Experiment 1

Aim : To write a C program for searching and sorting.

i. LinearSearch

Source code :

```
#include
<stdio.h>

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

int main() { intarr[] = { 2, 3, 4, 10, 40
}; int x = 10; int n = sizeof(arr) /
sizeof(arr[0]);


    int result = search(arr, n, x); (result
    == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

Output:-

```
Element is present at index 3
```

ii. BinarySearch

SourceCode:

```
#include <stdio.h>

intbinarySearch(intarr[], int l, int r, int x) {
if (r >= l) { int mid = l + (r - l) / 2; if
(arr[mid] == x) return mid; if (arr[mid] >
x) returnbinarySearch(arr, l, mid - 1, x);
returnbinarySearch(arr, mid + 1, r, x);

  }
return -1;
}

int main(void) {
intarr[] = {
    2,
    3,
    4,
    10,
    40
  };
int n = sizeof(arr) / sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n - 1, x);
```

```
 (result == -1) ? printf("Element is not present in array"): printf("Element is present at index %d",
result);
return 0;
}
```

Output:

Element is present at index 3

iii.BubbleSort

Source Code:

```
#include <stdio.h>

void swap(int *xp, int*yp)
{ int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

voidbubbleSort(intarr[], intn)
{ int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in
       place for (j = 0; j < n-i-1; j++)
       if (arr[j] >arr[j+1])
       swap(&arr[j], &arr[j+1]);
}

voidprintArray(intarr[], intsize)
{ int i;
    for (i=0; i < size; i++) printf("%d
        ", arr[i]);
    printf("\n");
} int

main() {

intarr[] =

{64, 34,

25, 12,

22, 11,

90}; int n

=

sizeof(arr

)/sizeof(a

rr[0]);

bubbleSort
(arr,n);

printf("So

rted

array:

\n");

printArray

(arr,n);

return 0;

}
```

Output:

```
Sortedarray:
11 12 22 25 34 64 90
```

Experiment 2 CPU Scheduling

Algorithm

A). FIRST COME FIRST SERVE:

AIM: To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS) DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be servedfirst.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the

burst time Step 4: Set the waiting of the first process as ‗0‘and its burst

time as its turnaround time Step 5: for each process in the Ready

Q calculate

a). Waiting time (n) = waiting time (n-1) + Burst time (n-1)

b). Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate

a)Average waiting time = Total waiting Time / Number ofprocess

b)Average Turnaround time = Total Turnaround Time /

Numberof process Step 7: Stop theprocess

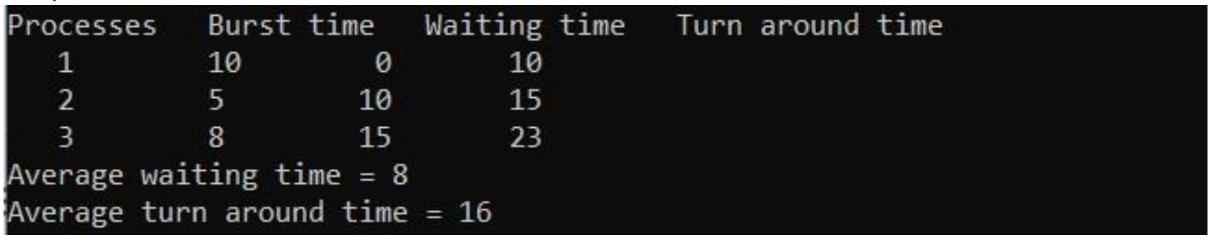
SOURCE CODE:

```
#include<stdio.h>
voidfindWaitingTime(int processes[], int n, intbt[], intwt[])
{
   wt[0] = 0;
   for (int i = 1; i < n ; i++ ) wt[i]
      = bt[i-1] + wt[i-1] ;
}
```

```
voidfindTurnAroundTime( int processes[], int n, intbt[], intwt[], int tat[])
{ for (int i = 0; i < n ; i++)
      tat[i] = bt[i] + wt[i];
}

voidfindavgTime( int processes[], int n, intbt[])
{
   intwt[n], tat[n], total_wt = 0, total_tat = 0;

   findWaitingTime(processes, n, bt, wt);

   findTurnAroundTime(processes, n, bt, wt, tat);

   printf("Processes Burst time Waiting time Turn around time\n");

   for (int i=0; i<n; i++)
   {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      printf(" %d ",(i+1));
      printf("        %d ", bt[i]);
      printf("        %d",wt[i]);
      printf("        %d\n",tat[i]);
   }
   int s=(float)total_wt / (float)n; int
   t=(float)total_tat / (float)n;
   printf("Average waiting time = %d",s);
   printf("\n");
   printf("Average turn around time = %d ",t);
}

int main()
{
   int processes[] = { 1, 2, 3}; int n = sizeof
   processes / sizeof processes[0];

   intburst_time[] = {10, 5, 8};
```

```
    findavgTime(processes, n, burst_time); return
    0;
}
```

Output:



```
Processes   Burst time   Waiting time   Turn around time
   1        10            0       10
   2        5            10       15
   3        8            15       23
Average waiting time = 8
Average turn around time = 16
```

B)SHORTESTJOBFIRST:

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest job first (Non- Preemption)

DESCRIPTION:

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of alltheprocesspreviousorbeforetothatprocess.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.

Step 5: Set the wait ing t ime of the first process as _0‘ and its turnaround time as its burst time.

Step 6: Sort the processes names based on their Burt time

Step 7: For each process in the ready queue, calculate

a) Waiting time(n)= waiting time (n-1) + Burst time(n-1)

b) Turnaround time (n)= waiting time(n)+Bursttime(n)

Step 8: Calculate

c) Average waiting time = Total waiting Time / Number ofprocess

d) Average Turnaround time = Total Turnaround Time / Number of process

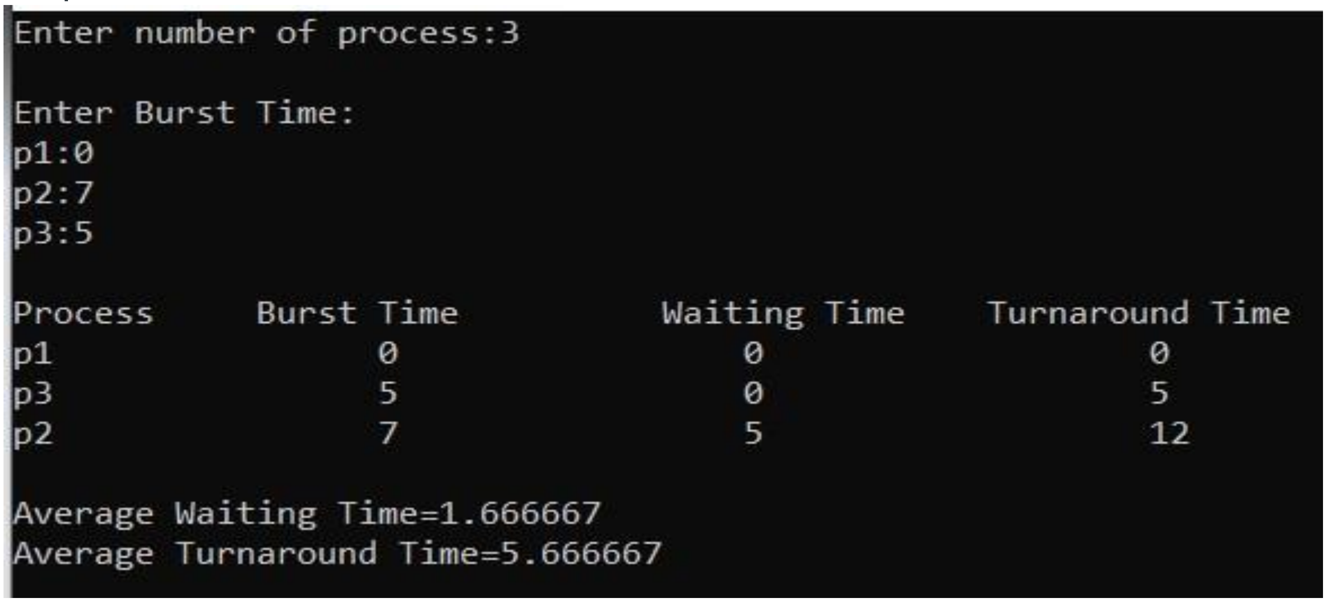
Step 9: Stop theprocess

SOURCE CODE:

