OPERATING SYSTEMS

DIGITAL ASSIGNMENT-1

Name: **VIBHU KUMAR SINGH**
Reg. No: **19BCE0215**
Teacher: **Manikandan K.**

Q1. Implement a program to allocate memory by applying the following strategies.
a. FIRST FIT
b. BEST FIT
c. WORST FIT

A1.

a)First Fit-

ALGORITHM-

```
1. Input the no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Now allocate processes
if(block size >= process size)
//allocate the process
else
//move on to next block
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.
```

SOURCE CODE-

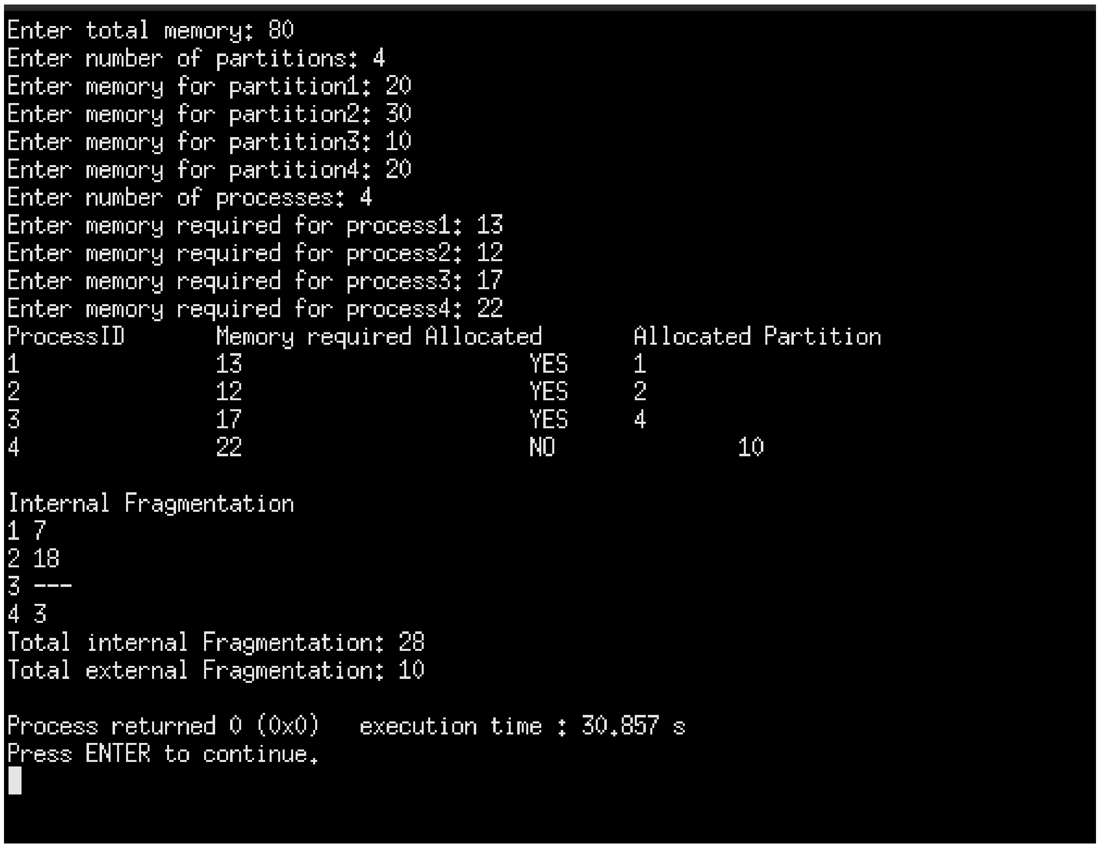
```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct process
{
        int id;
        int memory_required;
        int allocated_partition;
        int external_fragment;
        bool allocated;
};
struct partition
{
        int id;
        int size;
        int internal_fragment;
        bool alloted;
};
```

```
int main()
{
        int memory,no_of_partitions,no_of_processes,i,j,sum=0;
        printf("Enter total memory: ");
        scanf("%d",&memory);
        printf("Enter number of partitions: ");
        scanf("%d",&no_of_partitions);
        struct partition parti[no_of_partitions];
        for(i=0;i<no_of_partitions;i++)
        {
                printf("Enter memory for partition%d: ",i+1);
                scanf("%d",&parti[i].size);
                parti[i].id = i+1;
                parti[i].alloted=false;

                sum+=parti[i].size;
        }
        if(sum < memory)
        {
                no_of_partitions++;
                parti[i].size = memory - sum;
                parti[i].id = i+1;
                parti[i].alloted=false;
                printf("Size of partition%d: %d\n",i,parti[i].size );
        }
        int total_internal_fragment=0, total_external_fragment=0;
        printf("Enter number of processes: ");
        scanf("%d",&no_of_processes);
        struct process proc[no_of_processes];
        for (i = 0; i < no_of_processes; ++i)
        {
                printf("Enter memory required for process%d: ",i+1 );
                scanf("%d",&proc[i].memory_required);
                proc[i].id = i+1;
                proc[i].external_fragment=0;
                proc[i].allocated = false;
                for(j=0;j<no_of_partitions;j++)
                {
                        if(proc[i].memory_required<=parti[j].size &&
!parti[j].alloted)
