VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

TRISHA L SALIAN (1WA23CS022)

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

Feb-2025 to June-2025

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 TRISHA L SALIAN (1WA23CS022), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025-June 2025. 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.

Seema Patil	Dr. Kavitha Sooda
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	Conduct practical experiments to implement the functionalities of Operating system.

PROGRAM-1

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

a) FCFS

b) SJF

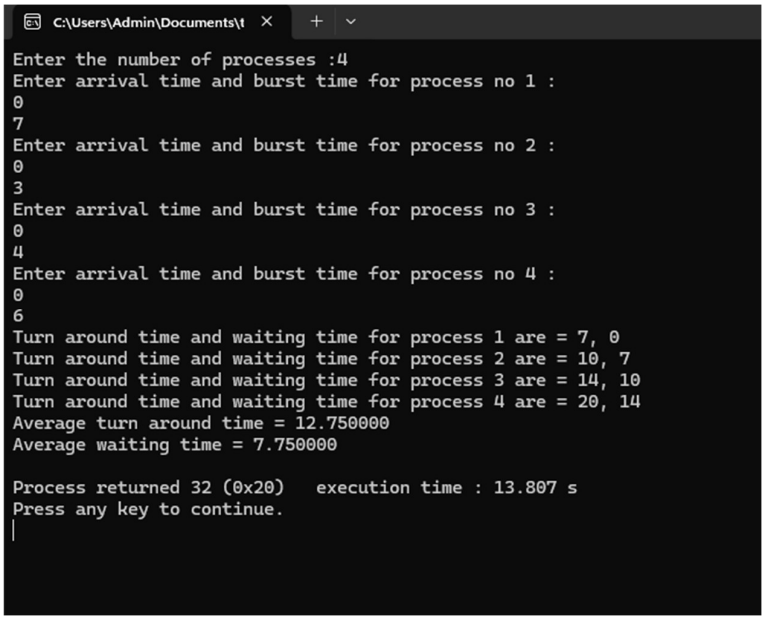
c) Priority Code:

FCFS:

```
#include<stdio.h>
void main()
{
   int i,n;
   printf("Enter the number of processes :");
   scanf("%d",&n);
   int pno[n],at[n],bt[n],ct[n],tat[n],wt[n];
   for(i=0;i<n;i++)
   {
      printf("Enter arrival time and burst time for process no %d :\n",i+1);
      scanf("%d",&at[i]);
      scanf("%d",&bt[i]);
   }
   ct[0]=bt[0];

   for(i=1;i<n;i++)
   {
      ct[i]=ct[i-1]+bt[i];
   }
   for(i=0;i<n;i++)
   {
      tat[i]=ct[i]-at[i];
      wt[i]=tat[i]-bt[i];
```

```
        printf("Turn around time and waiting time for process %d are = %d, %d
\n",i+1,tat[i],wt[i]);
    }
    float avgtat=0.0,avgwt=0.0;
    for(i=0;i<n;i++)
    {
        avgtat=avgtat+tat[i];
        avgwt=avgwt+wt[i];
    }
    avgtat=(avgtat/n);
    avgwt=(avgwt/n);
    printf("Average turn around time = %f\n",avgtat);
    printf("Average waiting time = %f\n",avgwt);
}
```

OUTPUT



LAB-1

Lab-1
Write a C program to simulate the following non-preemtive CPU scheduling algorithm to find their turn around time & waiting time.

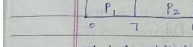
1) FCFS (First come First Serve)

Process	AT	BT	CT	TAT	WT
1	0	7	7	7	0
2	0	3	10	10	7
3	0	4	14	14	10
4	0	6	20	20	14

AVG=12.75 AVG=7.75

TAT = CT - AT
WT = TAT - BT

Gantt Chart




```
#include<stdio.h>
void main()
{
  int i,n;
  printf("Enter the number of processes : ");
  scanf("%d",&n);
  int pro[n], at[n], bt[n], ct[n], tat[n], wt[n];
  for(i=0; i<n; i++)
  {
    printf("Enter arrival time and burst time for process no %d : \n",i+1);
    scanf("%d", &at[i]);
    scanf("%d", &bt[i]);
  }
  ct[0] = bt[0];
  for(i=1; i<n; i++)
  {
    ct[i] = ct[i-1] + bt[i];
  }
  for(i=0; i<n; i++)
  {
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
    printf("Turn around time and waiting time for process %d are = %d, %d \n",
        i+1, tat[i], wt[i]);
  }
  float avgtat = 0.0, avgwt = 0.0;
  for(i=0; i<n; i++)
  {
    avgtat = avgtat + tat[i];
    avgwt = avgwt + wt[i];
  }
  avgtat = (avgtat/n);
  avgwt = (avgwt/n);
  printf("Average turn around time = %f \n",
      avgtat);
  printf("Average waiting time = %f \n", avgwt);
}
```

Output :

```
Enter the number of processes : 4
Enter arrival time and burst time for process no 1:
0
7
Enter arrival time and burst time for process no 2:
0
3
Enter arrival time and burst time for process no 3:
0
4
Enter arrival time and burst time for process no 4:
0
6
Turn around time and waiting time for process 1 are = 7, 0
Turn around time and waiting time for process 2 are = 10, 7
Turn around time and waiting time for process 3 are = 14, 10
Turn around time and waiting time for process 4 are = 20, 14
Average turn around time = 12.750000
Average waiting time = 7.750000
```

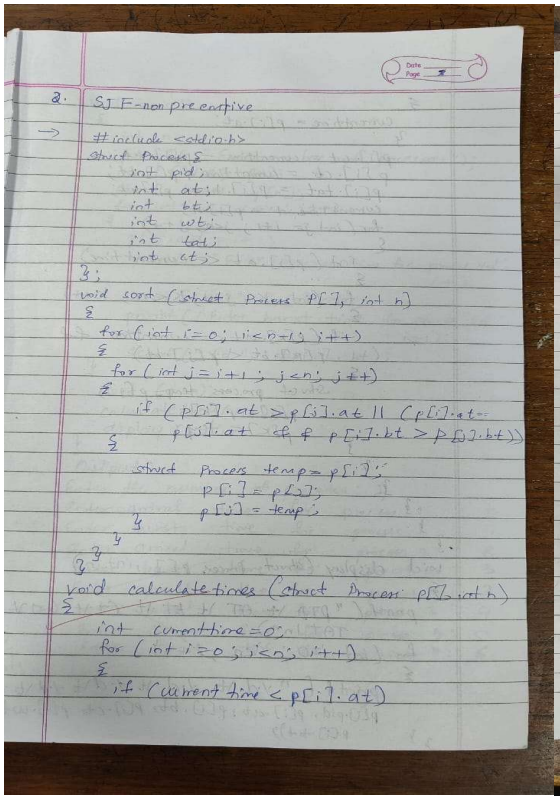
SJF (NON-PREEMTIVE):

```
#include <stdio.h>

#define MAX 100

void main() {
   int n, at[MAX], bt[MAX], ct[MAX], tat[MAX], wt[MAX], rt[MAX];
   int is_completed[MAX] = {0};
   int time = 0, completed = 0, min_bt, idx;

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

   for(int i = 0; i < n; i++) {
      printf("Enter arrival time and burst time for process %d: ", i + 1);
      scanf("%d%d", &at[i], &bt[i]);
   }

   while(completed != n) {
      min_bt = 1e9;
      idx = -1;
      for(int i = 0; i < n; i++) {
         if(at[i] <= time && !is_completed[i]) {
            if(bt[i] < min_bt) {
               min_bt = bt[i];
               idx = i;
            }
            else if(bt[i] == min_bt) {
               if(at[i] < at[idx]) {
```

```
                    idx = i;
                }
            }
        }
    }

    if(idx != -1) {
        rt[idx] = time - at[idx];
        time += bt[idx];
        ct[idx] = time;
        tat[idx] = ct[idx] - at[idx];
        wt[idx] = tat[idx] - bt[idx];
        is_completed[idx] = 1;
        completed++;
    } else {
        time++;
    }
  }

  printf("\nP\tAT\tBT\tCT\tTAT\tWT\tRT\n");
  for(int i = 0; i < n; i++) {
    printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);
  }

}
```

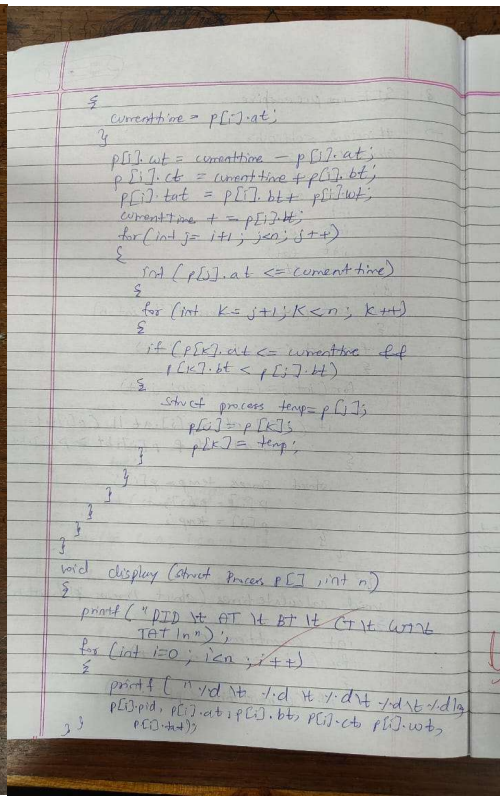
OUTPUT :

```
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0 8
Enter arrival time and burst time for process 2: 1 4
Enter arrival time and burst time for process 3: 2 9
Enter arrival time and burst time for process 4: 3 5

P       AT      BT      CT      TAT     WT      RT
P1      0       8       8       8       0       0
P2      1       4       12      11      7       7
P3      2       9       26      24      15      15
P4      3       5       17      14      9       9

Process returned 0 (0x0)   execution time : 57.585 s
Press any key to continue.
```

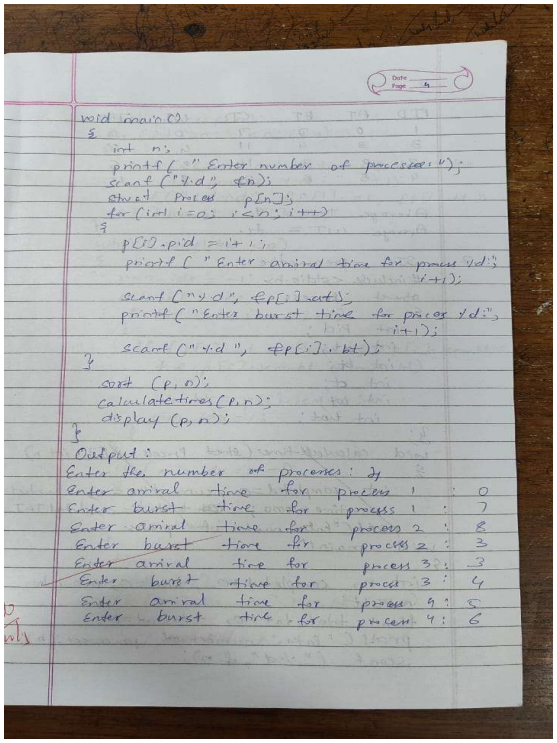


2. SJF=non preemptive

```
#include <stdio.h>
struct Process{
    int pid;
    int at;
    int bt;
    int wt;
    int tat;
    int ct;
};
void sort (struct Process p[], int n)
{
  for (int i=0; i<n-1; i++)
  {
    for (int j=i+1; j<n; j++)
    {
      if (p[i].at > p[j].at || (p[i].at==
          p[j].at && p[i].bt > p[j].bt))
      {
        struct Process temp=p[i];
        p[i] = p[j];
        p[j] = temp;
      }
    }
  }
}
void calculatetimes (struct Process p[], int n)
{
  int currenttime=0;
  for (int i=0; i<n; i++)
  {
    if (current time < p[i].at)
```



```
    currenttime = p[i].at;
  }
  p[i].wt = currenttime - p[i].at;
  p[i].ct = currenttime + p[i].bt;
  p[i].tat = p[i].bt + p[i].wt;
  currenttime += p[i].bt;
  for (int j=i+1; j<n; j++)
  {
    int (p[j].at <= current time)
    {
      for (int k=j+1; k<n; k++)
      {
        if (p[k].at <= currenttime &&
            p[k].bt < p[j].bt)
        {
          struct process temp=p[j];
          p[j]=p[k];
          p[k]=temp;
        }
      }
    }
  }
}
void display (struct Process p[], int n)
{
  printf("PID \t AT \t BT \t CT \t WT\t
          TAT \n");
  for (int i=0; i<n; i++)
  {
    printf("%d \t %d \t %d \t %d \t %d \t %d\n",
    p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt,
    p[i].tat);
  }
}
```



```
void main()
{
  int n;
  printf("Enter number of processes: ");
  scanf("%d", &n);
  struct Process p[n];
  for (int i=0; i<n; i++)
  {
    p[i].pid = i+1;
    printf("Enter arrival time for process %d:",
            i+1);
    scanf("%d", &p[i].at);
    printf("Enter burst time for process %d:",
            i+1);
    scanf("%d", &p[i].bt);
  }
  sort(p, n);
  calculatetimes(p, n);
  display(p, n);
}
```

Output:

```
Enter the number of processes: 4
Enter arrival time for process 1 : 0
Enter burst time for process 1 : 7
Enter arrival time for process 2 : 8
Enter burst time for process 2 : 3
Enter arrival time for process 3 : 3
Enter burst time for process 3 : 4
Enter arrival time for process 4 : 5
Enter burst time for process 4 : 6
```

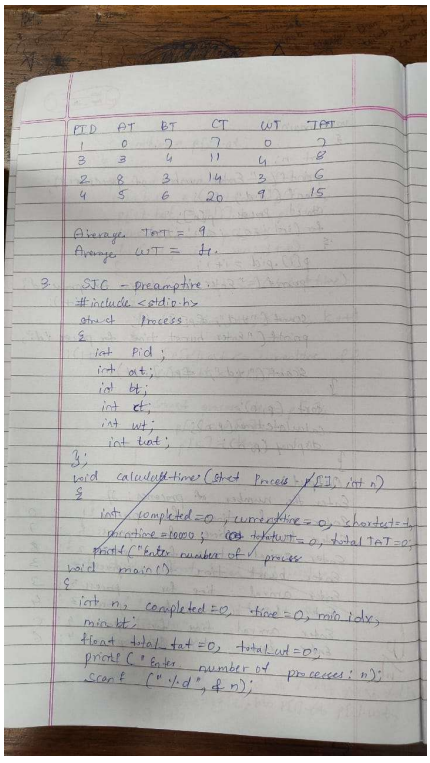
SRTF :