```
#include<stdio.h>intmain()
{ intbt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
  float avg_wt,avg_tat; printf("Enter number of
  process:"); scanf("%d",&n); printf("\nEnter Burst
  Time:\n"); for(i=0;i<n;i++)
  { printf("p%d:",i+1);
    scanf("%d",&bt[i]);
    p[i]=i+1;
  }
  //sorting of burst times
  for(i=0;i<n;i++)
  {
    pos=i; for(j=i+1;j<n;j++)
    { if(bt[j]<bt[pos])
      pos=j;
    } temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp
    ; temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
  }
  wt[0]=0;
  for(i=1;i<n;i++)
  { wt[i]=0;
    for(j=0;j<i;j++)
    wt[i]+=bt[j];
    total+=wt[i];
  } avg_wt=(float)total/n; total=0; printf("\nProcess\t     BurstTime
  \tWaiting Time\tTurnaround Time");
  for(i=0;i<n;i++)
```

```
    { tat[i]=bt[i]+wt[i]; total+=tat[i]; printf("\np%d\t\t %d\t\t
            %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=(float)total/n; printf("\n\nAverage Waiting
    Time=%f",avg_wt); printf("\nAverage Turnaround
    Time=%f\n",avg_tat);
}
```

Output:



```
Enter number of process:3

Enter Burst Time:
p1:0
p2:7
p3:5

Process     Burst Time          Waiting Time    Turnaround Time
p1                 0                  0                  0
p3                 5                  0                  5
p2                 7                  5                  12

Average Waiting Time=1.666667
Average Turnaround Time=5.666667
```

c) PRIORITY:

AIM: To simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of processes in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the wait ing of the first process as ‗0' and its burst time as its turnaround time Step 6: Arrange the processes based on process priority

Step 7: For each process in the Ready Q calculate Step 8:

for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time(n-1)

b) Turnaround time (n)= waiting time(n)+Bursttime(n)

Step 9: Calculate

c) Average waiting time = Total waiting Time / Number ofprocess

d) Average Turnaround time = Total Turnaround Time / Number of process Print the results in anorder.

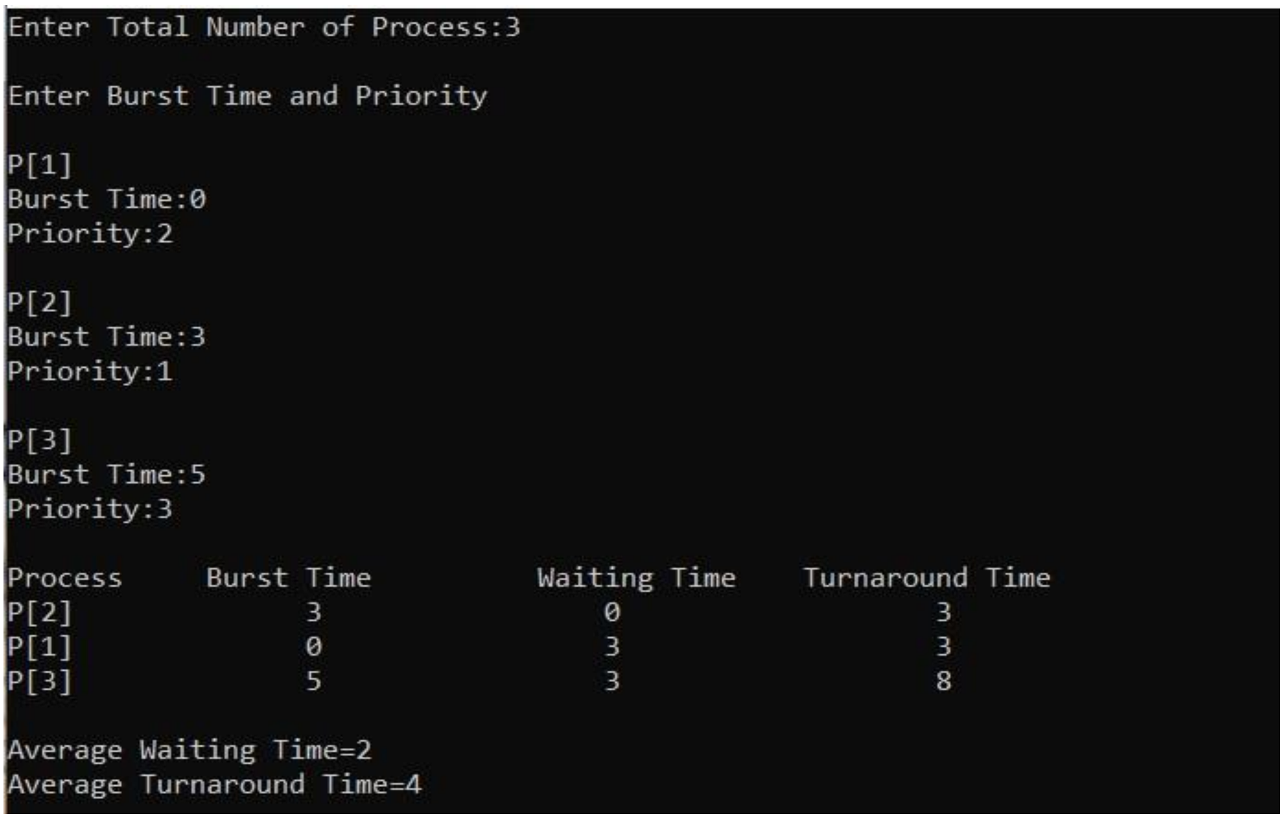
Step10: Stop

SOURCE CODE:

```
#include<stdio.h>

int main()
{
   int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
   printf("Enter Total Number of Process:"); scanf("%d",&n);

   printf("\nEnter Burst Time and Priority\n"); for(i=0;i<n;i++)
   {
       printf("\nP[%d]\n",i+1);
       printf("Burst Time:");
       scanf("%d",&bt[i]);
       printf("Priority:");
       scanf("%d",&pr[i]);
       p[i]=i+1;
   }



   for(i=0;i<n;i++)
   {
       pos=i;
       for(j=i+1;j<n;j++)
       { if(pr[j]<pr[pos])
               pos=j;
       }
```

```
        temp=pr[i]; pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i]; p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;

    for(i=1;i<n;i++)
    {
        wt[i]=0; for(j=0;j<i;j++)
        wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=total/n; total=0;

    printf("\nProcess\t     BurstTime      \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    { tat[i]=bt[i]+wt[i]; total+=tat[i]; printf("\nP[%d]\t\t %d\t\t
              %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n; printf("\n\nAverage Waiting
    Time=%d",avg_wt); printf("\nAverage Turnaround
    Time=%d\n",avg_tat);

return 0;
}
```

Output:



```
Enter Total Number of Process:3

Enter Burst Time and Priority

P[1]
Burst Time:0
Priority:2

P[2]
Burst Time:3
Priority:1

P[3]
Burst Time:5
Priority:3

Process     Burst Time          Waiting Time    Turnaround Time
P[2]            3                   0                   3
P[1]            0                   3                   3
P[3]            5                   3                   8

Average Waiting Time=2
Average Turnaround Time=4
```

Experiment 3

contiguous allocation techniques

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

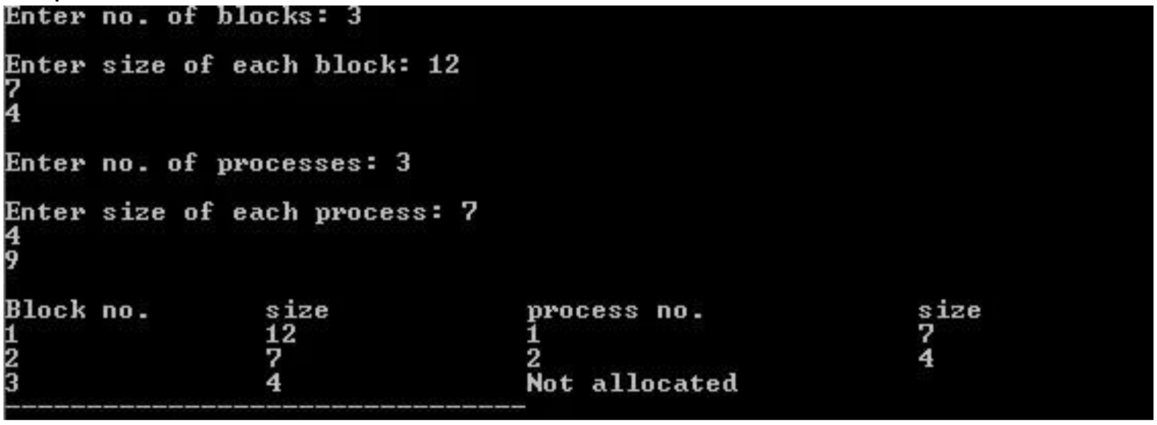
DESCRIPTION:

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest availableblock.

First-FIT

```
#include<stdio.h>


void main()

{ intbsize[10], psize[10], bno, pno, flags[10], allocation[10], i,

j;
for(i = 0; i < 10; i++)
{ flags[i] = 0;
allocation[i] = -
1;
} printf("Enter no. of blocks: ");
scanf("%d", &bno); printf("\nEnter
size of each block: "); for(i = 0; i
<bno; i++) scanf("%d", &bsize[i]);
printf("\nEnter no. of processes: ");
scanf("%d", &pno); printf("\nEnter size
of each process: "); for(i = 0; i
<pno;i++) scanf("%d", &psize[i]); for(i
= 0; i <pno; i++) for(j = 0; j <bno;j++)
if(flags[j] == 0 &&bsize[j] >= psize[i])
{ allocation[j] =
i; flags[j] = 1;
break;
}

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
for(i = 0; i <bno; i++)
{ printf("\n%d\t\t%d\t\t", i+1, bsize[i]); if(flags[i] == 1)
printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]])
; else printf("Not allocated");
}
}
```

Output:

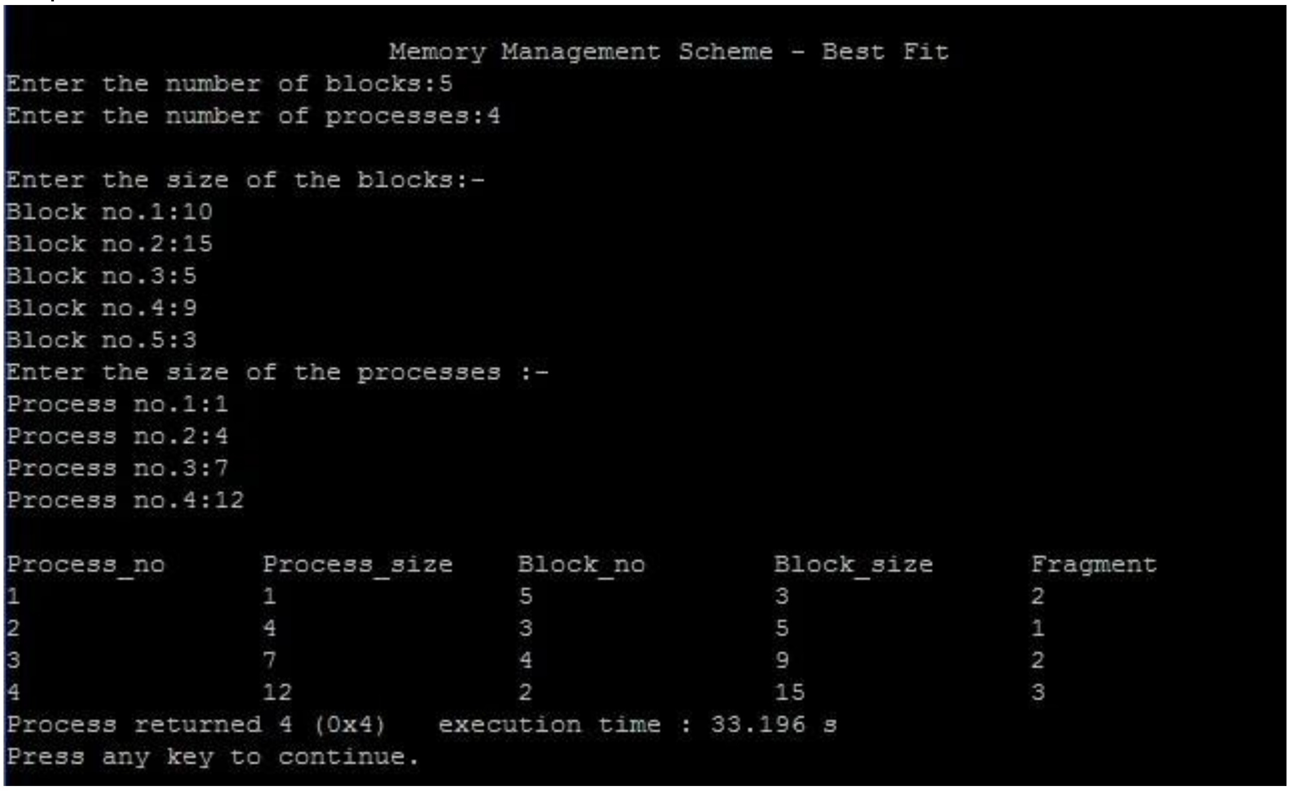