                        {
                                proc[i].allocated = true;
```

```
                                        proc[i].allocated_partition = parti[j].id;
                                        parti[j].internal_fragment = parti[j].size -
proc[i].memory_required;
                                        total_internal_fragment+=parti[j].internal_fragment;
                                        parti[j].alloted=true;
                                        break;
                                }
                        }
                        for(j=0;j<no_of_partitions;j++)
                        {
                                if(parti[j].alloted==false)
                                {
                                        proc[i].external_fragment+=parti[j].size;
                                }
                        }
                }
                for(j=0;j<no_of_partitions;j++)
                {
                        if(!parti[j].alloted)
                        {
                                total_external_fragment+=parti[j].size;
                                parti[j].internal_fragment = -1;
                        }
                }
                printf("ProcessID\tMemory required\tAllocated\tAllocated
Partition\n");
                for(i=0;i<no_of_processes;i++)
                {
                        if(proc[i].allocated)
                        {
                                printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id,
proc[i].memory_required,  proc[i].allocated_partition);
                        }
                        else
                        {
                                printf("%d\t\t%d\t\t\tNO\t\t%d\n",proc[i].id,
proc[i].memory_required, proc[i].external_fragment);
                        }
                }
                printf("\n");
                printf("Internal Fragmentation\n");
                for(i=0;i<no_of_partitions;i++)
```

```
        {
                if(parti[i].internal_fragment==-1)
                {
                        printf("%d ---\n",parti[i].id);
                }
                else
                        printf("%d %d\n",parti[i].id,parti[i].internal_fragment );
        }
        printf("Total internal Fragmentation: %d\nTotal external
Fragmentation: %d\n",total_internal_fragment, total_external_fragment);

}
```