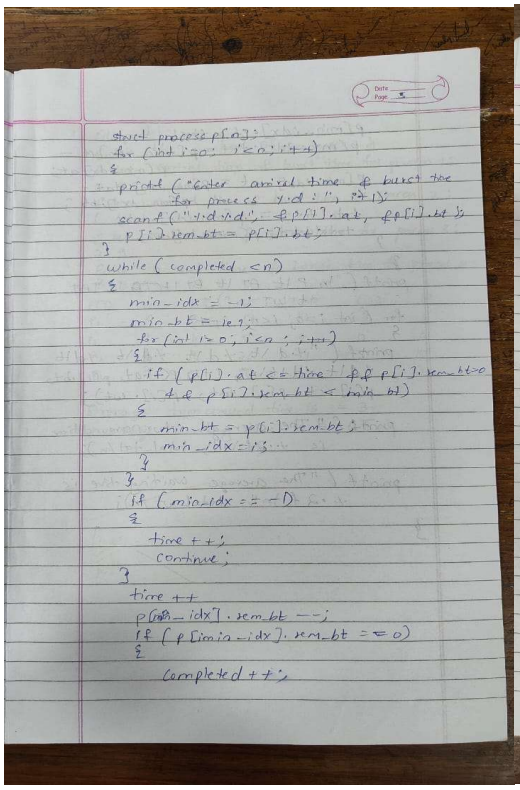
```
#include <stdio.h>

#define MAX 100

void main() {
   int n, at[MAX], bt[MAX], rt[MAX], ct[MAX], tat[MAX], wt[MAX], start_time[MAX];
   int time = 0, completed = 0, shortest = -1, min_rt = 1e9;
   int is_completed[MAX] = {0};

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

   for(int i = 0; i < n; i++) {
      printf("Enter arrival time and burst time for process %d: ", i+1);
      scanf("%d%d", &at[i], &bt[i]);
      rt[i] = bt[i];
      start_time[i] = -1;
   }

   while(completed != n) {
      shortest = -1;
      min_rt = 1e9;
      for(int i = 0; i < n; i++) {
         if(at[i] <= time && is_completed[i] == 0 && rt[i] < min_rt && rt[i] > 0) {
            min_rt = rt[i];
            shortest = i;
         }
      }
```

```
        if(shortest == -1) {
            time++;
            continue;
        }

        if(start_time[shortest] == -1)
            start_time[shortest] = time;

        rt[shortest]--;
        time++;

        if(rt[shortest] == 0) {
            ct[shortest] = time;
            tat[shortest] = ct[shortest] - at[shortest];
            wt[shortest] = tat[shortest] - bt[shortest];
            is_completed[shortest] = 1;
            completed++;
        }
    }

    printf("\nP\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for(int i = 0; i < n; i++)
    {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt[i], ct[i], tat[i], wt[i], start_time[i] -
at[i]);
    }
}
```

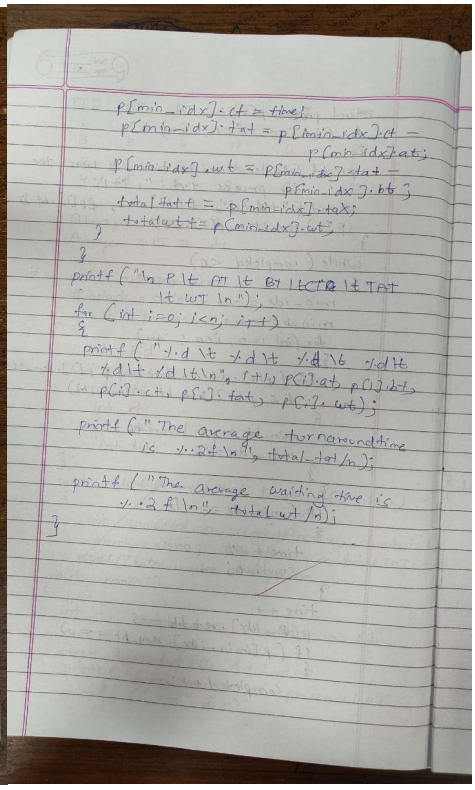
OUTPUT :

```
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0 8
Enter arrival time and burst time for process 2: 1 4
Enter arrival time and burst time for process 3: 2 9
Enter arrival time and burst time for process 4: 3 5

P       AT      BT      CT      TAT     WT      RT
P1      0       8       17      17      9       0
P2      1       4       5       4       0       0
P3      2       9       26      24      15      15
P4      3       5       10      7       2       2

Process returned 0 (0x0)   execution time : 71.138 s
Press any key to continue.
```

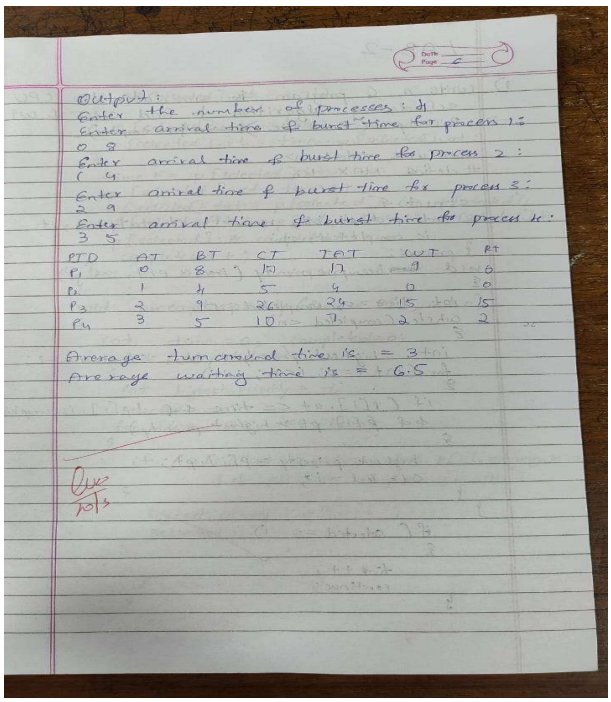




```
struct process p[n];
for (int i=0; i<n; i++)
{
printf ("Enter arrival time & burst time
        for process %d :", i+1);
scanf ("%d %d", &p[i].at, &p[i].bt);
p[i].rem_bt = p[i].bt;
}
while (completed < n)
{
  min_idx = -1;
  min_bt = 1e7;
  for (int i=0; i<n; i++)
  {
    if (p[i].at <= time && p[i].rem_bt>0
        && p[i].rem_bt < min_bt)
    {
      min_bt = p[i].rem_bt;
      min_idx = i;
    }
  }
  if (min_idx == -1)
  {
    time++;
    continue;
  }
  time++
  p[min_idx].rem_bt--;
  if (p[min_idx].rem_bt == 0)
  {
    completed++;
```



```
p[min_idx].ct = time;
p[min_idx].tat = p[min_idx].ct -
                 p[min_idx].at;
p[min_idx].wt = p[min_idx].tat -
                p[min_idx].bt;
total_tat += p[min_idx].tat;
total_wt += p[min_idx].wt;
}
}
printf ("\n P \t AT \t BT \t CT \t TAT
        \t WT \n");
for (int i=0; i<n; i++)
{
printf ("%d \t %d \t %d \t %d \t
%d \t %d \t\n", i+1, p[i].at, p[i].bt,
p[i].ct, p[i].tat, p[i].wt);
printf ("The average turnaround time
        is %.2f \n", total_tat/n);
printf ("The average waiting time is
        %.2f \n", total_wt/n);
}
```



Output:

Enter the number of processes : 4
Enter arrival time & burst time for process 1 :
0 8
Enter arrival time & burst time for process 2 :
1 4
Enter arrival time & burst time for process 3 :
2 9
Enter arrival time & burst time for process 4 :
3 5

PID	AT	BT	CT	TAT	WT	RT
P_1	0	8	17	17	9	0
P_2	1	4	5	4	0	0
P_3	2	9	26	24	15	15
P_4	3	5	10	7	2	2

Average turnaround time is = 13
Average waiting time is = 6.5

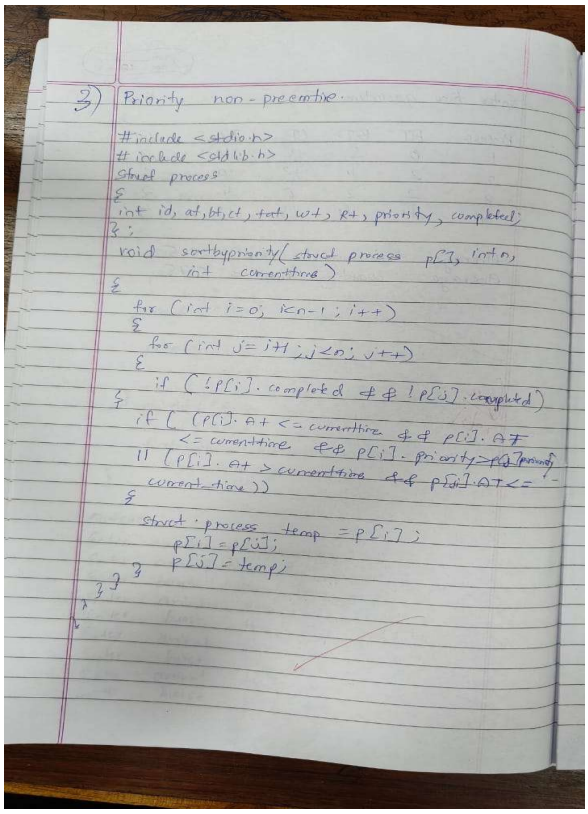
PRIORITY (NON PRE-EMTIVE)

```
#include <stdio.h>

#define MAX 100

void main() {
    int n, at[MAX], bt[MAX], ct[MAX], tat[MAX], wt[MAX], pr[MAX];
    int is_completed[MAX] = {0};
    int time = 0, completed = 0, min_priority, idx;

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

    for(int i = 0; i < n; i++) {
        printf("Enter arrival time, burst time, and priority for process %d: ", i + 1);
        scanf("%d%d%d", &at[i], &bt[i], &pr[i]);
    }

    while(completed != n) {
        min_priority = 1e9;
        idx = -1;
        for(int i = 0; i < n; i++) {
            if(at[i] <= time && !is_completed[i]) {
                if(pr[i] < min_priority) {
                    min_priority = pr[i];
                    idx = i;
                }
                else if(pr[i] == min_priority) {
                    if(at[i] < at[idx]) {
```

```
                    idx = i;
                }
            }
        }
    }

    if(idx != -1) {
        time += bt[idx];
        ct[idx] = time;
        tat[idx] = ct[idx] - at[idx];
        wt[idx] = tat[idx] - bt[idx];
        is_completed[idx] = 1;
        completed++;
    } else {
        time++;
    }
}

printf("\nP\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
for(int i = 0; i < n; i++) {
    printf("P%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n", i+1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i]);
}
}
```

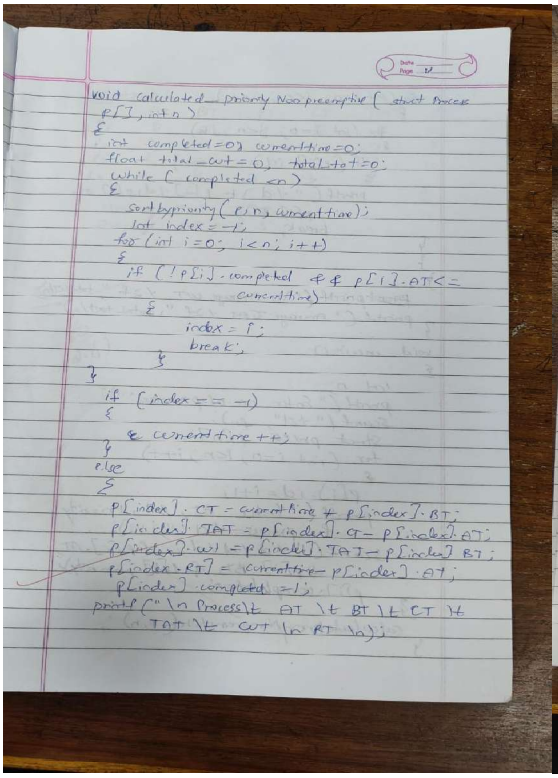
OUTPUT:

```
Enter number of processes: 4
Enter arrival time, burst time, and priority for process 1: 0 5 4
Enter arrival time, burst time, and priority for process 2: 2 4 2
Enter arrival time, burst time, and priority for process 3: 2 2 6
Enter arrival time, burst time, and priority for process 4: 4 4 3

P       AT      BT      Priority        CT      TAT     WT
P1      0       5       4               5       5       0
P2      2       4       2               9       7       3
P3      2       2       6               15      13      11
P4      4       4       3               13      9       5

Process returned 0 (0x0)   execution time : 18.703 s
Press any key to continue.
```

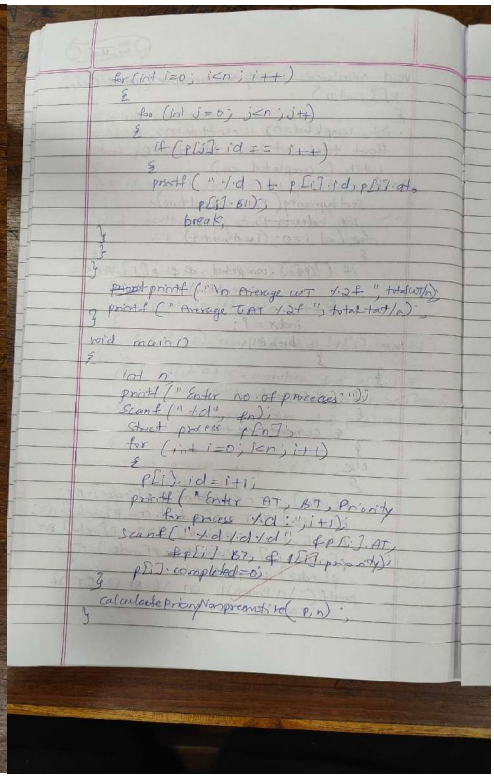


3) Priority non-preemtive.

