f = r = 0;
q[0] = proc_id[0];

int p = 0, i = 0;
while (f >= 0){
    p = q[f++];
    i = 0;
    while (p != proc_id[i])
        i++;
    if (b[i] >= t)
    {
        if (rt[i] == -1)
            rt[i] = c;
        b[i] -= t;
        c += t;
        m[i] = 1;
    }
    else

```

```

    {
        if (rt[i] == -1)
            rt[i] = c;
        c += b[i];
        b[i] = 0;
        m[i] = 1;
    }
    m[0] = 1;
    for (int j = 0; j < n; j++){
        if (at[j] <= c && proc_id[j] != p && m[j] != 1)
        {
            q[++r] = proc_id[j];
            m[j] = 1;
        }
    }
    if (b[i] == 0){
        count++;
        ct[i] = c;
    }
    else
        q[++r] = proc_id[i];
    if (f > r)
        f = -1;
}
for (int i = 0; i < n; i++){
    tat[i] = ct[i] - at[i];
    rt[i] = rt[i] - at[i];
}

for (int i = 0; i < n; i++)
    wt[i] = tat[i] - bt[i];
printf ("\nRRS scheduling:\n");
printf ("PID\tAT\tBT\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++)
    printf ("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i],
        tat[i], wt[i], rt[i]);
for (int i = 0; i < n; i++)
{
    ttat += tat[i];
    twt += wt[i];
}

```



```

    }
    avg_tat = ttat / (double) n;
    avg_wt = twt / (double) n;
    printf ("\nAverage turnaround time:%lfms\n", avg_tat);
    printf ("\nAverage waiting time:%lfms\n", avg_wt);
}

```

### Result:

```

Enter number of processes: 5
Enter priorities:
2 1 3 2 1
Enter arrival times:
2 4 5 1 7
Enter burst times:
4 7 2 4 6

Priority scheduling:

```

PID	Prior	AT	BT	CT	TAT	WT	RT	
P5	1		7	6	13	6	0	-7
P2	1		4	7	30	26	19	19
P1	2		2	4	23	21	17	17
P4	2		1	4	19	18	14	14
P3	3		5	2	15	10	8	8

```

Average turnaround time:16.200000ms
Average waiting time:11.600000ms

```

### Program - 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.

#### Code:

```
#include <stdio.h>

void sort(int proc_id[], int at[], int bt[], int n) {
    int min, temp;
    for(int i=0; i<n-1; i++) {
        for(int j=i+1; j<n; j++) {
            if(at[j] < at[i]) {
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}

void simulateFCFS(int proc_id[], int at[], int bt[], int n, int start_time) {
    int c = start_time, ct[n], tat[n], wt[n];
    double ttat = 0.0, twt = 0.0;
    for(int i=0; i<n; i++) {
        if(c >= at[i])
            c += bt[i];
        else
            c = at[i] + bt[i];
        ct[i] = c;
    }
}
```

```

for(int i=0; i<n; i++)
    tat[i] = ct[i] - at[i];

for(int i=0; i<n; i++)
    wt[i] = tat[i] - bt[i];
printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
for(int i=0; i<n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    ttat += tat[i];
    twt += wt[i];
}
printf("Average Turnaround Time: %.2lf ms\n", ttat/n);
printf("Average Waiting Time: %.2lf ms\n", twt/n);
}

void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int proc_id[n], at[n], bt[n], type[n];
    int sys_proc_id[n], sys_at[n], sys_bt[n], user_proc_id[n], user_at[n], user_bt[n];
    int sys_count = 0, user_count = 0;
    for(int i=0; i<n; i++) {
        proc_id[i] = i + 1;
        printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ",
i+1);
        scanf("%d %d %d", &at[i], &bt[i], &type[i]);
        if(type[i] == 0) {
            sys_proc_id[sys_count] = proc_id[i];
            sys_at[sys_count] = at[i];
            sys_bt[sys_count] = bt[i];
            sys_count++;
        } else {
            user_proc_id[user_count] = proc_id[i];
            user_at[user_count] = at[i];
            user_bt[user_count] = bt[i];
            user_count++;
        }
    }
    sort(sys_proc_id, sys_at, sys_bt, sys_count);

```