```
Enter no. of blocks: 3

Enter size of each block: 12
7
4

Enter no. of processes: 3

Enter size of each process: 7
4
9

Block no.       size            process no.             size
1               12              1                       7
2               7               2                       4
3               4               Not allocated
--------------------------------
```

Best-Fit:

```
#include<stdio.h> void
main()
{
int
fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
static intbarray[20],parray[20]; printf("\n\t\t\tMemory
Management Scheme - Best Fit"); printf("\nEnter the
number of blocks:"); scanf("%d",&nb); printf("Enter
the number of processes:"); scanf("%d",&np);
printf("\nEnter the size of the blocks:-
\n"); for(i=1;i<=nb;i++)
  { printf("Block
no.%d:",i);
scanf("%d",&b[i]);
  } printf("\nEnter the size of the
processes :-
\n"); for(i=1;i<=np;i++)
  {
     printf("Process no.%d:",i);
     scanf("%d",&p[i]);
  }
for(i=1;i<=np;i++)
{
for(j=1;j<=nb;j++)
{
if(barray[j]!=1)
{
temp=b[j]-
p[i];
if(temp>=0)
if(lowest>temp)
{
parray[i]=j;
lowest=temp
;
}
}
}
fragment[i]=lowest; barray[parray[i]]=
1; lowest=10000;
}
printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
for(i=1;i<=np&&parray[i]!=0;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

Output:



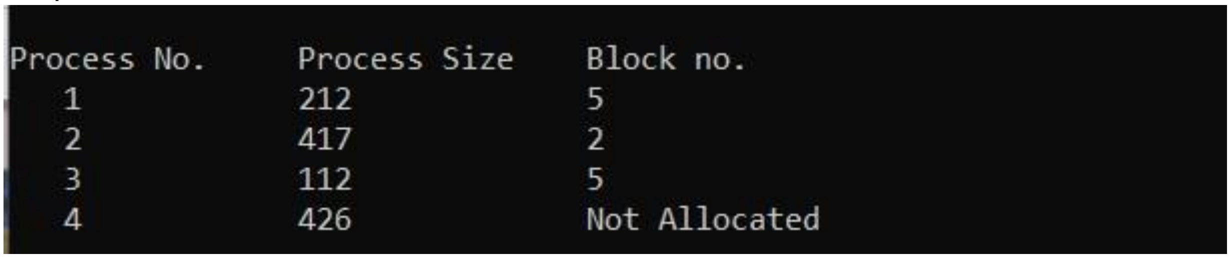
```
                        Memory Management Scheme - Best Fit
Enter the number of blocks:5
Enter the number of processes:4

Enter the size of the blocks:-
Block no.1:10
Block no.2:15
Block no.3:5
Block no.4:9
Block no.5:3
Enter the size of the processes :-
Process no.1:1
Process no.2:4
Process no.3:7
Process no.4:12

Process_no      Process_size    Block_no        Block_size      Fragment
1               1               5               3               2
2               4               3               5               1
3               7               4               9               2
4               12              2               15              3
Process returned 4 (0x4)   execution time : 33.196 s
Press any key to continue.
```

Worst-Fit:

```
#include<bits/stdc++.h> using
namespace std;


voidworstFit(intblockSize[], int m, intprocessSize[],int n)
{ int allocation[n];
   memset(allocation, -1,
   sizeof(allocation)); for (int i=0; i<n;
   i++)
```

```
    { intwstIdx = -1; for
      (int j=0; j<m; j++)
      { if (blockSize[j] >= processSize[i])
        { if (wstIdx == -1) wstIdx
          =j; else if
          (blockSize[wstIdx]
            <blockSize[j]) wstIdx =j;
        } } if (wstIdx
      != -1)
      { allocation[i] = wstIdx;
        blockSize[wstIdx] -=
        processSize[i];
      }
    }

    cout<< "\nProcess No.\tProcess Size\tBlock
    no.\n"; for (int i = 0; i < n; i++)
    {
      cout<< " " << i+1 << "\t\t" <<processSize[i] <<
      "\t\t"; if (allocation[i] != -1)
        cout<< allocation[i] +
      1; else
        cout<< "Not
      Allocated"; cout<<endl;
    }
  }

  int main()
  {
    intblockSize[] = {100, 500, 200, 300, 600};
    intprocessSize[] = {212, 417, 112, 426}; int m =
    sizeof(blockSize)/sizeof(blockSize[0]); int n =
    sizeof(processSize)/sizeof(processSize[0]);

    worstFit(blockSize, m, processSize, n);
    return 0 ;
}
```

Output:



```
Process No.     Process Size    Block no.
   1            212             5
   2            417             2
   3            112             5
   4            426             Not Allocated
```

Experiment 3

Dead lock prevention

Aim : To implement Banker’s Algorithm.