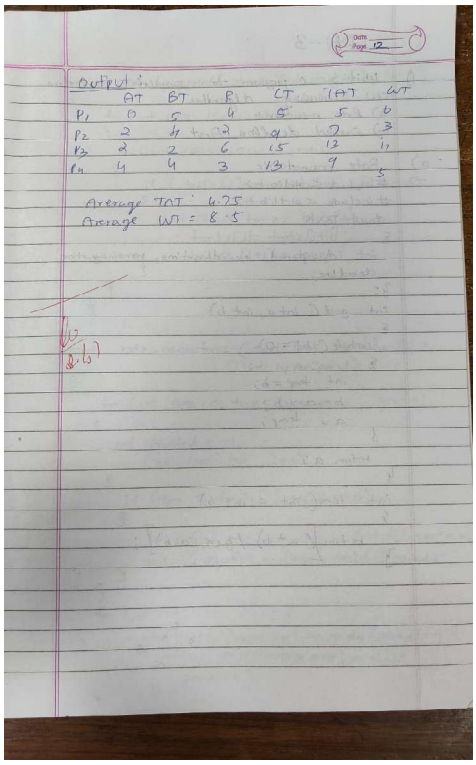
```
#include <stdio.h>
#include <stdlib.h>
struct process
{
int id, at, bt, ct, tat, wt, rt, priority, completed;
};
void sortbypriority( struct process p[], int n,
        int currenttime )
{
  for (int i=0; i<n-1 ; i++)
  {
    for (int j= i+1 ; j<n; j++)
    {
      if ( !p[i].completed && !p[j].completed)
  {
  if ( (p[j].at <= currenttime && p[j].at
      <= currenttime && p[i].priority>p[j].priority)
  || (p[j].at > currenttime && p[j].at <=
    currenttime ))
  {
    struct process temp = p[i];
      p[i] = p[j];
      p[j] = temp;
  }
  }
  }
}
```



```
void calculate_priority_Nonpreemptive ( struct Process
p[], int n )
{
  int completed=0, current_time=0;
  float total_wt=0, total_tat=0;
  while ( completed <n )
  {
    sort_by_priority ( p, n, current_time);
    int index = -1;
    for (int i=0; i<n; i++)
    {
      if ( !p[i].completed && p[i].AT<=
                current_time)
      {
        index = i;
        break;
      }
    }
    if (index == -1)
    {
      current_time++;
    }
    else
    {
      p[index].CT = current_time + p[index].BT;
      p[index].TAT = p[index].CT - p[index].AT;
      p[index].WT = p[index].TAT - p[index].BT;
      p[index].RT = current_time - p[index].AT;
      p[index].completed = 1;
      printf("\n Process\t AT \t BT \t CT \t
        TAT \t WT \n RT \n");
```



```
for (int i=0; i<n; i++)
{
  for (int j=0; j<n; j++)
  {
    if (p[j].id == i+1)
    {
      printf ("%d \t %d \t %d \t %d \t %d \t %d \n", p[i].id, p[i].at,
        p[i].BT);
      break;
    }
  }
}
printf("\n Average WT %.2f ", total_wt/n);
printf (" Average TAT %.2f ", total_tat/n);
}
void main()
{
  int n;
  printf ("Enter no. of processes:");
  scanf ("%d", &n);
  struct process p[n];
  for (int i=0; i<n; i++)
  {
    p[i].id = i+1;
    printf ("Enter AT, BT, Priority
      for process %d :", i+1);
    scanf ("%d %d %d", &p[i].AT,
      &p[i].BT, &p[i].priority);
    p[i].completed=0;
  }
  calculate_priority_Nonpreemptive( p, n);
}
```



Output :

	AT	BT	P	CT	TAT	WT
P_1	0	5	4	5	5	0
P_2	2	4	2	9	7	3
P_3	2	2	6	15	12	10
P_4	4	4	3	13	9	5

Average TAT : 4.75
Average WT = 8.5

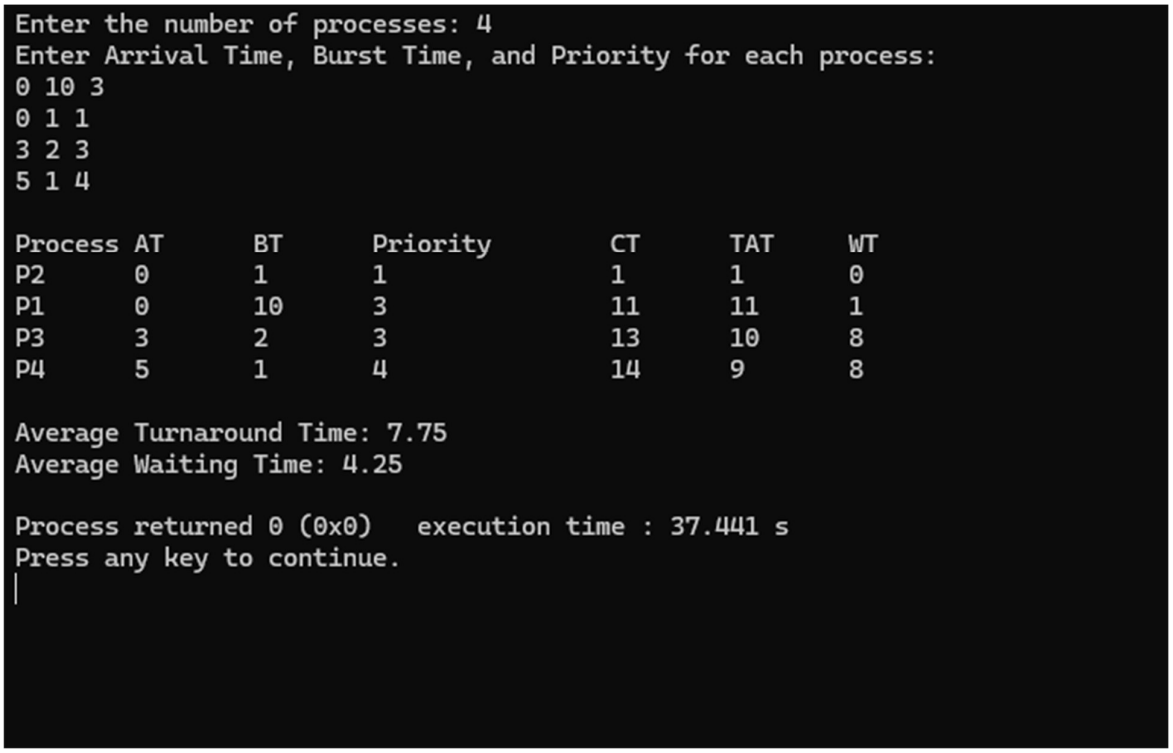
PRIORITY (PRE-EMTIVE)

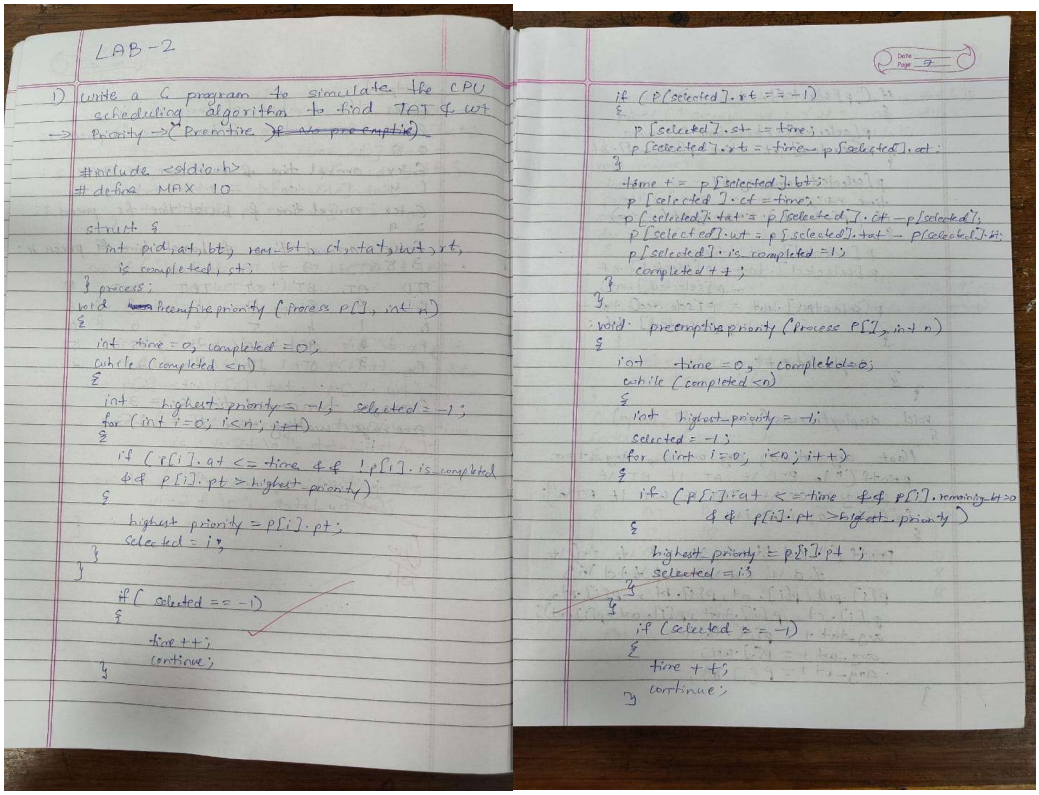
```
#include <stdio.h>

struct Process {
    int id, arrival_time, burst_time, priority, completion_time, turnaround_time, waiting_time;
};

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

void sort_by_priority(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].priority > p[j + 1].priority ||
                (p[j].priority == p[j + 1].priority && p[j].arrival_time > p[j + 1].arrival_time)) {
                swap(&p[j], &p[j + 1]);
            }
        }
    }
}

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

```
    struct Process p[n];
    printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
    }

    sort_by_priority(p, n);
    int current_time = 0;
    float total_tat = 0, total_wt = 0;

    for (int i = 0; i < n; i++) {
        if (current_time < p[i].arrival_time) {
            current_time = p[i].arrival_time;
        }
        p[i].completion_time = current_time + p[i].burst_time;
        p[i].turnaround_time = p[i].completion_time - p[i].arrival_time;
        p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
        current_time = p[i].completion_time;
        total_tat += p[i].turnaround_time;
        total_wt += p[i].waiting_time;
    }

    printf("\nProcess    AT    BT    PriorityCT    TAT    WT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d        %d    %d    %d            %d    %d    %d\n", p[i].id,
p[i].arrival_time, p[i].burst_time, p[i].priority, p[i].completion_time, p[i].turnaround_time,
p[i].waiting_time);
    }
```

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

    return 0;
}
```

OUTPUT:





LAB-2

1) Write a C program to simulate the CPU scheduling algorithm to find TAT & WT

-> Priority -> (Preemtive) (~~If No preemptive~~).

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

struct {
  int pid, at, bt, rem_bt, ct, tat, wt, rt,
    is_completed, st;
} process;
void preemtivepriority (process p[], int n)
{
  int time = 0, completed = 0;
  while (completed < n)
  {
    int highest_priority = -1, selected = -1;
    for (int i=0; i<n; i++)
    {
      if (p[i].at <= time && !p[i].is_completed
      && p[i].pt > highest_priority)
      {
        highest_priority = p[i].pt;
        selected = i;
      }
    }
    if (selected == -1)
    {
      time++;
      continue;
    }
    if (p[selected].rt == -1)
    {
      p[selected].st = time;
      p[selected].rt = time - p[selected].at;
    }
    time += p[selected].bt;
    p[selected].ct = time;
    p[selected].tat = p[selected].ct - p[selected].at;
    p[selected].wt = p[selected].tat - p[selected].bt;
    p[selected].is_completed = 1;
    completed++;
  }
}
void preemtivepriority (Process p[], int n)
{
  int time = 0, completed = 0;
  while (completed < n)
  {
    int highest_priority = -1;
    selected = -1;
    for (int i=0; i<n; i++)
    {
      if (p[i].at <= time && p[i].remaining_bt > 0
      && p[i].pt > highest_priority)
      {
        highest_priority = p[i].pt;
        selected = i;
      }
    }
    if (selected == -1)
    {
      time++;
      continue;
    }
    if (p[selected].rt == -1)
    {
      p[selected].st = time;
      p[selected].rt = time - p[selected].at;
    }
    p[selected].remaining_bt--;
    time++;
    if (p[selected].remaining_bt == 0)
    {
      p[selected].ct = time;
      p[selected].tat = p[selected].ct
        - p[selected].at;
      p[selected].wt = p[selected].tat -
        p[selected].bt;
    }
    completed++;
  }
}
void displayProcesses (process p[], int n)
{
  float avg_tat = 0, avg_wt = 0, avg_rt = 0;
  printf("\n PID \t AT \t BT \t
  Priority \t CT \t TAT \t WT \t RT \n");
  for (int i=0; i<n; i++)
  {
    printf("\n %d \t %d \t %d \t %d \t %d \t
    %d \t %d \t %d \n",
    p[i].pid, p[i].at, p[i].bt, p[i].pt,
    p[i].ct, p[i].tat, p[i].wt, p[i].rt);
    avg_tat += p[i].tat;
    avg_wt += p[i].wt;
    avg_rt += p[i].rt;
  }
  printf("\n Average tat : %f ", avg_tat/n);
  printf("\n Average wt : %f ", avg_wt/n);
  printf("\n Average rt : %f ", avg_rt/n);
}
void main()
{
  int n;
  printf("\n Enter no. of processes: ");
  scanf("%d", &n);
  for (int i=0; i<n; i++)
  {
    p[i].pid = i+1;
    printf("\n Enter at, bt, priority for process %d: ", p[i].pid);
    scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pt);
    p[i].completed = 0;
  }
  preemtivepriority(n);
}
```

Output:

Enter no. of process: 4

Enter pid, AT, BT, PT

P	AT	BT	P	CT	TAT	WT
1	0	10	3	11	11	1
2	0	1	1	1	1	0
3	3	2	3	13	10	8
4	5	1	4	14	9	6

Average WT : 3.75

Average TAT : 7.75

PROGRAM-2

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.

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

typedef struct {
    int id;
    int burst_time;
    int arrival_time;
    int queue; // 1 for system process (RR), 2 for user process (FCFS)
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int response_time;
} Process;

void round_robin(Process processes[], int n, int tq, int *time) {
    int done, i;
    do {
        done = 1;
        for (i = 0; i < n; i++) {
            if (processes[i].queue == 1 && processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time == processes[i].burst_time)
                    processes[i].response_time = *time - processes[i].arrival_time;
```