```

sort(user_proc_id, user_at, user_bt, user_count); //arrival time sort

printf("System Processes Scheduling:\n");
simulateFCFS(sys_proc_id, sys_at, sys_bt, sys_count, 0);

int system_end_time = 0;
if (sys_count > 0) {
    system_end_time = sys_at[sys_count - 1] + sys_bt[sys_count - 1];
    for (int i = 0; i < sys_count - 1; i++) {
        if (sys_at[i + 1] > system_end_time) {
            system_end_time = sys_at[i + 1];
        }
        system_end_time += sys_bt[i];
    }
}
printf("\nUser Processes Scheduling:\n");
simulateFCFS(user_proc_id, user_at, user_bt, user_count, system_end_time);
}

```

## Result:

```

Enter number of processes: 5
Enter arrival time, burst time and type (0 for system, 1 for user) for process 1: 0 4 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 2: 1 2 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 3: 2 3 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 4: 2 2 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 5: 8 3 0
System Processes Scheduling:
PID    AT    BT    CT    TAT    WT
1       0     4     4     4     0
4       2     2     6     4     2
5       8     3    11     3     0
Average Turnaround Time: 3.67 ms
Average Waiting Time: 0.67 ms

User Processes Scheduling:
PID    AT    BT    CT    TAT    WT
2       1     2    19    18    16
3       2     3    22    20    17
Average Turnaround Time: 19.00 ms
Average Waiting Time: 16.50 ms

```

## Program - 4

Write a C program to simulate Real-Time CPU Scheduling algorithms:

- a) Rate- Monotonic
- b) Earliest-deadline First
- c) Proportional scheduling

### a) Rate-Monotonic Code:

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

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

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

```
void sort (int proc[], int b[], int pt[], int n){  
    int temp = 0;  
    for (int i = 0; i < n; i++)  
    {  
        for (int j = i; j < n; j++)  
        {  
            if (pt[j] < pt[i])  
            {  
                temp = pt[i];  
                pt[i] = pt[j];  
                pt[j] = temp;  
                temp = b[j];  
                b[j] = b[i];  
                b[i] = temp;  
                temp = proc[i];  
                proc[i] = proc[j];  
                proc[j] = temp;  
            }  
        }  
    }  
}
```

```
int gcd (int a, int b){  
    int r;  
    while (b > 0)  
    {  
        r = a % b;  
        a = b;
```

```

        b = r;
    }
    return a;
}

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

int main(){
    int n;
    printf ("Enter the number of processes:");
    scanf ("%d", &n);
    int proc[n], b[n], pt[n], rem[n];
    printf ("Enter the CPU burst times:\n");

    for (int i = 0; i < n; i++){
        scanf ("%d", &b[i]);
        rem[i] = b[i];
    }
    printf ("Enter the time periods:\n");

    for (int i = 0; i < n; i++)
        scanf ("%d", &pt[i]);

    for (int i = 0; i < n; i++)
        proc[i] = i + 1;

    sort (proc, b, pt, n);
    int l = lcmul (pt, n);
    printf ("LCM=%d\n", l);
    printf ("\nRate Monotone Scheduling:\n");
    printf ("PID\tBurst\tPeriod\n");
    for (int i = 0; i < n; i++)
        printf ("%d\t%d\t%d\n", proc[i], b[i], pt[i]);
    double sum = 0.0;

```

```

for (int i = 0; i < n; i++){
    sum += (double) b[i] / pt[i];
}

double rhs = n * (pow (2.0, (1.0 / n)) - 1.0);
printf ("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");

if (sum > rhs)
    exit (0);
printf ("Scheduling occurs for %d ms\n\n", l);
int time = 0, prev = 0, x = 0;

while (time < l){
    int f = 0;

    for (int i = 0; i < n; i++)
    {
        if (time % pt[i] == 0)
            rem[i] = b[i];
        if (rem[i] > 0)
        {
            if (prev != proc[i])
            {
                printf ("%dms onwards: Process %d running\n", time,
                    proc[i]);
                prev = proc[i];
            }
            rem[i]--;
            f = 1;
            break;
            x = 0;
        }
    }
    if (!f)
    {
        if (x != 1)
        {
            printf ("%dms onwards: CPU is idle\n", time);
            x = 1;
        }
    }
}

```