Description:

When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user request a set of resources, the system must determine whether the allocation of each resources will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Source Code:

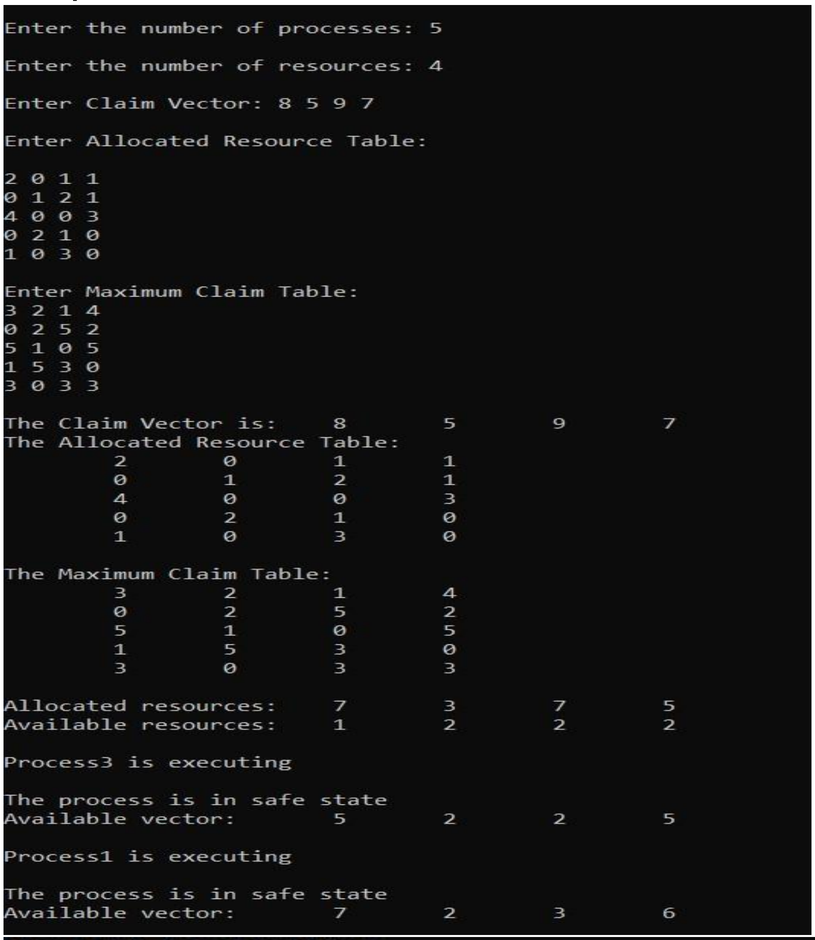
```
#include <stdio.h> intcurr[5][5],
maxclaim[5][5], avl[5]; intalloc[5]
= {0, 0, 0, 0, 0}; intmaxres[5],
running[5], safe=0; int count =
0, i, j, exec, r, p, k = 1;

int main()
{
    printf("\nEnter the number of processes:
    "); scanf("%d", &p);

    for (i = 0; i < p; i++)
        { running[i]
```

```
   = 1; count++;
}

printf("\nEnter the number of resources: ");
scanf("%d", &r);

printf("\nEnter Claim
Vector:"); for (i = 0; i < r; i++)
{
   scanf("%d", &maxres[i]);
}

printf("\nEnter Allocated Resource
Table:\n"); for (i = 0; i < p; i++) {
   for(j = 0; j < r; j++) {
      scanf("%d",
      &curr[i][j]);
   }
}

printf("\nEnter Maximum Claim
Table:\n"); for (i = 0; i < p; i++) {
for(j = 0; j < r; j++) {
scanf("%d",&maxclaim[i][j]);
   }
}

printf("\nThe Claim Vector is:
"); for (i = 0; i < r; i++) {
   printf("\t%d", maxres[i]);
}

printf("\nThe Allocated Resource
Table:\n"); for (i = 0; i < p; i++) {
   for (j = 0; j < r; j++) {
   printf("\t%d", curr[i][j]);
   }
   printf("\n");
}

printf("\nThe Maximum Claim
Table:\n"); for (i = 0; i < p; i++) {
   for (j = 0; j < r; j++) {
   printf("\t%d", maxclaim[i][j]); }

   printf("\n");
}

for (i = 0; i < p; i++) {
   for (j = 0; j < r; j++) {
   alloc[j] += curr[i][j];
   }
}

printf("\nAllocated
resources:"); for (i = 0; i <
r; i++) { printf("\t%d",
alloc[i]); }

for (i = 0; i < r; i++) { avl[i] =
   maxres[i] - alloc[i];
}

printf("\nAvailable
resources:"); for (i = 0; i <
r; i++) { printf("\t%d", avl[i]);

}
printf("\n");
```

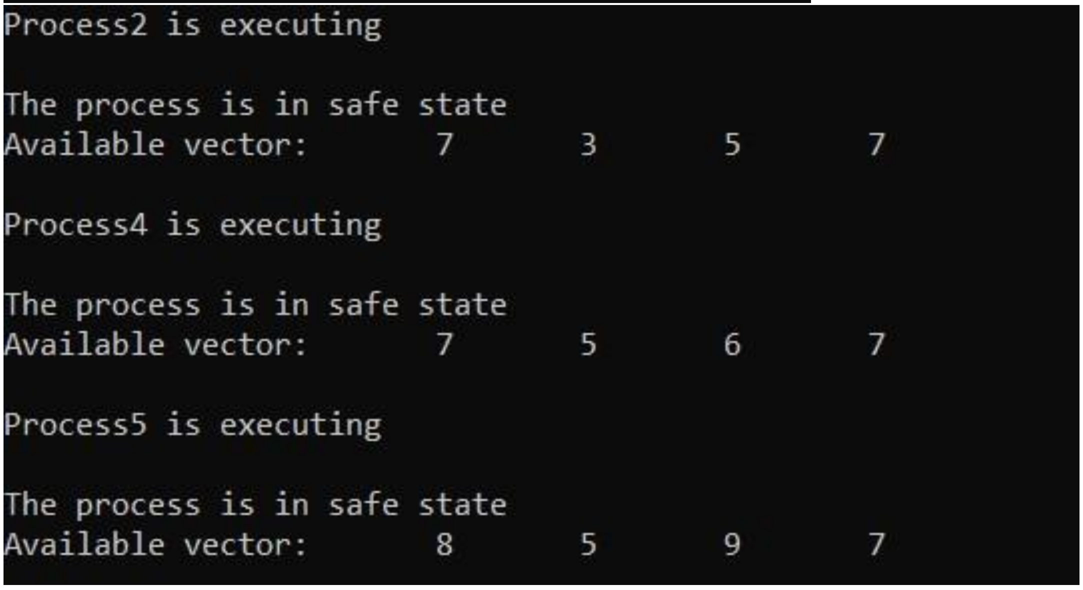
```
//Main procedure goes below to check for unsafe
state. while (count != 0) { safe = 0;
  for (i = 0; i < p; i++)
    { if
    (running[i]) {
      exec = 1;
      for (j = 0; j < r; j++) { if
        (maxclaim[i][j] - curr[i][j] >avl[j])
          { exec = 0;
          break;
        }
      } if (exec) { printf("\nProcess%d is
      executing\n", i +
        1); running[i] =
        0; count--; safe
        = 1;

        for (j = 0; j < r; j++)
          { avl[j] +=
          curr[i][j];
        }

        break;
      }
    } } if (!safe) { printf("\nThe processes are in
  unsafe state.\n"); break;

  } else { printf("\nThe process is in
    safe state"); printf("\nAvailable
    vector:");

    for (i = 0; i < r; i++)
    { printf("\t%d",
    avl[i]); }

    printf("\n");
```

```
        }
    }
}
```