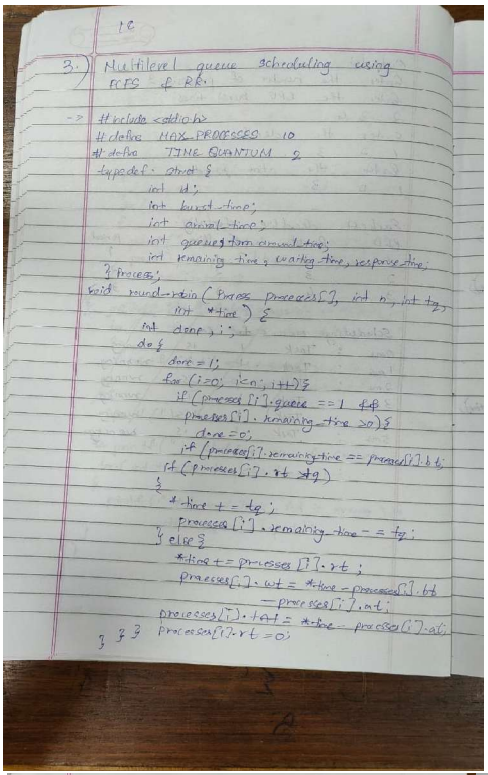
```
            if (processes[i].remaining_time > tq) {
                *time += tq;
                processes[i].remaining_time -= tq;
            } else {
                *time += processes[i].remaining_time;
                processes[i].waiting_time = *time - processes[i].burst_time - processes[i].arrival_time;
                processes[i].turnaround_time = *time - processes[i].arrival_time;
                processes[i].remaining_time = 0;
            }
        }
    }
  } while (!done);
}

void fcfs(Process processes[], int n, int *time) {
  for (int i = 0; i < n; i++) {
    if (processes[i].queue == 2) {
      if (*time < processes[i].arrival_time)
        *time = processes[i].arrival_time;
      processes[i].response_time = *time - processes[i].arrival_time;
      processes[i].waiting_time = *time - processes[i].arrival_time;
      *time += processes[i].burst_time;
      processes[i].turnaround_time = *time - processes[i].arrival_time;
    }
  }
}

void calculate_average(Process processes[], int n) {
  float total_waiting = 0, total_turnaround = 0, total_response = 0;
```

```
    for (int i = 0; i < n; i++) {
        total_waiting += processes[i].waiting_time;
        total_turnaround += processes[i].turnaround_time;
        total_response += processes[i].response_time;
    }
    printf("\nAverage Waiting Time: %.2f", total_waiting / n);
    printf("\nAverage Turn Around Time: %.2f", total_turnaround / n);
    printf("\nAverage Response Time: %.2f", total_response / n);
    printf("\nThroughput: %.2f\n", n / (total_turnaround / n));
}

int main() {
    int n, time = 0;
    Process processes[MAX_PROCESSES];

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

    printf("Queue 1 is system process\nQueue 2 is User Process\n");

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue);
        processes[i].id = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].response_time = 0;
```

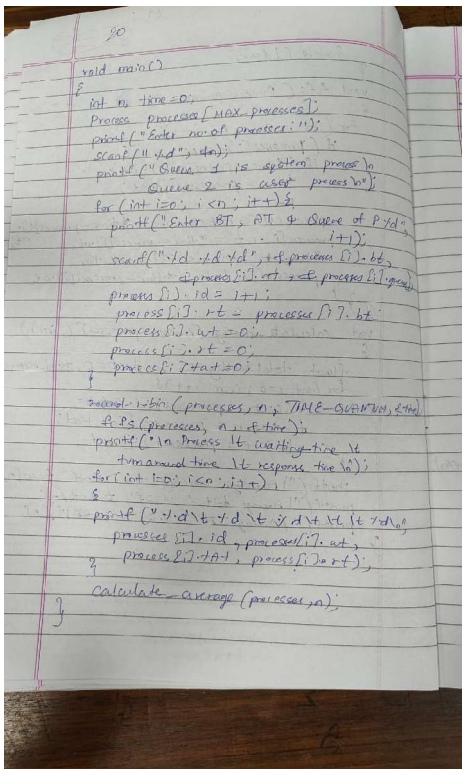
```
    }

    round_robin(processes, n, TIME_QUANTUM, &time);
    fcfs(processes, n, &time);

    printf("\nProcess\tWaiting Time\tTurn Around Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].waiting_time,
processes[i].turnaround_time, processes[i].response_time);
    }
  calculate_average(processes, n);
}
```

OUTPUT

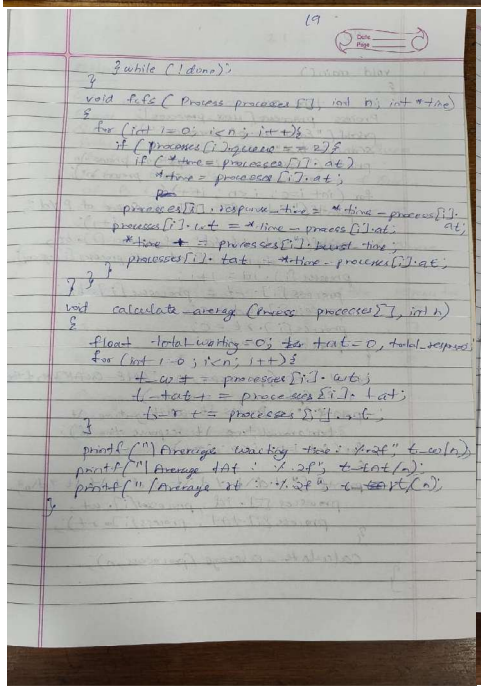
```
Enter number of processes: 4
Queue 1 is system process
Queue 2 is User Process
Enter Burst Time, Arrival Time and Queue of P1: 2 0 1
Enter Burst Time, Arrival Time and Queue of P2: 1 0 2
Enter Burst Time, Arrival Time and Queue of P3: 5 0 1
Enter Burst Time, Arrival Time and Queue of P4: 3 0 2

Process Waiting Time    Turn Around Time        Response Time
1       0               2                       0
2       7               8                       7
3       2               7                       2
4       8               11                      8

Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.57

Process returned 0 (0x0)   execution time : 71.241 s
Press any key to continue.
```



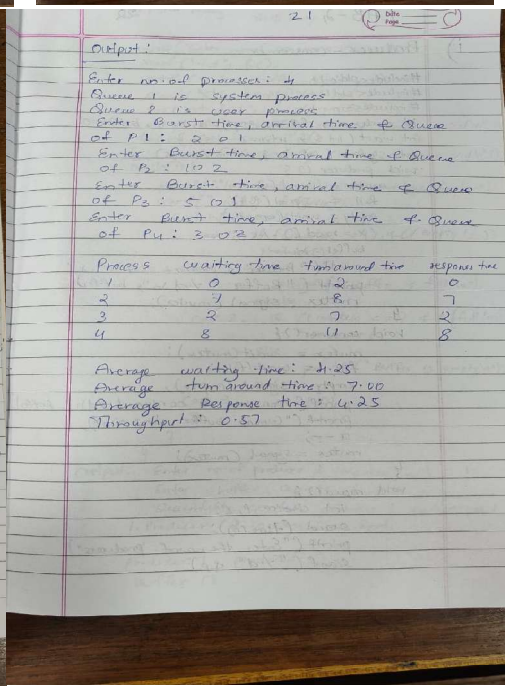
```
3.) Multilevel queue scheduling using FCFS & RR:

-> #include <stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUANTUM 2
typedef struct {
    int id;
    int burst_time;
    int arrival_time;
    int queue_type; // 1 for system, 2 for user
    int remaining_time, waiting_time, response_time;
} Process;
void round_robin(Process processes[], int n, int tq,
                 int *time) {
    int done, i;
    do {
        done = 1;
        for (i=0; i<n; i++) {
            if (processes[i].queue == 1 &&
                processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time == processes[i].bt
                    && processes[i].rt > tq)
                {
                    *time += tq;
                    processes[i].remaining_time -= tq;
                } else {
                    *time += processes[i].rt;
                    processes[i].wt = *time - processes[i].bt
                                      - processes[i].at;
                    processes[i].tat = *time - processes[i].at;
                    processes[i].rt = 0;
```



```
void main()
{
    int n, time=0;
    Process processes[MAX_processes];
    printf("Enter no. of processes: ");
    scanf("%d", &n);
    printf("Queue 1 is system process\n");
    printf("Queue 2 is user process\n");
    for (int i=0; i<n; i++) {
        printf("Enter BT, AT & Queue of P%d",
                                    i+1);
        scanf("%d %d %d", &processes[i].bt,
                &processes[i].at, &processes[i].queue);
        processes[i].id = i+1;
        processes[i].rt = processes[i].bt;
        processes[i].wt = 0;
        processes[i].rt = 0;
        processes[i].tat = 0;
    }
    round_robin(processes, n, TIME_QUANTUM, &time);
    fcfs(processes, n, &time);
    printf("\n Process \t waiting time \t
        turnaround time \t response time \n");
    for(int i=0; i<n; i++)
    {
        printf("%d\t %d\t %d\t %d\t %d\n",
            processes[i].id, processes[i].wt,
            processes[i].tat, processes[i].rt);
    }
    calculate_average(processes,n);
}
```



```
}while (!done);
}
void fcfs(Process processes[], int n, int *time)
{
    for (int i=0; i<n; i++){
        if (processes[i].queue == 2){
            if (*time < processes[i].at)
                *time = processes[i].at;
            processes[i].response_time = *time - processes[i].
            processes[i].wt = *time - processes[i].at;
            *time += processes[i].burst_time;
            processes[i].tat = *time - processes[i].at;
        }
    }
}
void calculate_average(Process processes[], int n)
{
    float total_waiting=0, tat=0, total_response
    for (int i=0; i<n; i++){
        t_wt += processes[i].wt;
        t_tat += processes[i].tat;
        t_rt += processes[i].rt;
    }
    printf("\nAverage waiting time: %f", t_wt/n);
    printf("\nAverage tat: %.2f", t_tat/n);
    printf("\nAverage rt: %.2f", t_rt/n);
}
```



```
Output:

Enter no. of processes: 4
Queue 1 is system process
Queue 2 is user process
Enter Burst time, arrival time & Queue
of P1: 2 0 1
Enter Burst time, arrival time & Queue
of P2: 1 0 2
Enter Burst time, arrival time & Queue
of P3: 5 0 1
Enter Burst time, arrival time & Queue
of P4: 3 0 2
```

Process	waiting time	turnaround time	response time
1	0	2	0
2	7	8	7
3	2	7	2
4	8	11	8

```
Average waiting time: 4.25
Average turnaround time: 7.00
Average response time: 4.25
Throughput: 0.57
```

PROGRAM 3

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

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

RATE MONOTONIC

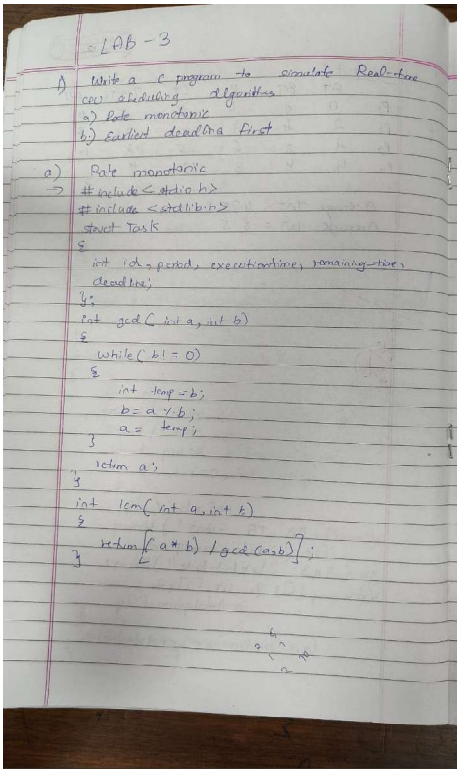
```
#include <stdio.h>
#include <stdlib.h>
#include<math.h>

struct Task
{
  int id, period, execution_time, remaining_time, deadline;
};

int gcd(int a, int b)
{
  while (b != 0)
  {
   int temp = b;
   b = a % b;
   a = temp;
  }
   return a;
}

int lcm(int a, int b)
{
 return (a * b) / gcd(a, b);
}

void sort_by_period(struct Task tasks[], int n)
{
  for (int i = 0; i < n - 1; i++)
  {
   for (int j = 0; j < n - i - 1; j++)
   {
    if (tasks[j].period > tasks[j + 1].period)
    {
     struct Task temp = tasks[j];
     tasks[j] = tasks[j + 1];
     tasks[j + 1] = temp;
     }  }
```

```
  }
}

void rate_monotonic(struct Task tasks[], int n, int hyper_period)
{
 for (int time = 0; time < hyper_period; time++)
 {
  int selected = -1;
  for (int i = 0; i < n; i++)
  {
   if (time % tasks[i].period == 0)
   {
    tasks[i].remaining_time = tasks[i].execution_time;
    tasks[i].deadline = time + tasks[i].period;
    }
  }
  for (int i = 0; i < n; i++)
  {
   if (tasks[i].remaining_time > 0 && (selected == -1 || tasks[i].period <
tasks[selected].period))
   {
    selected = i;
   }
  }
  if (selected != -1)
  {
    tasks[selected].remaining_time--;
    printf("Time %d: Task %d\n", time, tasks[selected].id);
  }
  else
  {
  printf("Time %d: Idle\n", time);
  }
  }
}

void main()
{
   int n, hyper_period;
   printf("Enter the number of tasks: ");
   scanf("%d", &n);
   double p=pow(2,(1.0/n))double c1= n*(p-1);
   struct Task tasks[n];
```

```
for (int i = 0; i < n; i++)
{
printf("Enter execution time and period for Task %d: ", i + 1);
scanf("%d %d", &tasks[i].execution_time, &tasks[i].period);
tasks[i].id = i + 1;
tasks[i].remaining_time = 0;
tasks[i].deadline = 0;
}

hyper_period = tasks[0].period;
for (int i = 1; i < n; i++)
{
hyper_period = lcm(hyper_period, tasks[i].period);
}
double c2=0.0;
for (int i=0;i<n;i++)
{
c2=c2+((double)(tasks[i].execution_time)/(tasks[i].period));
}
if(c2<=c1)
{
sort_by_period(tasks, n);
rate_monotonic(tasks, n, hyper_period);
}
else
{
printf("%lf <= %lf ",c2,c1);
printf("\nCondition not satisfied\n");
}
}
```

OUTPUT

```
Enter the number of tasks: 3
Enter execution time and period for Task 1: 3 3
Enter execution time and period for Task 2: 6 4
Enter execution time and period for Task 3: 8 5
4.100000 <= 0.779763
Condition not satisfied

Process returned 0 (0x0)   execution time : 7.308 s
Press any key to continue.
```