OUTPUT-



```
Enter total memory: 80
Enter number of partitions: 4
Enter memory for partition1: 20
Enter memory for partition2: 30
Enter memory for partition3: 10
Enter memory for partition4: 20
Enter number of processes: 4
Enter memory required for process1: 13
Enter memory required for process2: 12
Enter memory required for process3: 17
Enter memory required for process4: 22
ProcessID       Memory required Allocated       Allocated Partition
1               13                      YES     1
2               12                      YES     2
3               17                      YES     4
4               22                      NO              10

Internal Fragmentation
1 7
2 18
3 ---
4 3
Total internal Fragmentation: 28
Total external Fragmentation: 10

Process returned 0 (0x0)   execution time : 30.857 s
Press ENTER to continue.
```

b)BEST FIT-

ALGORITHM-

1)Enter the memory blocks with size.
2)Enter the process blocks with size.
3)Set all the memory blocks as free.
4)Start by picking up each process

5)Find the minimum block size that is best to assign to the current process.
6)If the best fit memory size is found, it is allocated to the process.
7)If the memory block and memory demand do not match, leave the process and search for another process.
8)End

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct process
{
        int id;
        int memory_required;
        int allocated_partition;
        int external_fragment;
        bool allocated;
};
struct partition
{
        int id;
        int size;
        int internal_fragment;
        bool alloted;
};
int main()
{
        int memory,no_of_partitions,no_of_processes,i,j,sum=0;
        printf("Enter total memory: ");
        scanf("%d",&memory);
        printf("Enter number of partitions: ");
        scanf("%d",&no_of_partitions);
        struct partition parti[no_of_partitions];
        for(i=0;i<no_of_partitions;i++)
        {
                printf("Enter memory for partition%d: ",i+1);
                scanf("%d",&parti[i].size);
                parti[i].id = i+1;
                parti[i].alloted=false;

                sum+=parti[i].size;
```

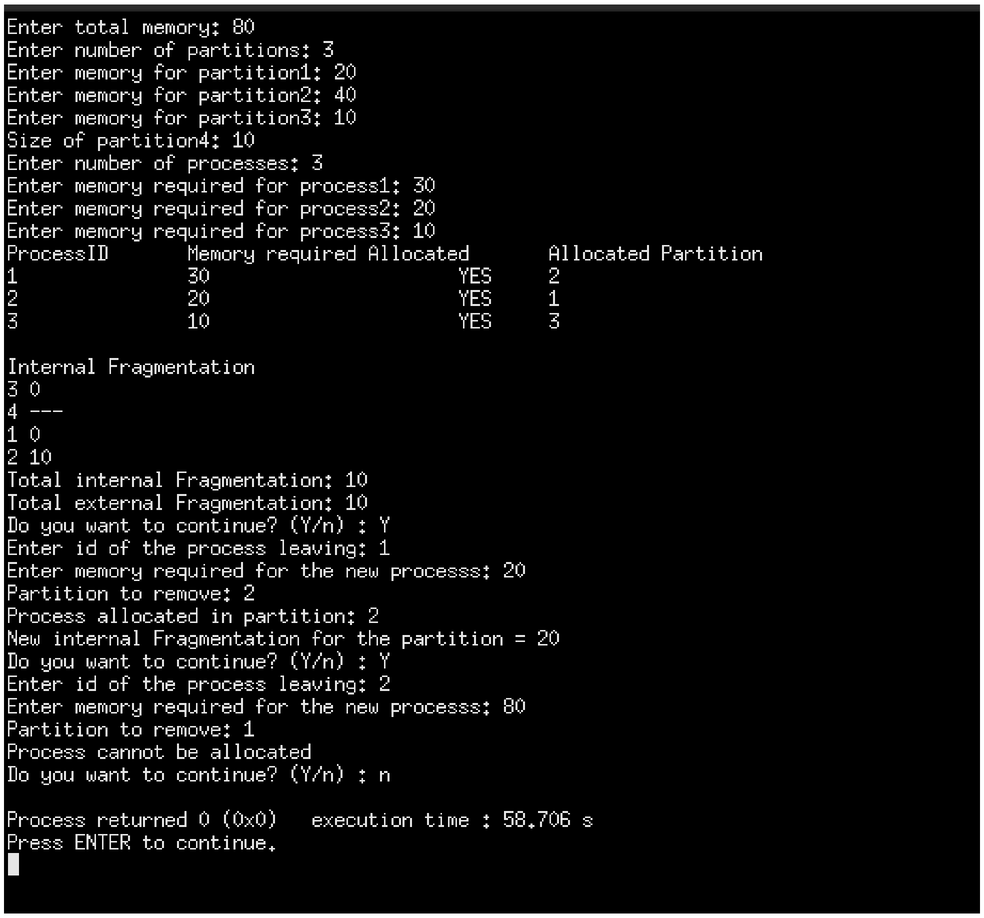
```
            }
            if(sum < memory)
            {
                    no_of_partitions++;
                    parti[i].size = memory - sum;
                    printf("Size of partition%d: %d\n",i+1,parti[i].size );
                    parti[i].id = i+1;
                    parti[i].alloted = false;
            }
            for(i=0;i<no_of_partitions-1;i++)
            {
                    for(j=0;j<no_of_partitions-1-i;j++)
                    {
                            if(parti[j].size>parti[j+1].size)
                            {
                                    struct partition temp = parti[j];
                                    parti[j] = parti[j+1];
                                    parti[j+1] = temp;
                            }
                    }
            }
            int total_internal_fragment=0, total_external_fragment=0;
            printf("Enter number of processes: ");
            scanf("%d",&no_of_processes);
            struct process proc[no_of_processes];
            for (i = 0; i < no_of_processes; ++i)
            {
                    printf("Enter memory required for process%d: ",i+1 );
                    scanf("%d",&proc[i].memory_required);
                    proc[i].id = i+1;
                    proc[i].external_fragment=0;
                    proc[i].allocated = false;
                    for(j=0;j<no_of_partitions;j++)
                    {
                            if(proc[i].memory_required<=parti[j].size &&
    !parti[j].alloted)
                            {
                                    proc[i].allocated = true;
                                    proc[i].allocated_partition = parti[j].id;
                                    parti[j].internal_fragment = parti[j].size -
    proc[i].memory_required;
                                    total_internal_fragment+=parti[j].internal_fragment;
```