```

    }
    time++;
}
}

```

### Result:

```

Enter the number of processes:2
Enter the CPU burst times:
20
35
Enter the time periods:
50 100
LCM=100

Rate Monotone Scheduling:
PID      Burst  Period
1         20    50
2         35   100

0.750000 <= 0.828427 =>true
Scheduling occurs for 100 ms

0ms onwards: Process 1 running
20ms onwards: Process 2 running
50ms onwards: Process 1 running
70ms onwards: Process 2 running
75ms onwards: CPU is idle

```

### b) Earliest Deadline First Code:

```

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

void sort (int proc[], int d[], int b[], int pt[], int n){
    int temp = 0;
    for (int i = 0; i < n; i++){
        for (int j = i; j < n; j++){
            if (d[j] < d[i]){
                temp = d[j];
                d[j] = d[i];
                d[i] = temp;
                temp = pt[i];
                pt[i] = pt[j];
            }
        }
    }
}

```



```

        pt[j] = temp;
        temp = b[j];
        b[j] = b[i];
        b[i] = temp;
        temp = proc[i];
        proc[i] = proc[j];
        proc[j] = temp;
    }
}
}

```

```

int gcd (int a, int b){
    int r;
    while (b > 0)
    {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}

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

```

```

int main (){
    int n;
    printf ("Enter the number of processes:");
    scanf ("%d", &n);
    int proc[n], b[n], pt[n], d[n], rem[n];
    printf ("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++)
    {
        scanf ("%d", &b[i]);
    }
}

```

```

    rem[i] = b[i];
}
printf ("Enter the deadlines:\n");
for (int i = 0; i < n; i++)
    scanf ("%d", &d[i]);
printf ("Enter the time periods:\n");
for (int i = 0; i < n; i++)
    scanf ("%d", &pt[i]);
for (int i = 0; i < n; i++)
    proc[i] = i + 1;
sort (proc, d, b, pt, n);
int l = lcmul (pt, n);
printf ("\nEarliest Deadline Scheduling:\n");
printf ("PID\tBurst\tDeadline\tPeriod\n");
for (int i = 0; i < n; i++)
    printf ("%d\t%d\t%d\t%d\n", proc[i], b[i], d[i], pt[i]);
printf ("Scheduling occurs for %d ms\n\n", l);
int time = 0, prev = 0, x = 0;
int nextDeadlines[n];
for (int i = 0; i < n; i++)
{
    nextDeadlines[i] = d[i];
    rem[i] = b[i];
}
while (time < l)
{
    for (int i = 0; i < n; i++)
    {
        if (time % pt[i] == 0 && time != 0)
        {
            nextDeadlines[i] = time + d[i];
            rem[i] = b[i];
        }
    }
    int minDeadline = l + 1;
    int taskToExecute = -1;
    for (int i = 0; i < n; i++){
        if (rem[i] > 0 && nextDeadlines[i] < minDeadline){
            minDeadline = nextDeadlines[i];
            taskToExecute = i;

```

```

    }
}
if (taskToExecute != -1){
    printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
    rem[taskToExecute]--;
}
else{
    printf ("%dms: CPU is idle.\n", time);
}
time++;
}
}

```

## Result:

```

Enter the number of processes:2
Enter the CPU burst times:
2 4
Enter the deadlines:
5 10
Enter the time periods:
5 10

Earliest Deadline Scheduling:
PID      Burst  Deadline      Period
1         2      5              5
2         4      10             10
Scheduling occurs for 10 ms