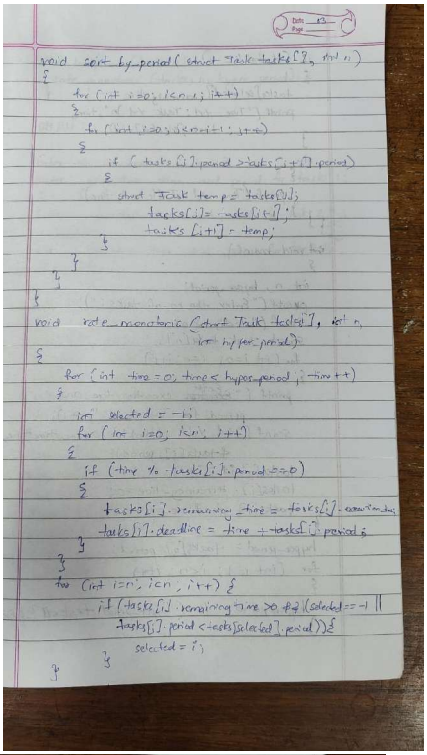


LAB - 3

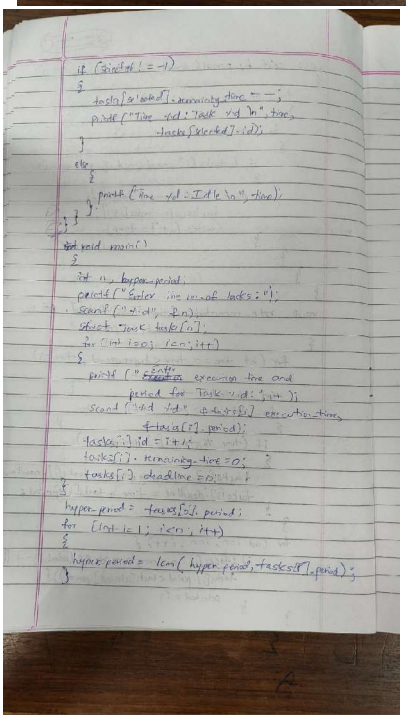
A) Write a C program to simulate Real-time CPU scheduling algorithms
a) Rate monotonic
b) Earliest deadline first

a) Rate monotonic

```
#include <stdio.h>
#include <stdlib.h>
struct Task
{
  int id, period, executiontime, remaining_time,
  deadline;
};
int gcd( int a, int b)
{
  while( b != 0)
  {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
int lcm( int a, int b)
{
  return (a * b) / gcd(a,b);
}
```



```
void sort_by_period( struct Task tasks[], int n)
{
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<n-i-1; j++)
    {
      if ( tasks[j].period > tasks[j+1].period)
      {
        struct Task temp = tasks[j];
        tasks[j] = tasks[j+1];
        tasks[j+1] = temp;
      }
    }
  }
}
void rate_monotonic (struct Task tasks[], int n,
                     int hyper_period)
{
  for (int time = 0; time < hyper_period; time++)
  {
    int selected = -1;
    for (int i=0; i<n; i++)
    {
      if (time % tasks[i].period == 0)
      {
        tasks[i].remaining_time = tasks[i].executiontime;
        tasks[i].deadline = time + tasks[i].period;
      }
    }
    for (int i=0; i<n; i++) {
      if (tasks[i].remaining_time > 0 && (selected == -1 ||
        tasks[i].period < tasks[selected].period)){
        selected = i;
      }
    }
```



```
    if (selected != -1)
    {
      tasks[selected].remaining_time--;
      printf("Time %d: Task %d\n", time,
             tasks[selected].id);
    }
    else
    {
      printf("Time %d: Idle\n", time);
    }
  }
}
void main()
{
  int n, hyper_period;
  printf("Enter no. of tasks: ");
  scanf("%d", &n);
  struct Task tasks[n];
  for (int i=0; i<n; i++)
  {
    printf("Enter execution time and
           period for Task %d: ", i+1);
    scanf("%d %d", &tasks[i].executiontime,
          &tasks[i].period);
    tasks[i].id = i+1;
    tasks[i].remaining_time = 0;
    tasks[i].deadline = 0;
  }
  hyper_period = tasks[0].period;
  for (int i=1; i<n; i++)
  {
    hyper_period = lcm(hyper_period, tasks[i].period);
  }
```

EARLIEST DEADLINE

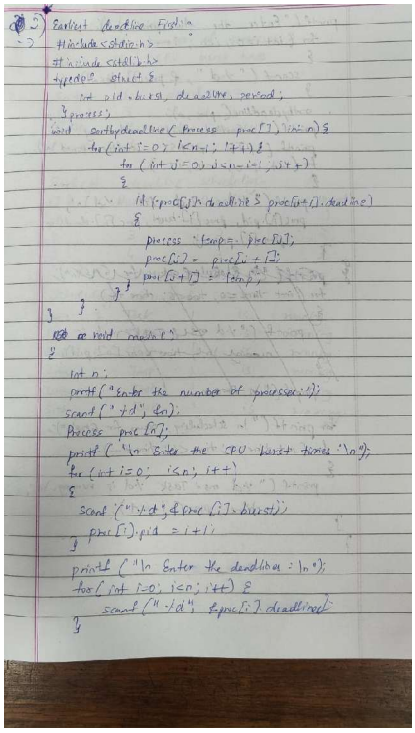
```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int pid;
    int burst;
    int deadline;
    int period;
} Process;

void sortByDeadline(Process proc[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].deadline > proc[j + 1].deadline) {
                Process temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }
}

void main() {
    int n;
    printf("Enter the number of processes:");
    scanf("%d", &n);

    Process proc[n];
```

```
    printf("\nEnter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &proc[i].burst);
        proc[i].pid = i + 1;
    }

    printf("\nEnter the deadlines:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &proc[i].deadline);
    }

    printf("\nEnter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &proc[i].period);
    }

    sortByDeadline(proc, n);

    printf("\nEarliest Deadline Scheduling:\n");
    printf("PID    Burst    Deadline    Period\n");
    for (int i = 0; i < n; i++) {
        printf("%d      %d        %d          %d\n", proc[i].pid, proc[i].burst, proc[i].deadline,
proc[i].period);
    }
    printf("\nScheduling occurs for 6 ms\n");
    for (int time = 0; time < 6; time++) {
        printf("%dms : Task %d is running.\n", time, proc[0].pid);
    }
```

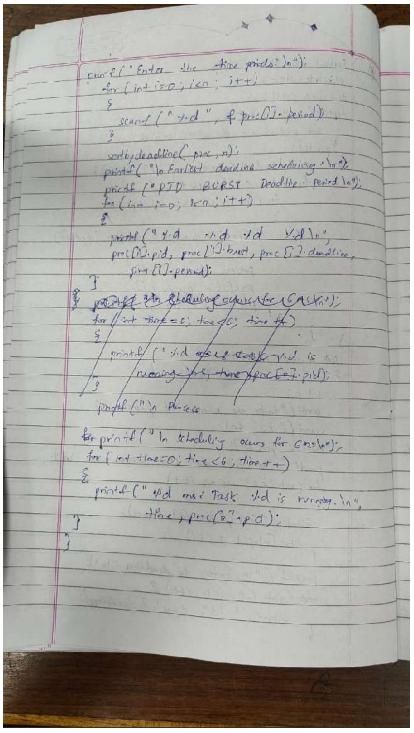
}

OUTPUT

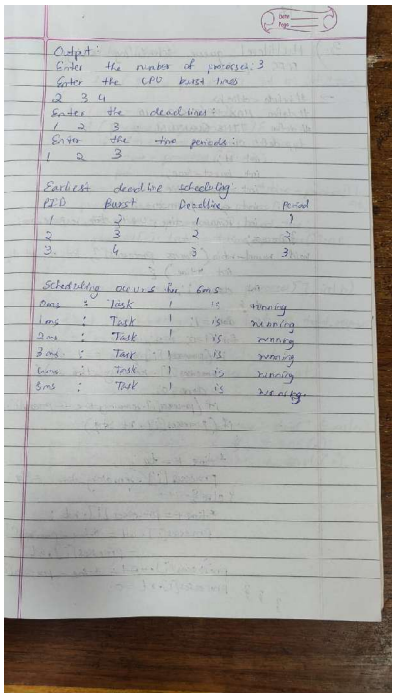
```
Enter the number of processes:3

Enter the CPU burst times:
2 3 5

Enter the deadlines:
1 2 3

Enter the time periods:
1 2 3

Earliest Deadline Scheduling:
PID    Burst    Deadline    Period
1      2        1           1
2      3        2           2
3      5        3           3

Scheduling occurs for 6 ms
0ms : Task 1 is running.
1ms : Task 1 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 1 is running.
5ms : Task 1 is running.

Process returned 6 (0x6)   execution time : 17.130 s
Press any key to continue.

Process returned 0 (0x0)   execution time : 28.063 s
Press any key to continue.
```



```
5) Earliest deadline First :-
#include<stdio.h>
#include<stdlib.h>
typedef struct {
    int pid, burst, deadline, period;
} process;
void sortbydeadline(process proc[], int n){
    for(int i=0; i<n-1; i++){
        for (int j=0; j<n-i-1; j++)
        {
            if(proc[j].deadline > proc[j+1].deadline)
            {
                process temp = proc[j];
                proc[j] = proc[j+1];
                proc[j+1] = temp;
            }
        }
}
void main()
{
    int n;
    printf("Enter the number of processes:");
    scanf("%d",&n);
    Process proc[n];
    printf("\n Enter the CPU burst times:\n");
    for(int i=0; i<n; i++)
    {
        scanf("%d",&proc[i].burst);
        proc[i].pid = i+1;
    }
    printf("\n Enter the deadlines:\n");
    for(int i=0; i<n; i++) {
        scanf("%d",&proc[i].deadline);
    }
```



```
    printf("Enter the time periods:\n");
    for(int i=0; i<n; i++)
    {
        scanf("%d",&proc[i].period);
    }
    sortbydeadline(proc,n);
    printf("\n Earliest deadline scheduling:\n");
    printf("PID BURST Deadline Period\n");
    for(int i=0; i<n; i++)
    {
        printf("%d %d %d %d\n",
        proc[i].pid, proc[i].burst, proc[i].deadline,
        proc[i].period);
    }
    printf("\n Scheduling occurs for (CPU\n");
    for(int time=0; time<6; time++)
    {
        printf("%d ms: Task %d is
            running.\n", time, proc[i].pid);
    }
    printf("\n Process
    for printf("\n Scheduling occurs for (ms)\n");
    for(int time=0; time<6; time++)
    {
        printf("%d ms: Task %d is running.\n",
            time, proc[i].pid);
    }
}
```



```
Output:
Enter the number of processes: 3
Enter the CPU burst times
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

Earliest deadline scheduling
PID   Burst   Deadline   Period
1     2       1          1
2     3       2          2
3     4       3          3

Scheduling occurs for ms
0ms : Task 1 is running
1ms : Task 1 is running
2ms : Task 1 is running
3ms : Task 1 is running
4ms : Task 1 is running
5ms : Task 1 is running.
```

PROGRAM 4

Write a C program to simulate:
a) Producer-Consumer problem using semaphores.
b) Dining-Philosopher's problem

PRODUCER-CONSUMER USING SEMAPHORES

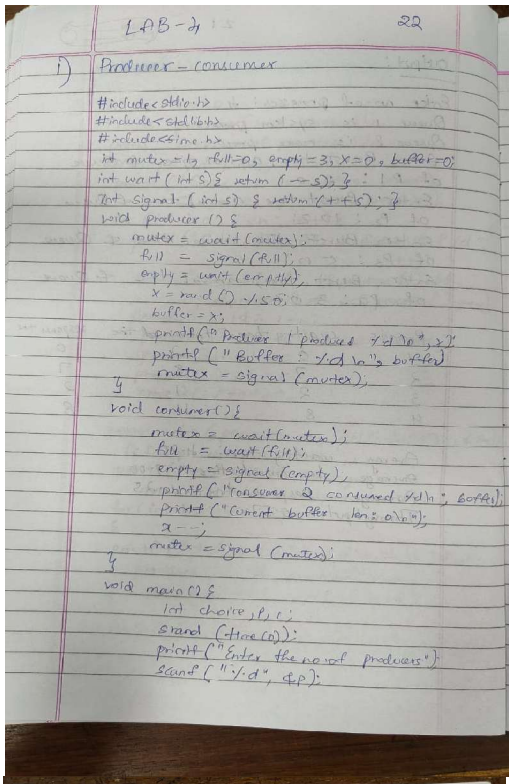
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int mutex = 1, full = 0, empty = 3, x = 0, buffer = 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 = rand() % 50;
   buffer = x;
   printf("Producer 1 produced %d\n", x);
   printf("Buffer:%d\n", buffer);
   mutex = signal(mutex);
}
```

```
void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("Consumer 2 consumed %d\n", buffer);
    printf("Current buffer len: 0\n");
    x--;
    mutex = signal(mutex);
}

void main() {
    int choice, p, c;

    srand(time(0));
    printf("Enter the number of Producers:");
    scanf("%d", &p);
    printf("Enter the number of Consumers:");
    scanf("%d", &c);
    printf("Enter buffer capacity:");
    scanf("%d", &empty);

    for (int i = 1; i <= p; i++)
        printf("Successfully created producer %d\n", i);
    for (int i = 1; i <= c; i++)
        printf("Successfully created consumer %d\n", i);

    while (1) {
        printf("\n1.Producer\n2.Consumer\n3.Exit\n");
```

```
        scanf("%d", &choice);
        switch (choice) {
            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);
        }
    }
}
```

OUTPUT

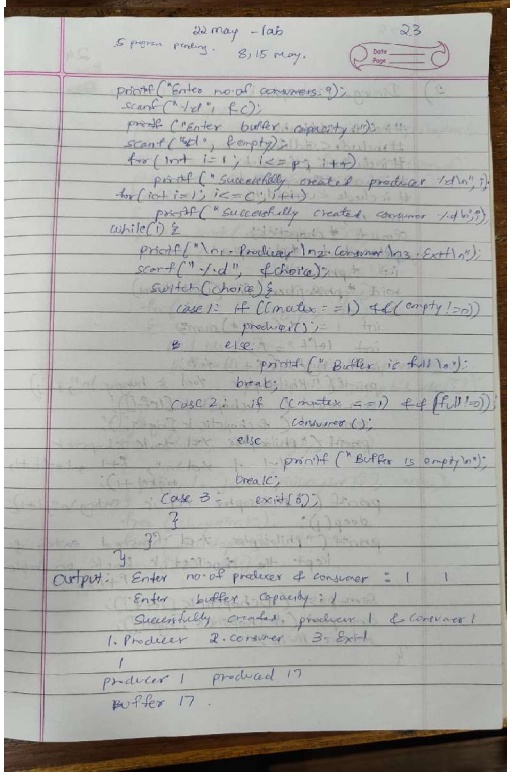
```
Enter the number of Producers:1
Enter the number of Consumers:1
Enter buffer capacity:1
Successfully created producer 1
Successfully created consumer 1

1.Producer
2.Consumer
3.Exit
1
Producer 1 produced 17
Buffer:17

1.Producer
2.Consumer
3.Exit
2
Consumer 2 consumed 17
Current buffer len: 0

1.Producer
2.Consumer
3.Exit
1
Producer 1 produced 16
Buffer:16

1.Producer
2.Consumer
3.Exit
2
Consumer 2 consumed 16
Current buffer len: 0

1.Producer
2.Consumer
3.Exit
```



LAB - 2 22

1) Producer - consumer

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int mutex = 1, full=0, empty = 3, x=0, buffer=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 = rand() %150;
    buffer = x;
    printf("Producer 1 produced %d \n", x);
    printf("Buffer : %d \n", buffer);
    mutex = signal(mutex);
}
void consumer(){
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("consumer 2 consumed %d\n", buffer);
    printf("Current buffer len: 0\n");
    x--;
    mutex = signal(mutex);
}
void main(){
    int choice, p, c;
    srand(time(0));
    printf("Enter the no of producers");
    scanf("%d", &p);
```



22 may - lab 23

5 program pending. 8, 15 may.