```
                                parti[j].alloted=true;
                                break;
                        }
                }
                for(j=0;j<no_of_partitions;j++)
                {
                        if(parti[j].alloted==false)
                        {
                                proc[i].external_fragment+=parti[j].size;
                        }
                }
        }
        for(j=0;j<no_of_partitions;j++)
        {
                if(!parti[j].alloted)
                {
                        total_external_fragment+=parti[j].size;
                        parti[j].internal_fragment = -1;
                }
        }
        printf("ProcessID\tMemory required\tAllocated\tAllocated
Partition\n");
        for(i=0;i<no_of_processes;i++)
        {
                if(proc[i].allocated)
                {
                        printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id,
proc[i].memory_required,  proc[i].allocated_partition);
                }
                else
                {
                        printf("%d\t\t%d\t\t\tNO\t%d\n",proc[i].id,
proc[i].memory_required, proc[i].external_fragment);
                }
        }
        printf("\n");
        printf("Internal Fragmentation\n");
        for(i=0;i<no_of_partitions;i++)
        {
                if(parti[i].internal_fragment==-1)
                {
                        printf("%d ---\n",parti[i].id);
```

```
			}
			else
				printf("%d %d\n",parti[i].id,parti[i].internal_fragment );
		}
		printf("Total internal Fragmentation: %d\nTotal external
	Fragmentation: %d\n",total_internal_fragment, total_external_fragment);
		while(1)
		{
			char choice,dummy;
			printf("Do you want to continue? (Y/n) : ");
			getchar();
			scanf("%c",&choice);
			if(choice=='Y')
			{
				int id,new_memory_required,allocated_partition;
				bool allocated;
				printf("Enter id of the process leaving: ");
				scanf("%d",&id);
				printf("Enter memory required for the new processs: ");
				scanf("%d",&new_memory_required);
				int partition_to_remove = proc[id-1].allocated_partition;
				int memory_to_remove = proc[id-1].memory_required;
				printf("Partition to remove: %d\n", partition_to_remove);
				for(i=0;i<no_of_partitions;i++)
				{
					if(parti[i].id == partition_to_remove)
					{
						parti[i].alloted = false;
					}
				}
				allocated = false;
				for(j=0;j<no_of_partitions;j++)
				{
					if(new_memory_required<=parti[j].size &&
	!parti[j].alloted)
					{
						allocated = true;
						allocated_partition = parti[j].id;
						parti[j].internal_fragment = parti[j].size -
	new_memory_required;
						parti[j].alloted=true;
						break;
```