0ms : Task 1 is running.
1ms : Task 1 is running.
2ms : Task 2 is running.
3ms : Task 2 is running.
4ms : Task 2 is running.
5ms : Task 1 is running.
6ms : Task 1 is running.
7ms : Task 2 is running.
8ms: CPU is idle.
9ms: CPU is idle.

```

### c) Proportional Scheduling Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    srand(time(NULL));
    int n;
    printf("Enter number of processes:");
    scanf("%d",&n);
    int p[n],t[n],cum[n],m[n];int c=0;int total = 0,count=0;
    printf("Enter tickets of the processes:\n");
    for(int i=0;i<n;i++){
        scanf("%d",&t[i]);
        c+=t[i];
        cum[i]=c;
        p[i]=i+1;
        m[i]=0;
        total+= t[i];
    }
    while(count<n){
        int wt=rand()%total;
        for (int i=0;i<n;i++)
        {
            if (wt<cum[i] && m[i]==0)
            {
                printf("The winning number is %d and winning participant is: %d\n",wt,p[i]);
                m[i]=1;count++;
            }
        }
    }
    printf("\nProbabilities:\n");
    for (int i = 0; i < n; i++)
    {
        printf("The probability of P%d winning: %.2f\n",p[i],((double)t[i]/total*100));
    }
}
```

### Result:

```
Enter number of processes:3
Enter tickets of the processes:
5 10 20
The winning number is 12 and winning participant is: 2
The winning number is 12 and winning participant is: 3
The winning number is 2 and winning participant is: 1

Probabilities:
The probability of P1 winning: 14.29
The probability of P2 winning: 28.57
The probability of P3 winning: 57.14
```

## Program - 5

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

### Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int buffer[MAX];
int empty = MAX;
int full = 0;
int mutex = 1;
int x = 0;
void custom_wait(int* s) {
    while (*s <= 0);
    --(*s);
}
void custom_signal(int* s) {
    ++(*s);
}
void producer() {
    custom_wait(&mutex);
    custom_wait(&empty);
    x++;
    buffer[full] = x;
    custom_signal(&full);
    custom_signal(&mutex);
    printf("Producer produced %d.\n", x);
    printf("Empty = %d\n", empty);
    printf("Buffer:\n");
    for (int i = 0; i < full; i++) {
        printf("%d\t", buffer[i]);
    }
    printf("%d\n", buffer[full - 1]); /
}
void consumer() {
    custom_wait(&full);
    custom_wait(&mutex);
    printf("Consumer consumed %d.\n", buffer[full - 1]);
```

```

    full--;
    custom_signal(&empty);
    custom_signal(&mutex);
    printf("Empty = %d\n", empty);
    printf("Buffer:\n");
    for (int i = 0; i < full; i++) {
        printf("%d\t", buffer[i]);
    }
    printf("\n");
}
int main() {
    int ch;
    while (1) {
        printf("1.Produce\t2.Consume\t3.Exit\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                if (mutex == 1 && empty != 0) {
                    producer();
                } else {
                    printf("Buffer is full\n");
                }
                break;
            case 2:
                if (mutex == 1 && full != 0) {
                    consumer();
                } else {
                    printf("Buffer is empty\n");
                }
                break;
            case 3:
                exit(0);
        }
    }
}

```

Result:

```
1.Produce      2.Consume      3.Exit
2
Buffer is empty
1.Produce      2.Consume      3.Exit
1
Producer produced 1.
Empty = 4
Buffer:
1      1
1.Produce      2.Consume      3.Exit
1
Producer produced 2.
Empty = 3
Buffer:
1      2      2
1.Produce      2.Consume      3.Exit
2
Consumer consumed 1.
Empty = 4
Buffer:

1.Produce      2.Consume      3.Exit
3
```



## Program - 6

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

### Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PHILOSOPHERS 5

void allow_one_to_eat(int hungry[], int n) {
    int isWaiting[MAX_PHILOSOPHERS];
    for (int i = 0; i < n; i++) {
        isWaiting[i] = 1;
    }
    for (int i = 0; i < n; i++) {
        printf("P %d is granted to eat\n", hungry[i]);
        isWaiting[hungry[i]] = 0;
        for (int j = 0; j < n; j++) {
            if (isWaiting[hungry[j]]) {
                printf("P %d is waiting\n", hungry[j]);
            }
        }
    }
    for (int k = 0; k < n; k++) {
        isWaiting[k] = 1;
    }
    isWaiting[hungry[i]] = 0;
}

void allow_two_to_eat(int hungry[], int n) {
    if (n < 2 || n > MAX_PHILOSOPHERS) {
        printf("Invalid number of philosophers.\n");
        return;
    }
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);
            for (int k = 0; k < n; k++) {
                if (k != i && k != j) {
```

```

        printf("P %d is waiting\n", hungry[k]);
    }
}
}
}

int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: ");
    scanf("%d", &total_philosophers);
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
        printf("Invalid number of philosophers.\n");
        return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);
    if (hungry_count < 1 || hungry_count > total_philosophers) {
        printf("Invalid number of hungry philosophers.\n");
        return 1;
    }
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter philosopher %d position: ", i + 1);
        scanf("%d", &hungry_positions[i]);
        if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
            printf("Invalid philosopher position.\n");
            return 1;
        }
    }
    int choice;
    while (1) {
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:

```

```

        allow_one_to_eat(hungry_positions, hungry_count);
        break;
    case 2:
        allow_two_to_eat(hungry_positions, hungry_count);
        break;
    case 3:
        exit(0);
    default:
        printf("Invalid choice\n");
    }
}
return 0;
}

```

## Result:

```

DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 4
Enter philosopher 1 position: 1
Enter philosopher 2 position: 2
Enter philosopher 3 position: 3
Enter philosopher 4 position: 4

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
P 1 is granted to eat
P 2 is waiting
P 3 is waiting
P 4 is waiting
P 2 is granted to eat
P 3 is waiting
P 4 is waiting
P 3 is granted to eat
P 1 is waiting
P 4 is waiting
P 4 is granted to eat
P 1 is waiting
P 2 is waiting

```

## Program - 7

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

### Code:

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

void calculateNeed(int P, int R, int need[P][R], int max[P][R], int allot[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allot[i][j];
}

bool isSafe(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {
    int need[P][R];
    calculateNeed(P, R, need, max, allot);
    bool finish[P];
    for (int i = 0; i < P; i++) {
        finish[i] = 0;
    }
    int safeSeq[P];
    int work[R];
    for (int i = 0; i < R; i++) {
        work[i] = avail[i];
    }
    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;
                if (j == R) {
                    printf("P%d is visited ", p);
                    for (int k = 0; k < R; k++) {
```

```

        work[k] += allot[p][k];
        printf("%d ", work[k]);
    }
    printf("\n");
    safeSeq[count++] = p;
    finish[p] = 1;
    found = true;
}
}
}
if (found == false) {
    printf("System is not in safe state\n");
    return false;
}
}
printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");
for (int i = 0; i < P; i++) {
    printf("P%d ", safeSeq[i]);
}
printf("\n");
return true;
}

int main() {
    int P, R;
    printf("Enter number of processes: ");
    scanf("%d", &P);
    printf("Enter number of resources: ");
    scanf("%d", &R);
    int processes[P];
    int avail[R];
    int max[P][R];
    int allot[P][R];
    for (int i = 0; i < P; i++) {
        processes[i] = i;
    }
    for (int i = 0; i < P; i++) {
        printf("Enter details for P%d\n", i);
        printf("Enter allocation -- ");
        for (int j = 0; j < R; j++) {

```

```

        scanf("%d", &allot[i][j]);
    }
    printf("Enter Max -- ");
    for (int j = 0; j < R; j++) {
        scanf("%d", &max[i][j]);
    }
}
printf("Enter Available Resources -- ");
for (int i = 0; i < R; i++) {
    scanf("%d", &avail[i]);
}
isSafe(P, R, processes, avail, max, allot);
printf("\nProcess\tAllocation\tMax\tNeed\n");
for (int i = 0; i < P; i++) {
    printf("P%d\t", i);
    for (int j = 0; j < R; j++) {
        printf("%d ", allot[i][j]);
    }
    printf("\t");
    for (int j = 0; j < R; j++) {
        printf("%d ", max[i][j]);
    }
    printf("\t");
    for (int j = 0; j < R; j++) {
        printf("%d ", max[i][j] - allot[i][j]);
    }
    printf("\n");
}
return 0;
}

```

## Result:

```
Enter number of processes: 3
Enter number of resources: 3
Enter details for P0
Enter allocation -- 1 2 3
Enter Max -- 2 4 3
Enter details for P1
Enter allocation -- 1 0 3
Enter Max -- 4 5 6
Enter details for P2
Enter allocation -- 1 0 0
Enter Max -- 1 2 1
Enter Available Resources -- 1 2 0
P0 is visited (2 4 3 )
P2 is visited (3 4 3 )
System is not in safe state
```

Process	Allocation	Max	Need
P0	1 2 3	2 4 3	1 2 0
P1	1 0 3	4 5 6	3 5 3
P2	1 0 0	1 2 1	0 2 1

## Program -8

Write a C program to simulate deadlock detection.

Code:

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function prototypes
bool isCycle(int process, bool visited[], bool *recStack, int allocation[][MAX_RESOURCES],
int n, int m);
bool isDeadlocked(int allocation[][MAX_RESOURCES], int n, int m);

int main() {
    int n, m; // n -> number of processes, m -> number of resources types

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

    printf("Enter number of resource types: ");
    scanf("%d", &m);

    int allocation[MAX_PROCESSES][MAX_RESOURCES]; // Allocation matrix

    // Input allocation matrix
    printf("Enter allocation matrix:\n");
```



```

for (int i = 0; i < n; i++) {
    printf("Process %d: ", i);
    for (int j = 0; j < m; j++) {
        scanf("%d", &allocation[i][j]);
    }
}

// Check if there is a deadlock
if (isDeadlocked(allocation, n, m)) {
    printf("\nDeadlock detected.\n");
} else {
    printf("\nNo deadlock detected.\n");
}

return 0;
}

// Function to detect if there is a cycle in the resource allocation graph using DFS
bool isCycle(int process, bool visited[], bool *recStack, int allocation[][MAX_RESOURCES],
int n, int m) {
    if (!visited[process]) {
        visited[process] = true;
        recStack[process] = true;

        for (int i = 0; i < m; i++) {
            int resource = allocation[process][i];
            if (resource > 0) {
                if (!visited[resource] && isCycle(resource, visited, recStack, allocation, n, m)) {
                    return true;
                }
            }
        }
    }
}

```

```

        } else if (recStack[resource]) {
            return true;
        }
    }
}

recStack[process] = false;
return false;
}

// Function to check if there is a deadlock in the system
bool isDeadlocked(int allocation[][MAX_RESOURCES], int n, int m) {
    bool visited[MAX_PROCESSES] = {false};
    bool recStack[MAX_PROCESSES] = {false};

    // Check for deadlock using DFS traversal
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            if (isCycle(i, visited, recStack, allocation, n, m)) {
                return true;
            }
        }
    }

    return false;
}

```

Result: --

```
Enter number of processes: 3
Enter number of resource types: 3
Enter allocation matrix:
Process 0: 1 0 2
Process 1: 2 1 3
Process 2: 1 1 2

Deadlock detected.
```

### Program 9: -

Write a C program to simulate the following contiguous memory allocation techniques:

- a) Worst-fit
- b) Best-fit
- c) First-fit

Code: --

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

#define MAX_BLOCKS 10
#define MAX_PROCESSES 10

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation as -1
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
```

```

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation as -1
    }

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    printf("\nBest Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation as -1
    }

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])
                    worstIdx = j;
            }
        }
    }
}

```

```

    }
}

if (worstIdx != -1) {
    allocation[i] = worstIdx;
    blockSize[worstIdx] -= processSize[i];
}
}

printf("\nWorst Fit Allocation:\n");
printf("Process No.\tProcess Size\tBlock No.\n");
for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
    else
        printf("Not Allocated\n");
}
}

int main() {
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];
    int m, n;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &m);
    printf("Enter the size of each memory block:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &blockSize[i]);
    }

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processSize[i]);
    }

    // Make copies of block sizes to reuse for different allocation methods
    int blockSize1[MAX_BLOCKS], blockSize2[MAX_BLOCKS],
    blockSize3[MAX_BLOCKS];
    for (int i = 0; i < m; i++) {
        blockSize1[i] = blockSize2[i] = blockSize3[i] = blockSize[i];
    }

    firstFit(blockSize1, m, processSize, n);

```

```

    bestFit(blockSize2, m, processSize, n);
    worstFit(blockSize3, m, processSize, n);

    return 0;
}

```

Result :-

```

Enter the number of memory blocks: 5
Enter the size of each memory block:
100 500 200 300 600
Enter the number of processes: 4
Enter the size of each process:
212 417 112 426