Date
Page

```
printf("Enter no of consumers: ");
scanf("%d", &c);
printf("Enter buffer capacity:\n");
scanf("%d", &empty);
for(int i=1; i<=p; i++)
    printf("Successfully created producer %d\n", i);
for(int i=1; i<=c; i++)
    printf("Successfully created consumer %d\n");
while(1){
    printf("\n1. Producer\n2. Consumer\n3. Exit\n");
    scanf("%d", &choice);
    switch(choice){
        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);
    }
}
}
```

Output: Enter no of producer & consumer : 1 1

Enter buffer capacity : 1

Successfully created producer 1 & Consumer 1

1. Producer 2. consumer 3. Exit

1

producer 1 produced 17

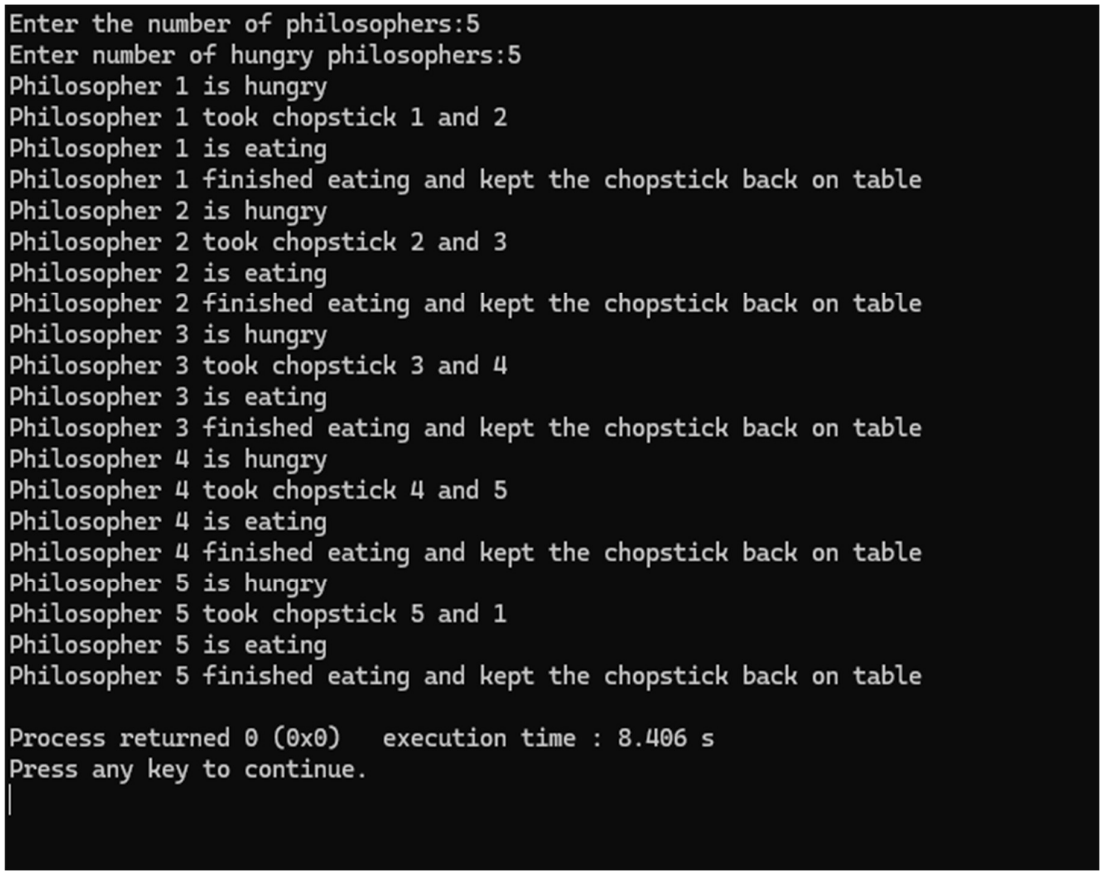
Buffer 17.

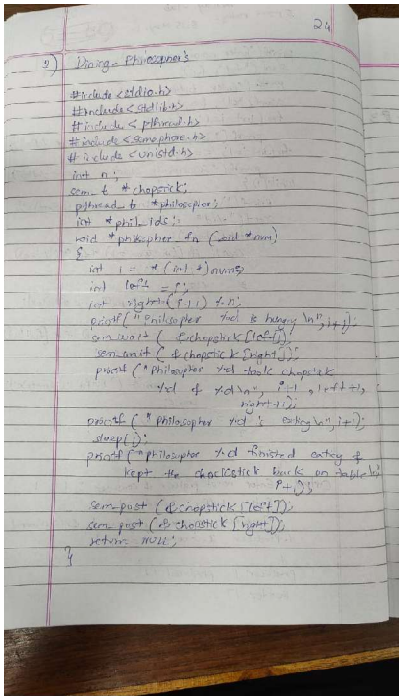
DINING PHOLOSOPHERS

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

int n;
sem_t *chopstick;
pthread_t *philosopher;
int *phil_ids;

void* philosopher_fn(void* num) {
    int i = *(int*)num;
    int left = i;
    int right = (i + 1) % n;
    printf("Philosopher %d is hungry\n", i + 1);
    sem_wait(&chopstick[left]);
    sem_wait(&chopstick[right]);
    printf("Philosopher %d took chopstick %d and %d\n", i + 1, left + 1, right + 1);
    printf("Philosopher %d is eating\n", i + 1);
    sleep(1);
    printf("Philosopher %d finished eating and kept the chopstick back on table\n", i + 1);
    sem_post(&chopstick[left]);
    sem_post(&chopstick[right]);
    return NULL;
}

int main() {
    int hungry, i;
    printf("Enter the number of philosophers:");
    scanf("%d", &n);
    chopstick = malloc(n * sizeof(sem_t));
    philosopher = malloc(n * sizeof(pthread_t));
    phil_ids = malloc(n * sizeof(int));
    for (i = 0; i < n; i++) sem_init(&chopstick[i], 0, 1);
    printf("Enter number of hungry philosophers:");
    scanf("%d", &hungry);
    for (i = 0; i < hungry; i++) {
        phil_ids[i] = i;
        pthread_create(&philosopher[i], NULL, philosopher_fn, &phil_ids[i]);
        sleep(1);
    }
```

```
    for (i = 0; i < hungry; i++) {
        pthread_join(philosopher[i], NULL);
    }
    free(chopstick);
    free(philosopher);
    free(phil_ids);
    return 0;
}
```

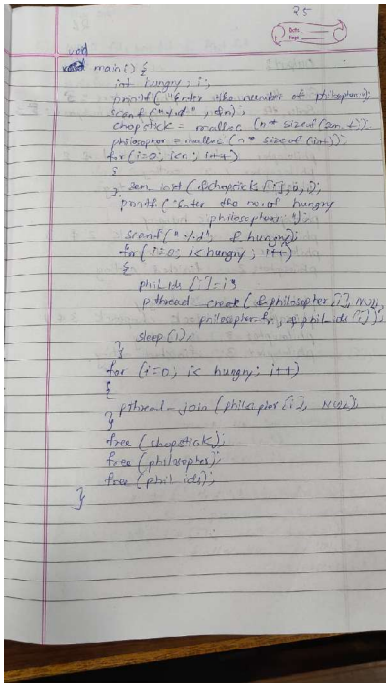
OUTPUT



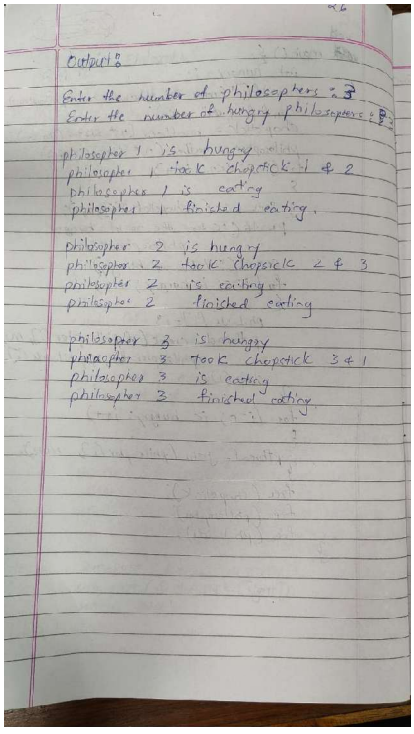


2) Dining - Philosophers

```
#include <stdio.h>
#include<stdlib>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
int n;
sem_t *chopstick;
pthread_t *philosopher;
int *phil_ids;
void *philosopher_fn (void *num)
{
  int i = *(int *)num;
  int left = i;
  int right = (i+1) % n;
  printf("Philosopher %d is hungry \n",i+1);
  sem_wait( &chopstick[left]);
  sem_wait( &chopstick[right]);
  printf("Philosopher %d took chopstick
          %d & %d\n", i+1, left+1,
                right+1);
  printf("Philosopher %d is eating\n",i+1);
  sleep(1);
  printf("Philosopher %d finished eating &
      kept the chopsticks back on table\n",
                i+1);
  sem_post (&chopstick[left]);
  sem_post (&chopstick[right]);
  return NULL;
}
```



```
void main() {
    int hungry, i;
    printf("Enter the number of philosophers");
    scanf("%d", &n);
    chopstick = malloc (n * sizeof(sem_t));
    philosopher = malloc (n * sizeof(pthread_t));
    for(i=0; i<n; i++)
    {
      sem_init(&chopsticks[i],0,1);
      printf("Enter the no. of hungry
                philosophers: ");
      scanf("%d", &hungry);
      for(i=0; i<hungry; i++)
      {
        phil_ids[i]=i;
        pthread_create(&philosopher[i], NULL,
                philosopher_fn, &phil_ids[i]);
        sleep(1);
      }
      for(i=0; i<hungry; i++)
      {
        pthread_join(philosopher[i], NULL);
      }
      free(chopstick);
      free(philosopher);
      free(phil_ids);
}
```



Output:

```
Enter the number of philosophers : 5
Enter the number of hungry philosophers : 3

philosopher 1 is hungry
philosopher 1 took chopstick 1 & 2
philosopher 1 is eating
philosopher 1 finished eating.

philosopher 2 is hungry
philosopher 2 took chopstick 2 & 3
philosopher 2 is eating
philosopher 2 finished eating

philosopher 3 is hungry
philosopher 3 took chopstick 3 & 4
philosopher 3 is eating
philosopher 3 finished eating
```

PROGRAM 5

Write a C program to simulate:
a) Bankers' algorithm for the purpose of deadlock avoidance.
b) Deadlock Detection

BANKER'S ALGORITHM

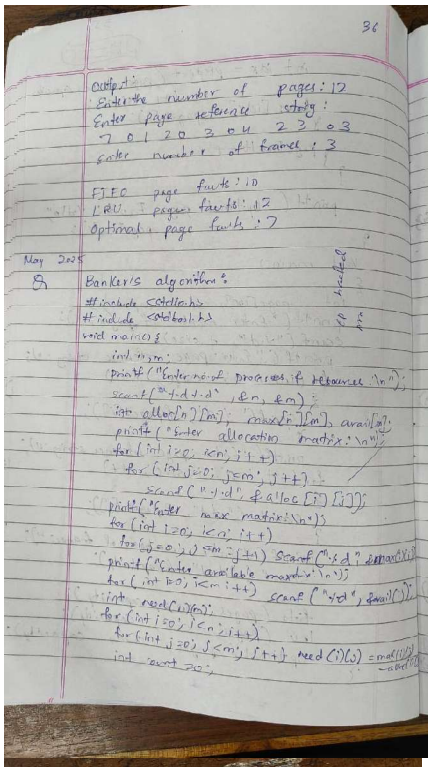
```
#include <stdio.h>
#include <stdbool.h>

void main() {
    int n, m;
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);

    int alloc[n][m], max[n][m], avail[m];
    printf("Enter allocation matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);

    printf("Enter max matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &max[i][j]);

    printf("Enter available matrix:\n");
    for (int i = 0; i < m; i++)
        scanf("%d", &avail[i]);

    int finish[n];
```

```
int safeSeq[n];
for (int i = 0; i < n; i++) finish[i] = 0;

int need[n][m];
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];

int count = 0;
while (count < n) {
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            int j;
            for (j = 0; j < m; j++)
                if (need[i][j] > avail[j])
                    break;
            if (j == m) {
                for (int k = 0; k < m; k++)
                    avail[k] += alloc[i][k];
                safeSeq[count++] = i;
                finish[i] = 1;
                found = true;
            }
        }
    }
    if (!found) break;
}
```

```
    if (count == n) {
        printf("System is in safe state.\n");
        printf("Safe sequence is: ");
        for (int i = 0; i < n; i++) {
            printf("P%d", safeSeq[i]);
            if (i != n - 1) printf(" -> ");
        }
        printf("\n");
    } else {
        printf("System is not in a safe state.\n");
    }
}
```

OUTPUT

```
Enter number of processes and resources:
5
3
Enter allocation matrix:
0 1 0 2 0 0 3 0 2 2 1 1  0 0 2
Enter max matrix:
7 5 3 3 2 2 9 0 2 2 2 2 4 3 3
Enter available matrix:
3 3 2
System is in safe state.
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2