```
                    }
                }
                if(allocated)
                {
                    printf("Process allocated in partition:
%d\n",allocated_partition );
                    printf("New internal Fragmentation for the partition
= %d\n",parti[j].internal_fragment );
                }
                else
                {
                    printf("Process cannot be allocated\n");
                }
            }
            else{
                return 0;
            }
        }
}
```

OUTPUT-



```
Enter total memory: 80
Enter number of partitions: 3
Enter memory for partition1: 20
Enter memory for partition2: 40
Enter memory for partition3: 10
Size of partition4: 10
Enter number of processes: 3
Enter memory required for process1: 30
Enter memory required for process2: 20
Enter memory required for process3: 10
ProcessID       Memory required Allocated       Allocated Partition
1               30                      YES     2
2               20                      YES     1
3               10                      YES     3

Internal Fragmentation
3 0
4 ---
1 0
2 10
Total internal Fragmentation: 10
Total external Fragmentation: 10
Do you want to continue? (Y/n) : Y
Enter id of the process leaving: 1
Enter memory required for the new processs: 20
Partition to remove: 2
Process allocated in partition: 2
New internal Fragmentation for the partition = 20
Do you want to continue? (Y/n) : Y
Enter id of the process leaving: 2
Enter memory required for the new processs: 80
Partition to remove: 1
Process cannot be allocated
Do you want to continue? (Y/n) : n