Output:



```
Enter the number of processes: 5

Enter the number of resources: 4

Enter Claim Vector: 8 5 9 7

Enter Allocated Resource Table:

2 0 1 1
0 1 2 1
4 0 0 3
0 2 1 0
1 0 3 0

Enter Maximum Claim Table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

The Claim Vector is:    8       5       9       7
The Allocated Resource Table:
        2       0       1       1
        0       1       2       1
        4       0       0       3
        0       2       1       0
        1       0       3       0

The Maximum Claim Table:
        3       2       1       4
        0       2       5       2
        5       1       0       5
        1       5       3       0
        3       0       3       3

Allocated resources:    7       3       7       5
Available resources:    1       2       2       2

Process3 is executing

The process is in safe state
Available vector:       5       2       2       5

Process1 is executing

The process is in safe state
Available vector:       7       2       3       6
```



```
Process2 is executing

The process is in safe state
Available vector:       7       3       5       7

Process4 is executing

The process is in safe state
Available vector:       7       5       6       7

Process5 is executing

The process is in safe state
Available vector:       8       5       9       7
```

Experiment 4

Disk Scheduling

AIM: To Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C- SCAN

DESCRIPTION

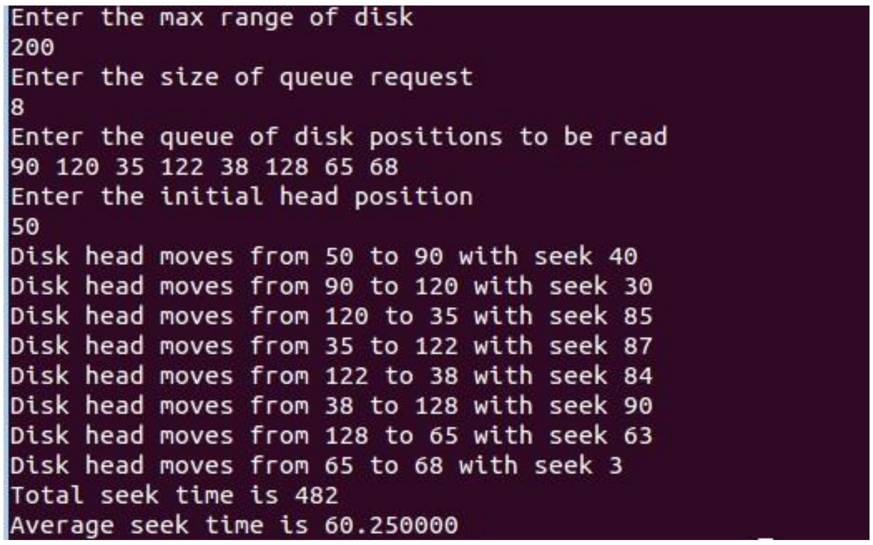
One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the first-come, firstserved (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the returntrip.

A)FCFS DISK SCHEDULING ALGORITHM

Source Code:

```
#include<stdio.h>intmain()
{
        int queue[20],n,head,i,j,k,seek=0,max,diff;
        float avg; printf("Enter the max range of
        disk\n"); scanf("%d",&max);
        printf("Enter the size of queue request\n");
        scanf("%d",&n); printf("Enter the queue of disk
        positions to be read\n"); for(i=1;i<=n;i++)
        scanf("%d",&queue[i]); printf("Enter the initial head
        position\n");
        scanf("%d",&head);
        queue[0]=head; for(j=0;j<=n-1;j++)
        { diff=abs(queue[j+1]-queue[j]); seek+=diff; printf("Disk head moves from %d to %d with
                seek %d\n",queue[j],queue[j+1],diff);
        }
        printf("Total seek time is %d\n",seek);
        avg=seek/(float)n; printf("Average seek
        time is %f\n",avg); return 0;
}
```

Output:



```
Enter the max range of disk
200
Enter the size of queue request
8
Enter the queue of disk positions to be read
90 120 35 122 38 128 65 68
Enter the initial head position
50
Disk head moves from 50 to 90 with seek 40
Disk head moves from 90 to 120 with seek 30
Disk head moves from 120 to 35 with seek 85
Disk head moves from 35 to 122 with seek 87
Disk head moves from 122 to 38 with seek 84
Disk head moves from 38 to 128 with seek 90
Disk head moves from 128 to 65 with seek 63
Disk head moves from 65 to 68 with seek 3
Total seek time is 482
Average seek time is 60.250000
```

B)SCAN DISK SCHEDULINGALGORITHM

Source code:

```
#include<stdio.h>intmain()
{
inti,j,sum=0,n;
int d[20]; int
disk; //loc of
head inttemp,max;
intdloc; //loc of disk in array
clrscr(); printf("enter number
of location\t");
scanf("%d",&n); printf("enter
position of head\t");
scanf("%d",&disk);
printf("enter elements of
disk queue\n");
for(i=0;i<n;i++)
{
scanf("%d",&d[i]);
}
d[n]=disk;
n=n+1;
for(i=0;i<n;i++) // sorting disk locations
{
for(j=i;j<n;j++
) { if(d[i]>d[j])
{
temp=d[i];
d[i]=d[j];
d[j]=temp;
}
}


}
max=d[n];
for(i=0;i<n;i++) // to find loc of disc in array
{ if(disk==d[i]) { dloc=i;
break; }
} for(i=dloc;i>=0;i--
)
{ printf("%d--
>",d[i]);
} printf("0 -->");
for(i=dloc+1;i<n;i++
)
{ printf("%d--
>",d[i]); }
sum=disk+max;
printf("\nmovement of total cylinders
%d",sum); getch();
return 0;
}
```

Output:

```
Output:
Enter no of location 8
Enter position of head
53
Enter elements of disk
queue 98
183
37
122
14
124
65
67
53->37->14->0->65->67->98->122->124->183->
Movement of total cylinders 236.
```

C)