Process returned 0 (0x0)   execution time : 74.057 s
Press any key to continue.
```



36

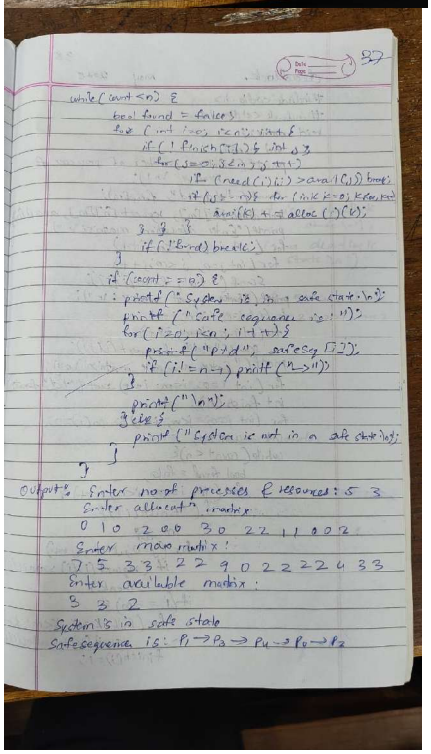
Output:
Enter the number of pages: 12
Enter page reference string:
7 0 1 2 0 3 0 4 2 3 0 3
Enter number of frames: 3

FIFO page faults: 10
LRU pages faults: 12
Optimal page faults: 7

May 2025

Bankers algorithm:

```
#include <stdio.h>
#include <stdbool.h>
void main() {
    int n, m;
    printf("Enter no. of process, & resources \n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], max[n][m], avail[m];
    printf("Enter allocation matrix: \n");
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter max matrix: \n");
    for (int i=0; i<n; i++)
        for (j=0; j<m; j++) scanf("%d", &max[i][j]);
    printf("Enter available matrix: \n");
    for (int i=0; i<m; i++) scanf("%d", &avail[i]);
    int need[n][m];
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++) need[i][j] = max[i][j] - alloc[i][j];
    int count = 0;
```



```
while (count < n) {
    bool found = false;
    for (int i=0; i<n; i++) {
        if (!finish[i]) {
            for (j=0; j<m; j++)
                if (need[i][j] > avail[j]) break;
            if (j == m) {
                for (int k=0; k<m; k++)
                    avail[k] += alloc[i][k];
                finish[i] = true;
                found = true;
                count++;
                printf("P%d ", i);
            }
        }
    }
    if (!found) break;
}
if (count == n) {
    printf("System is in safe state.\n");
    printf("Safe sequence is: ");
    for (i=0; i<n; i++) {
        printf("P%d ", safeSeq[i]);
        if (i != n-1) printf("-> ");
    }
    printf("\n");
} else {
    printf("System is not in a safe state.\n");
}
}
```

Output: Enter no. of processes & resources: 5 3
Enter allocation matrix:
0 1 0 2 0 0 3 0 2 2 1 1 0 0 2
Enter max matrix:
7 5 3 3 2 2 9 0 2 2 2 2 4 3 3
Enter available matrix:
3 3 2
System is in safe state
Safe sequence is: $P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0 \rightarrow P_2$

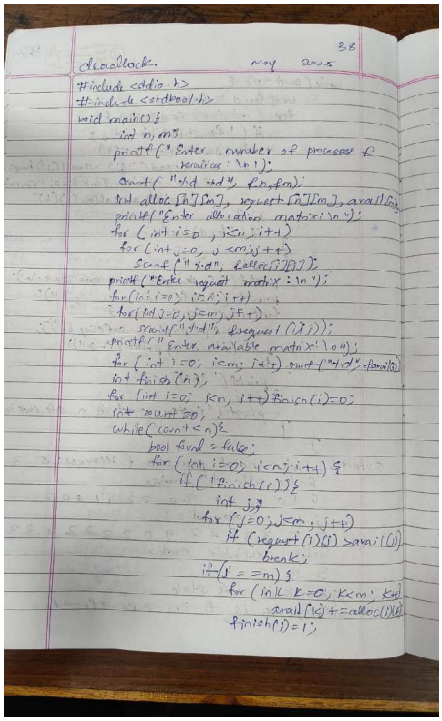
DEADLOCK DETECTION

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

void main() {
    int n, m;
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);

    int alloc[n][m], request[n][m], avail[m];
    printf("Enter allocation matrix:\n");
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);

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

    printf("Enter available matrix:\n");
    for (int i = 0; i < m; i++)
        scanf("%d", &avail[i]);

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

```
    int count = 0;
    while (count < n) {
        bool found = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < m; j++)
                    if (request[i][j] > avail[j])
                        break;
                if (j == m) {
                    for (int k = 0; k < m; k++)
                        avail[k] += alloc[i][k];
                    finish[i] = 1;
                    printf("Process %d can finish.\n", i);
                    count++;
                    found = true;
                }
            }
        }
        if (!found) break;
    }

    if (count == n)
        printf("System is not in a deadlock state.\n");
    else
        printf("System is in a deadlock state.\n");
}
```

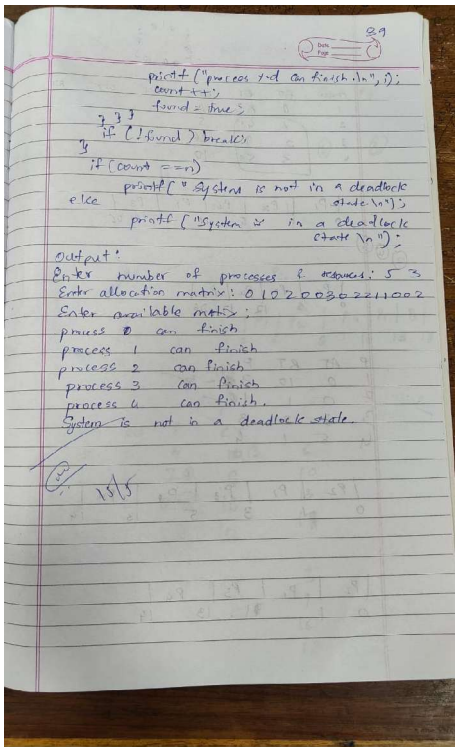
OUTPUT

```
Enter the number of philosophers:5
Enter number of hungry philosophers:5
Philosopher 1 is hungry
Philosopher 1 took chopstick 1 and 2
Philosopher 1 is eating
Philosopher 1 finished eating and kept the chopstick back on table
Philosopher 2 is hungry
Philosopher 2 took chopstick 2 and 3
Philosopher 2 is eating
Philosopher 2 finished eating and kept the chopstick back on table
Philosopher 3 is hungry
Philosopher 3 took chopstick 3 and 4
Philosopher 3 is eating
Philosopher 3 finished eating and kept the chopstick back on table
Philosopher 4 is hungry
Philosopher 4 took chopstick 4 and 5
Philosopher 4 is eating
Philosopher 4 finished eating and kept the chopstick back on table
Philosopher 5 is hungry
Philosopher 5 took chopstick 5 and 1
Philosopher 5 is eating
Philosopher 5 finished eating and kept the chopstick back on table

Process returned 0 (0x0)   execution time : 8.406 s
Press any key to continue.
```



deadlock.

```
#include <stdio.h>
#include <stdbool.h>
void main(){
    int n, m;
    printf("Enter number of processes &
            resources: \n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], request[n][m], avail[m];
    printf("Enter allocation matrix: \n");
    for (int i=0 ; i<n; i++)
      for (int j=0, j<m; j++)
        scanf("%d", &alloc[i][j]);
    printf("Enter request matrix: \n");
    for(int i=0; i<n; i++)
      for(int j=0; j<m; j++)
        scanf("%d", &request[i][j]);
    printf("Enter available matrix: \n");
    for (int i=0; i<m; i++) scanf("%d", &avail[i]);
    int finish[n];
    for (int i=0; i<n; i++) finish[i]=0;
    int count=0;
    while(count<n){
      bool found = false;
      for (int i=0; i<n; i++) {
        if (!finish[i]){
          int j;
          for (j=0; j<m; j++)
            if (request[i][j] > avail[j])
              break;
          if (j == m) {
            for (int k=0; k<m; k++)
              avail[j] += alloc[i][j];
            finish[i] = 1;
```



```
            printf("process %d can finish.\n", i);
            count++;
            found = true;
          }
        }
      }
      if (!found) break;
    }
    if (count == n)
      printf("System is not in a deadlock
              state.\n");
    else
      printf("System is in a deadlock
              state.\n");
}
```

Output:

```
Enter number of processes & resources: 5 3
Enter allocation matrix: 0 1 0 2 0 0 3 0 2 2 1 1 0 0 2
Enter available matrix:
process 0 can finish
process 1 can finish
process 2 can finish
process 3 can finish
process 4 can finish.
System is not in a deadlock state.
```

15/5

PROGRAM 6

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

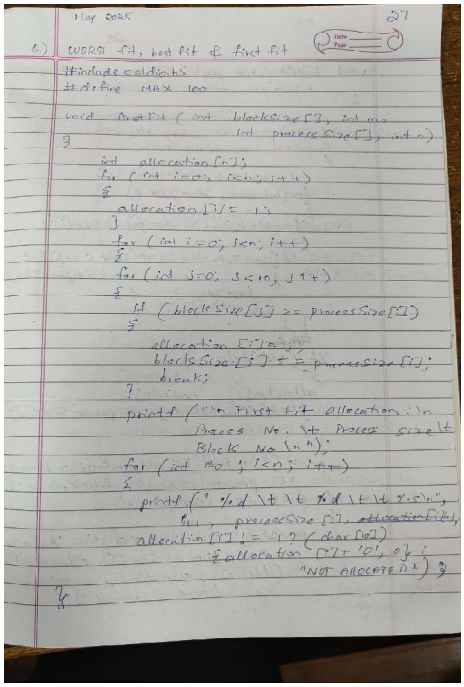
```
#include <stdio.h>

#define MAX 100


void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -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:\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%s\n", i + 1, processSize[i], allocation[i] != -1 ? (char[10]){allocation[i] +
'0', 0} : "Not Allocated");
}


void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -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:\nProcess No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%s\n", i + 1, processSize[i], allocation[i] != -1 ? (char[10]){allocation[i] +
'0', 0} : "Not Allocated");
}

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -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:\nProcess No.\tProcess Size\tBlock No.\n");
```

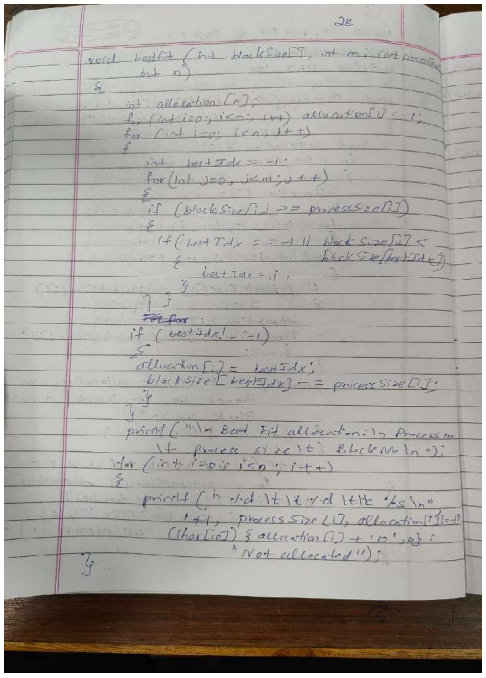
```
    for (int i = 0; i < n; i++)
        printf("%d\t\t%d\t\t%s\n", i + 1, processSize[i], allocation[i] != -1 ? (char[10]){allocation[i] + '0', 0} : "Not Allocated");
}

void main() {
    int m, n, blockSize[MAX], processSize[MAX], temp[MAX];
    printf("Enter number of memory blocks: ");
    scanf("%d", &m);
    printf("Enter sizes of memory blocks: ");
    for (int i = 0; i < m; i++) scanf("%d", &blockSize[i]);
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter sizes of processes: ");
    for (int i = 0; i < n; i++) scanf("%d", &processSize[i]);

    for (int i = 0; i < m; i++) temp[i] = blockSize[i];
    firstFit(temp, m, processSize, n);

    for (int i = 0; i < m; i++) temp[i] = blockSize[i];
    bestFit(temp, m, processSize, n);

    for (int i = 0; i < m; i++) temp[i] = blockSize[i];
    worstFit(temp, m, processSize, n);

}
```

OUTPUT

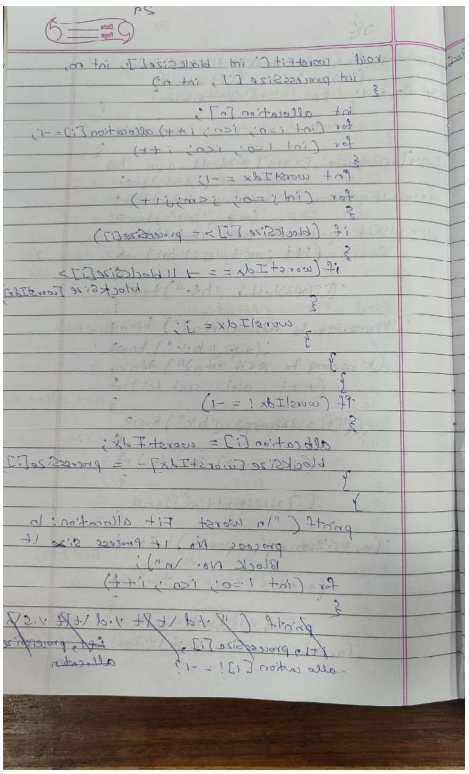
```
Enter number of memory blocks: 5
Enter sizes of memory blocks: 100 500 200 300 600
Enter number of processes: 4
Enter sizes of processes: 212 417 112 426

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

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

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

Process returned 0 (0x0)   execution time : 19.860 s
Press any key to continue.
```

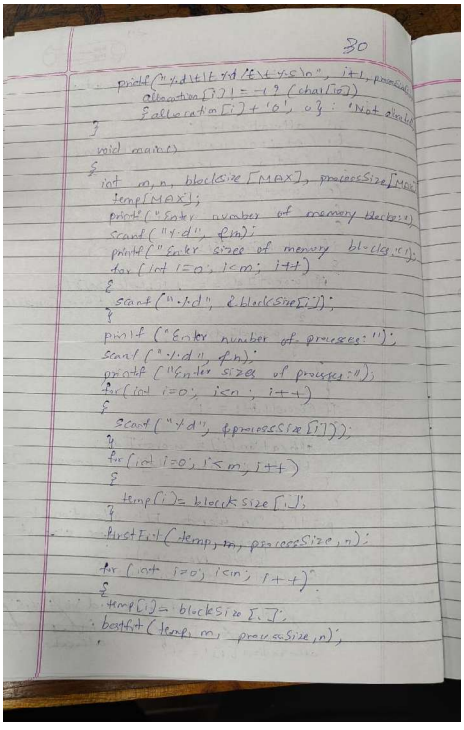