First Fit Allocation:
Process No.    Process Size    Block No.
1              212           2
2              417           5
3              112           2
4              426          Not Allocated

Best Fit Allocation:
Process No.    Process Size    Block No.
1              212           4
2              417           2
3              112           3
4              426           5

Worst Fit Allocation:
Process No.    Process Size    Block No.
1              212           5
2              417           2
3              112           5
4              426          Not Allocated

```

### Program 10:-

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

Code: -

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

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

void FIFO(int pages[], int n, int frame_size) {
    int frames[frame_size];
    for (int i = 0; i < frame_size; i++) frames[i] = -1;
    int index = 0, faults = 0;

    printf("FIFO: \n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < frame_size; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frames[index] = pages[i];
            index = (index + 1) % frame_size;
            faults++;
        }
        printFrames(frames, frame_size);
    }
    printf("Total faults: %d\n", faults);
}

void LRU(int pages[], int n, int frame_size) {
    int frames[frame_size], last_used[frame_size];
    for (int i = 0; i < frame_size; i++) frames[i] = -1;
    for (int i = 0; i < frame_size; i++) last_used[i] = 0;
    int time = 0, faults = 0;

    printf("LRU: \n");
```



```

for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < frame_size; j++) {
        if (frames[j] == pages[i]) {
            found = 1;
            last_used[j] = ++time;
            break;
        }
    }
    if (!found) {
        int lru_index = 0;
        for (int j = 1; j < frame_size; j++) {
            if (last_used[j] < last_used[lru_index])
                lru_index = j;
        }
        frames[lru_index] = pages[i];
        last_used[lru_index] = ++time;
        faults++;
    }
    printFrames(frames, frame_size);
}
printf("Total faults: %d\n", faults);
}

void OPTIMAL(int pages[], int n, int frame_size) {
    int frames[frame_size];
    for (int i = 0; i < frame_size; i++) frames[i] = -1;
    int faults = 0;

    printf("OPTIMAL: \n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < frame_size; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            int replace_index = 0, farthest = -1;
            for (int j = 0; j < frame_size; j++) {
                int k;
                for (k = i + 1; k < n; k++) {
                    if (frames[j] == pages[k]) {
                        if (k > farthest) {
                            farthest = k;
                        }
                    }
                }
            }
        }
    }
}

```

```

        replace_index = j;
    }
    break;
}
}
if (k == n) {
    replace_index = j;
    break;
}
}
frames[replace_index] = pages[i];
faults++;
}
printFrames(frames, frame_size);
}
printf("Total faults: %d\n", faults);
}

```

```

int main() {
    int n, frame_size;

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

    int *pages = (int *)malloc(n * sizeof(int));
    printf("Enter the page sequence: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &frame_size);

    FIFO(pages, n, frame_size);
    LRU(pages, n, frame_size);
    OPTIMAL(pages, n, frame_size);

    free(pages);
    return 0;
}

```

Result:-

```
Enter the number of pages: 14
Enter the page sequence: 7 0 1 2 0 3 0 4 2 3 0 3 2 3
Enter the number of frames: 4
```

FIFO:

```
7 -1 -1 -1
7 0 -1 -1
7 0 1 -1
7 0 1 2
7 0 1 2
3 0 1 2
3 0 1 2
3 4 1 2
3 4 1 2
3 4 1 2
3 4 0 2
3 4 0 2
3 4 0 2
3 4 0 2
Total faults: 7
```

LRU:

```
7 -1 -1 -1
7 0 -1 -1
7 0 1 -1
7 0 1 2
7 0 1 2
3 0 1 2
3 0 1 2
3 0 4 2
3 0 4 2
3 0 4 2
3 0 4 2
3 0 4 2
3 0 4 2
3 0 4 2
Total faults: 6
```

OPTIMAL:

```
7 -1 -1 -1
0 -1 -1 -1
0 1 -1 -1
0 2 -1 -1
0 2 -1 -1
0 2 3 -1
0 2 3 -1
0 2 3 4
0 2 3 4
0 2 3 4
0 2 3 4
0 2 3 4
0 2 3 4
0 2 3 4
Total faults: 6
```