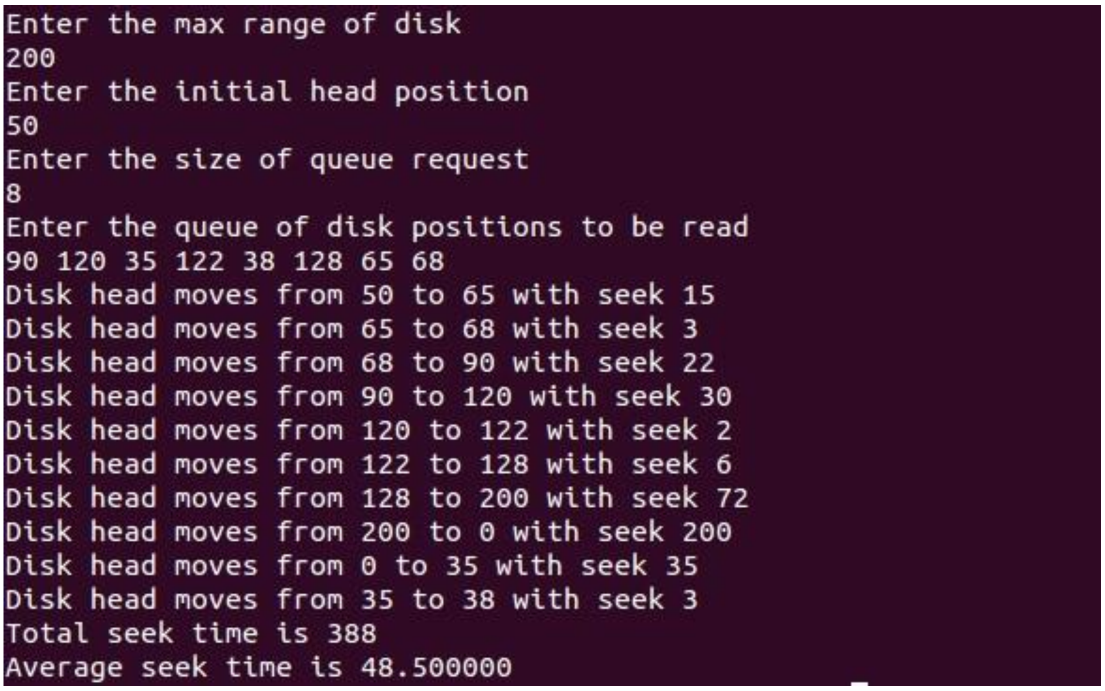
C-

SCAN DISK SCHEDULINGALGORITHM

Source Code:-

```
#include<stdio.h>intmain()
{ int queue[20],n,head,i,j,k,seek=0,max,diff,temp,queue1[20],queue2[20],
        temp1=0,temp2=0;
        floatavg;
        printf("Enter the max range of disk\n");
        scanf("%d",&max);
        printf("Enter the initial head position\n");
        scanf("%d",&head); printf("Enter the size
        of queue request\n");
        scanf("%d",&n); printf("Enter the queue of disk
        positions to be read\n"); for(i=1;i<=n;i++)
        { scanf("%d",&temp);
                if(temp>=head)
                { queue1[temp1]=temp;
                        temp1++;
                }
                else
                { queue2[temp2]=temp;
                        temp2++;
                }
        } for(i=0;i<temp1-
        1;i++)
        { for(j=i+1;j<temp1;j++)
                { if(queue1[i]>queue1[j])
                        { temp=queue1[i];
                                queue1[i]=queue1[j];
                                queue1[j]=temp;
                        }
                }
        }
        for(i=0;i<temp2-1;i++)
        { for(j=i+1;j<temp2;j++)
                { if(queue2[i]>queue2[j])
                        { temp=queue2[i];
                                queue2[i]=queue2[j];
                                queue2[j]=temp;
                        }
                }
        }
        for(i=1,j=0;j<temp1;i++,j++)
        queue[i]=queue1[j]; queue[i]=max;
        queue[i+1]=0;
        for(i=temp1+3,j=0;j<temp2;i++,j++)
        queue[i]=queue2[j]; queue[0]=head;
        for(j=0;j<=n+1;j++)
        { diff=abs(queue[j+1]-queue[j]);
                seek+=diff;
                printf("Disk head moves from %d to %d with
seek    %d\n",queue[j],queue[j+1],diff); }
        printf("Total seek time is %d\n",seek);
        avg=seek/(float)n; printf("Average seek
        time is %f\n",avg); return 0; }
```

Output:



```
Enter the max range of disk
200
Enter the initial head position
50
Enter the size of queue request
8
Enter the queue of disk positions to be read
90 120 35 122 38 128 65 68
Disk head moves from 50 to 65 with seek 15
Disk head moves from 65 to 68 with seek 3
Disk head moves from 68 to 90 with seek 22
Disk head moves from 90 to 120 with seek 30
Disk head moves from 120 to 122 with seek 2
Disk head moves from 122 to 128 with seek 6
Disk head moves from 128 to 200 with seek 72
Disk head moves from 200 to 0 with seek 200
Disk head moves from 0 to 35 with seek 35
Disk head moves from 35 to 38 with seek 3
Total seek time is 388
Average seek time is 48.500000
```

Experiment 5

FCFS Page replacement AIM: To

implement FCFS page replacement technique.

DESCRIPTION:

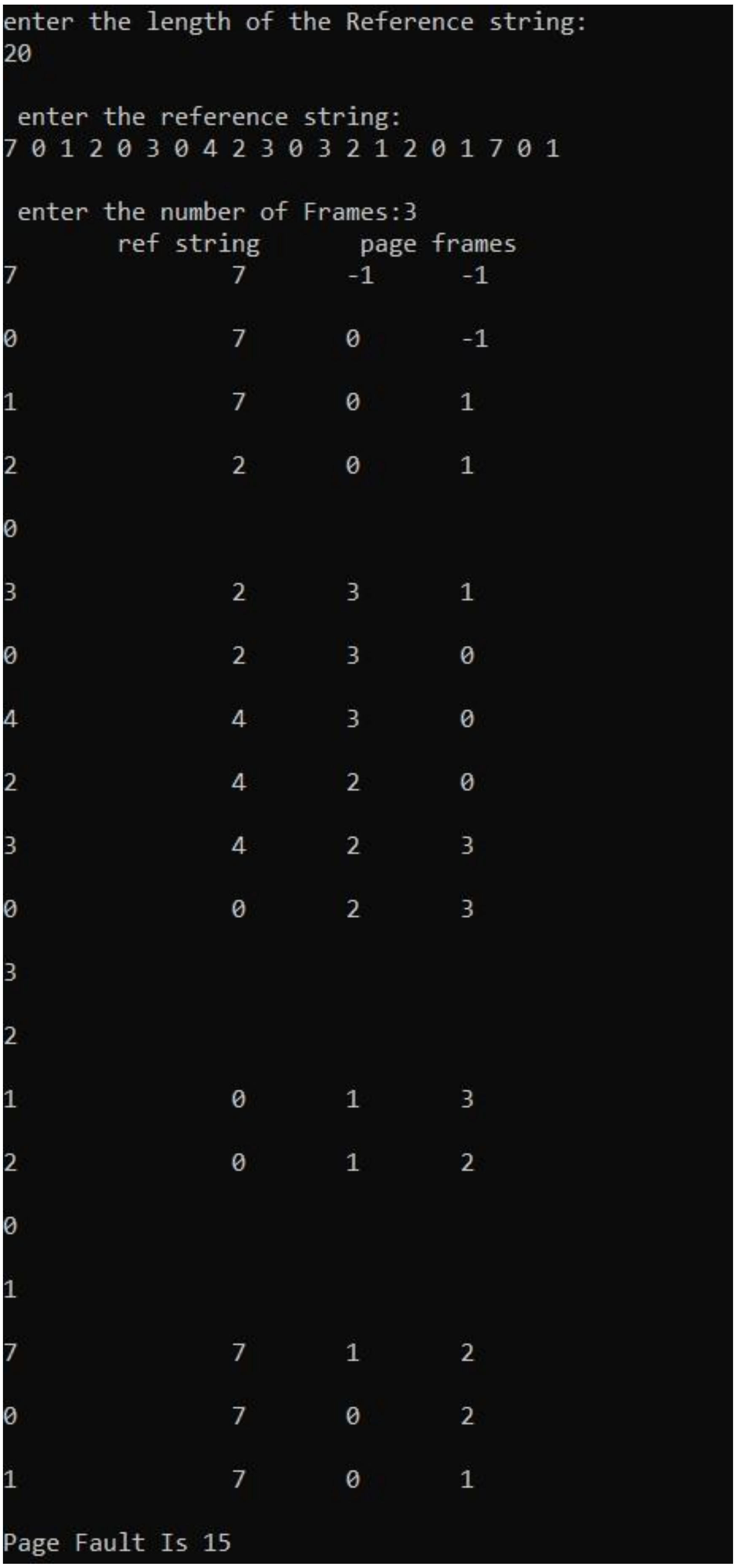
In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