```
6) WORST fit, best fit & first fit
#include<stdio.h>
#define MAX 100
void FirstFit ( int blockSize[], int m,
                int processSize[], int n)
{
    int allocation[n];
    for (int i=0; i<n; i++)
    {
      allocation[i] = -1;
    }
    for (int i=0; i<n; i++)
    {
      for (int j=0; j<m; j++)
      {
        if (blockSize[j] >= processSize[i])
        {
          allocation[i] = j;
          blockSize[j] -= processSize[i];
          break;
        }
      }
    }
    printf("\n First Fit Allocation :\n
            Process No.\t Process Size\t
            Block No\n");
    for (int i=0; i<n; i++)
    {
      printf("  %d \t\t %d \t\t %d\n",
          i+1, processSize[i], allocation[i]+1);
      allocation[i] != -1 ? (char)('0'
          + allocation[i]) : 0);
          "NOT ALLOCATED");
    }
}
```



```
void bestFit (int blockSize[], int m, int processSize[],
              int n)
{
    int allocation[n];
    for (int i=0; i<n; i++) allocation[i] = -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("\n Best Fit allocation:\n Process no.
            \t Process size\t Block No\n");
    for (int i=0; i<n; i++)
    {
      printf("  %d \t\t %d \t\t %s\n",
          i+1, processSize[i], allocation[i]!=-1
          ? (char)('0'+allocation[i]) : "Not allocated");
    }
}
```





```
    printf("%d\t\t%d\t\t%s\n", i+1, processSize[i],
        allocation[i]!=-1 ? (char)('0'+allocation[i])
        : 'Not allocated');
}
void main()
{
    int m,n, blockSize[MAX], processSize[MAX],
        temp[MAX];
    printf("Enter number of memory blocks:");
    scanf("%d", &m);
    printf("Enter size of memory blocks:");
    for(int i=0; i<m; i++)
    {
      scanf("%d", &blockSize[i]);
    }
    printf("Enter number of process:");
    scanf("%d", &n);
    printf("Enter size of process:");
    for(int i=0; i<n; i++)
    {
      scanf("%d", &processSize[i]);
    }
    for(int i=0; i<m; i++)
    {
      temp[i] = blockSize[i];
    }
    firstFit(temp, m, processSize, n);
    for (int i=0; i<m; i++)
    {
      temp[i] = blockSize[i];
    }
    bestFit(temp, m, processSize, n);
```

PROGRAMM 7

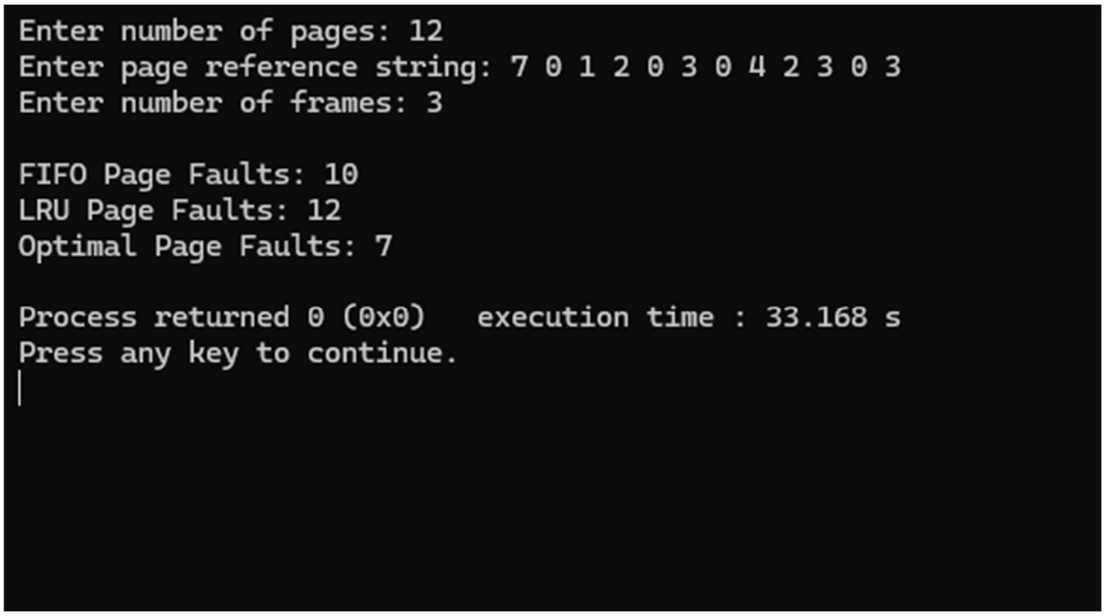
Write a C program to simulate page replacement algorithms.
a) FIFO
b) LRU
c) Optimal

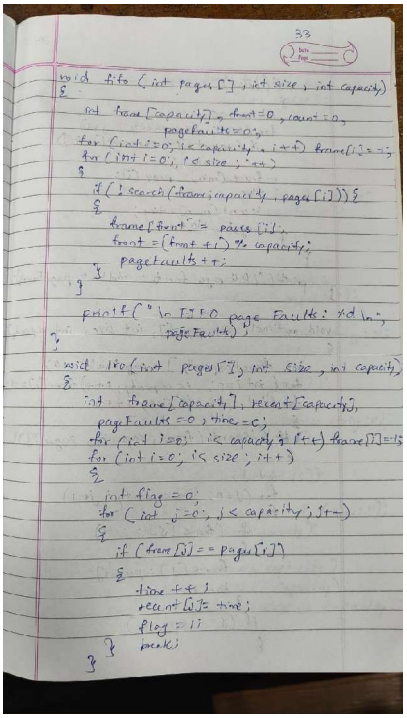
```
#include <stdio.h>

int search(int frame[], int n, int page) {
   for (int i = 0; i < n; i++)
      if (frame[i] == page)
         return 1;
   return 0;
}

int predict(int pages[], int frame[], int n, int index, int size) {
   int res = -1, farthest = index;
   for (int i = 0; i < n; i++) {
      int j;
      for (j = index; j < size; j++) {
         if (frame[i] == pages[j]) {
            if (j > farthest) {
               farthest = j;
               res = i;
            }
            break;
         }
      }
      if (j == size)
         return i;
   }
   return res == -1 ? 0 : res;
}

void fifo(int pages[], int size, int capacity) {
   int frame[capacity], front = 0, count = 0, pageFaults = 0;
   for (int i = 0; i < capacity; i++) frame[i] = -1;
   for (int i = 0; i < size; i++) {
      if (!search(frame, capacity, pages[i])) {
         frame[front] = pages[i];
         front = (front + 1) % capacity;
         pageFaults++;
      }
```

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

void lru(int pages[], int size, int capacity) {
    int frame[capacity], recent[capacity], pageFaults = 0, time = 0;
    for (int i = 0; i < capacity; i++) frame[i] = -1;
    for (int i = 0; i < size; i++) {
        int flag = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                time++;
                recent[j] = time;
                flag = 1;
                break;
            }
        }
        if (!flag) {
            int min = 0;
            for (int j = 1; j < capacity; j++)
                if (recent[j] < recent[min])
                    min = j;
            frame[min] = pages[i];
            time++;
            recent[min] = time;
            pageFaults++;
        }
    }
    printf("LRU Page Faults: %d\n", pageFaults);
}

void optimal(int pages[], int size, int capacity) {
    int frame[capacity], pageFaults = 0;
    for (int i = 0; i < capacity; i++) frame[i] = -1;
    for (int i = 0; i < size; i++) {
        if (!search(frame, capacity, pages[i])) {
            int j;
            for (j = 0; j < capacity; j++)
                if (frame[j] == -1) {
                    frame[j] = pages[i];
                    break;
                }
            if (j == capacity) {
```

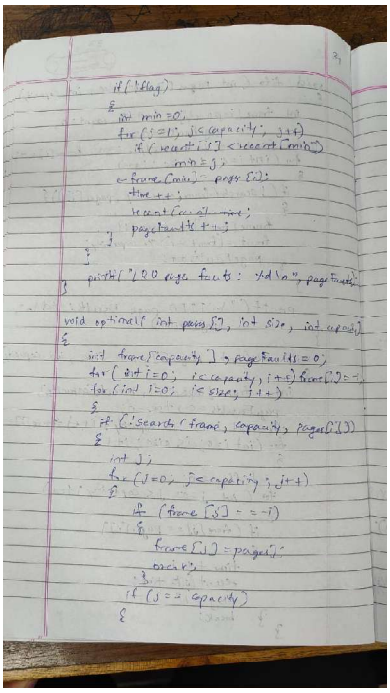
```
            int idx = predict(pages, frame, capacity, i + 1, size);
            frame[idx] = pages[i];
          }
          pageFaults++;
        }
    }
    printf("Optimal Page Faults: %d\n", pageFaults);
}

void main() {
    int pages[100], size, capacity;
    printf("Enter number of pages: ");
    scanf("%d", &size);
    printf("Enter page reference string: ");
    for (int i = 0; i < size; i++) scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &capacity);

    fifo(pages, size, capacity);
    lru(pages, size, capacity);
    optimal(pages, size, capacity);

}
```

OUTPUT

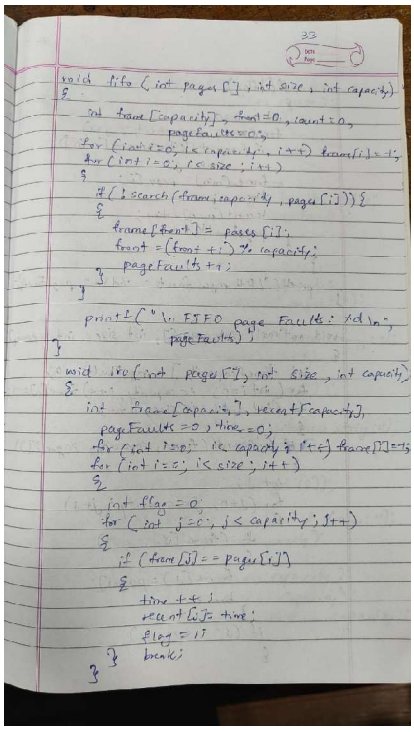




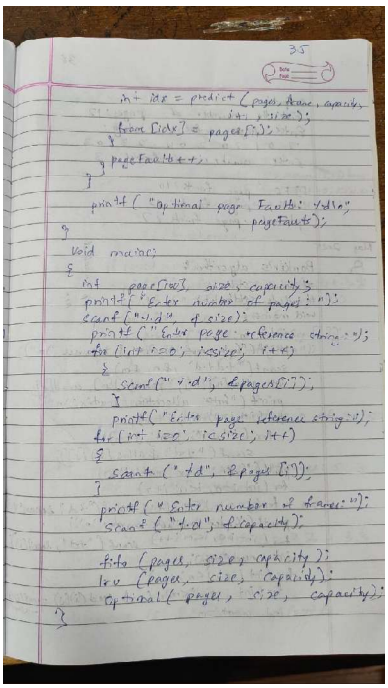
```
void fifo ( int pages[], int size, int capacity)
{
    int frame[capacity], front=0, count=0,
        pageFaults=0;
    for (int i=0; i<capacity; i++) frame[i]=-1;
    for (int i=0; i<size; i++)
    {
        if ( !search (frame, capacity, pages[i])){
        {
            frame[front] = pages[i];
            front = (front+1) % capacity;
            pageFaults++;
        }
    }
    printf("\n FIFO page Faults: %d \n",
            pageFaults);
}
void lru(int pages[], int size, int capacity)
{
    int frame[capacity], recent[capacity],
    pageFaults=0, time=0;
    for (int i=0; i<capacity; i++) frame[i]=-1;
    for (int i=0; i<size; i++)
    {
        int flag=0;
        for (int j=0; j<capacity; j++)
        {
            if (frame[j]==pages[i])
            {
                time++;
                recent[j]=time;
                flag=1;
                break;
            }
        }
```



```
    if(!flag)
    {
        int min=0;
        for(j=1; j<capacity; j++)
            if(recent[j]<recent[min])
                min=j;
        frame[min]=pages[i];
        time++;
        recent[min]=time;
        pageFaults++;
    }
}
printf("\n\n Page Faults: %d\n", pageFaults);
}
void optimal( int pages[], int size, int capacity)
{
    int frame[capacity], pageFaults=0;
    for(int i=0; i<capacity; i++) frame[i]=-1;
    for(int i=0; i<size; i++)
    {
        if(!search(frame, capacity, pages[i]))
        {
            int j;
            for(j=0; j<capacity; j++)
            {
                if(frame[j]==-1)
                {
                    frame[j]=pages[i];
                    break;
                }
            }
            if(j==capacity)
            {
```



```
void fifo ( int pages[], int size, int capacity)
{
    int frame[capacity], front=0, count=0,
        pageFaults=0;
    for (int i=0; i<capacity; i++) frame[i]=-1;
    for (int i=0; i<size; i++)
    {
        if ( !search (frame, capacity, pages[i])){
        {
            frame[front] = pages[i];
            front = (front+1) % capacity;
            pageFaults++;
        }
    }
    printf("\n FIFO page Faults: %d \n",
            pageFaults);
}
void lru(int pages[], int size, int capacity)
{
    int frame[capacity], recent[capacity],
    pageFaults=0, time=0;
    for (int i=0; i<capacity; i++) frame[i]=-1;
    for (int i=0; i<size; i++)
    {
        int flag=0;
        for (int j=0; j<capacity; j++)
        {
            if (frame[j]==pages[i])
            {
                time++;
                recent[j]=time;
                flag=1;
                break;
            }
        }
```



```
    int lru = predict(pages, frame, capacity,
                      i+1, size);
    frame[lru] = pages[i];
}
pageFaults++;
}
printf("Optimal page Faults: %d\n",
        pageFaults);
}
void main()
{
    int pages[50], size, capacity;
    printf("Enter number of pages: ");
    scanf("%d", &size);
    printf("Enter page reference string: ");
    for(int i=0; i<size; i++)
    {
        scanf("%d", &pages[i]);
    }
    printf("Enter page reference string:");
    for(int i=0; i<size; i++)
    {
        scanf("%d", &pages[i]);
    }
    printf("Enter number of frames:");
    scanf("%d", &capacity);
    fifo(pages, size, capacity);
    lru(pages, size, capacity);
    optimal(pages, size, capacity);
}
```