First In First Out (FIFO) page replacement algorithm – This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected forremoval.

Source Code:

```
#include<stdio.h>intmain()

{ inti,j,n,a[50],frame[10],no,k,avail,count=0; printf("\nenter

 the length of the Reference string:\n"); scanf("%d",&n);

 printf("\n enter the reference string:\n"); for(i=1;i<=n;i++)
scanf("%d",&a[i]); printf("\n enter the number of Frames:");

scanf("%d",&no); for(i=0;i<no;i++)

          frame[i]= -1; j=0; printf("\tref string\t page

    frames\n"); for(i=1;i<=n;i++)

                   { printf("%d\t\t",a[i]);

                           avail=0;

                           for(k=0;k<no;k++)

       if(frame[k]==a[i])

                                    avail=1; if

                           (avail==0)

                           { frame[j]=a[i]; j=(j+1)%no;

                                    count++;

                                    for(k=0;k<no;k++)

                                    printf("%d\t",frame[k]);

      }

                           printf("\n\n");

     }

                  printf("Page Fault Is %d",count); return

                  0;

}
```

Output:



```
enter the length of the Reference string:
20

 enter the reference string:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

 enter the number of Frames:3
        ref string       page frames
7               7       -1      -1

0               7       0       -1

1               7       0       1

2               2       0       1

0

3               2       3       1

0               2       3       0

4               4       3       0

2               4       2       0

3               4       2       3

0               0       2       3

3

2

1               0       1       3

2               0       1       2

0

1

7               7       1       2

0               7       0       2

1               7       0       1

Page Fault Is 15
```

Experiment 6

LRU Page replacement AIM: To

implement LUR page replacement technique.

DESCRIPTION:

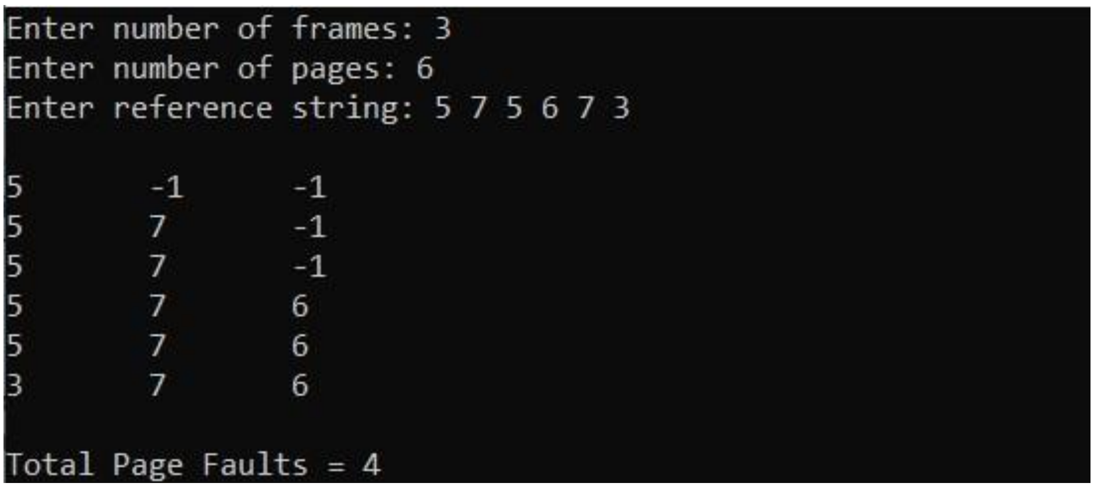
In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.
In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used.

Source Code:

```
#include<stdio.h>


intfindLRU(int time[], int n){ int i,

minimum = time[0], pos = 0;


for(i = 1; i < n; ++i){ if(time[i]

<

minimum){ minimum

= time[i]; pos = i;

}

}

returnpos;

}


int main()

{

   intno_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j, pos, faults = 0; printf("Enter

number of frames: ");

scanf("%d", &no_of_frames);

printf("Enter number of pages:
```

```
"); scanf("%d", &no_of_pages);
printf("Enter reference string: ");
for(i = 0; i <no_of_pages;
    ++i){ scanf("%d",
    &pages[i]);
   }

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

   for(i = 0; i <no_of_pages; ++i){
    flag1 = flag2 = 0;

    for(j = 0; j <no_of_frames; ++j){
    if(frames[j] == pages[i]){
    counter++;
    time[j] = counter;
  flag1 = flag2 = 1;
  break;
  }
    }

    if(flag1 == 0){ for(j = 0;
j <no_of_frames;
    ++j){ if(frames[j] == -1){ counter++;

    faults++; frames[j]
    = pages[i]; time[j] =
    counter; flag2 =1;
    break;
    }
    }
    }
    if(flag2 == 0){ pos =
    findLRU(time,
    no_of_frames);
    counter++; faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
    }

    printf("\n");

    for(j = 0; j <no_of_frames;
    ++j){ printf("%d\t", frames[j]);
    }
} printf("\n\nTotal Page Faults = %d",
faults);

return 0
```

OUTPUT-



```
Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3

5       -1      -1
5       7       -1
5       7       -1
5       7       6
5       7       6
3       7       6

Total Page Faults = 4
```