Process returned 0 (0x0)   execution time : 58.706 s
Press ENTER to continue.
```

C)WORST FIT-

ALGORITHM-

1)Input memory block with a size.
2)Input process with size.
3)Initialize by selecting each process to find the maximum block size that can be assigned to the current process.
4)If the condition does not fulfill, they leave the process.
5)If the condition is not fulfilled, then leave the process and check for the next process.
6)Stop.

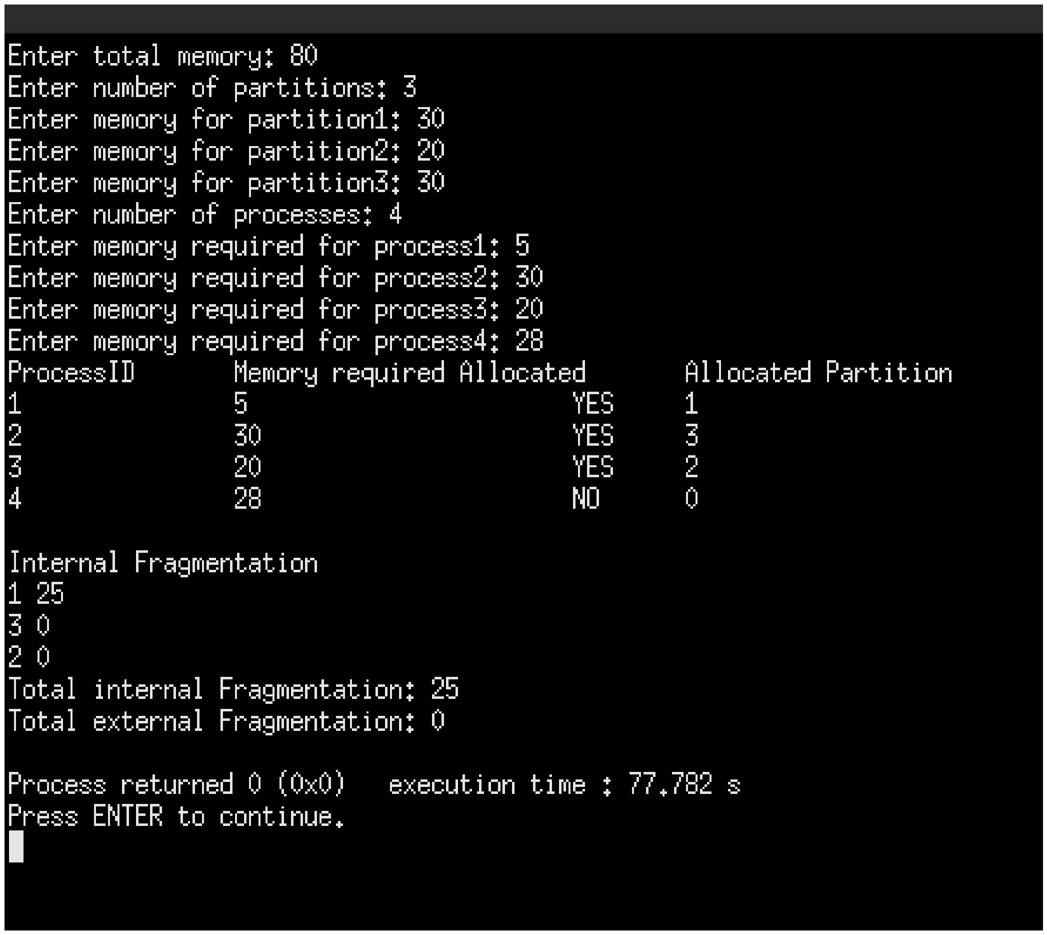
SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct process
{
	int id;
	int memory_required;
	int allocated_partition;
	int external_fragment;
	bool allocated;
};
struct partition
{
	int id;
	int size;
	int internal_fragment;
	bool alloted;
};
int main()
{
	int memory,no_of_partitions,no_of_processes,i,j,sum=0;
	printf("Enter total memory: ");
	scanf("%d",&memory);
	printf("Enter number of partitions: ");
	scanf("%d",&no_of_partitions);
	struct partition parti[no_of_partitions];
	for(i=0;i<no_of_partitions;i++)
```

```
{
        printf("Enter memory for partition%d: ",i+1);
        scanf("%d",&parti[i].size);
        parti[i].id = i+1;
        parti[i].alloted=false;

        sum+=parti[i].size;
}
if(sum < memory)
{
        no_of_partitions++;
        parti[i].size = memory - sum;
        parti[i].id = i+1;
        parti[i].alloted=false;
        printf("Size of partition%d: %d\n",i+1,parti[i].size );
}
for(i=0;i<no_of_partitions-1;i++)
{
        for(j=0;j<no_of_partitions-1-i;j++)
        {
                if(parti[j].size<parti[j+1].size)
                {
                        struct partition temp = parti[j];
                        parti[j] = parti[j+1];
                        parti[j+1] = temp;
                }
        }
}
int total_internal_fragment=0, total_external_fragment=0;
printf("Enter number of processes: ");
scanf("%d",&no_of_processes);
struct process proc[no_of_processes];
for (i = 0; i < no_of_processes; ++i)
{
        printf("Enter memory required for process%d: ",i+1 );
        scanf("%d",&proc[i].memory_required);
        proc[i].id = i+1;
        proc[i].external_fragment=0;
        proc[i].allocated = false;
        for(j=0;j<no_of_partitions;j++)
        {
```

```
                        if(proc[i].memory_required<=parti[j].size &&
!parti[j].alloted)
                        {
                                proc[i].allocated = true;
                                proc[i].allocated_partition = parti[j].id;
                                parti[j].internal_fragment = parti[j].size -
proc[i].memory_required;
                                total_internal_fragment+=parti[j].internal_fragment;
                                parti[j].alloted=true;
                                break;
                        }
                }
                for(j=0;j<no_of_partitions;j++)
                {
                        if(parti[j].alloted==false)
                        {
                                proc[i].external_fragment+=parti[j].size;
                        }
                }
        }
        for(j=0;j<no_of_partitions;j++)
        {
                if(!parti[j].alloted)
                {
                        total_external_fragment+=parti[j].size;
                        parti[j].internal_fragment = -1;
                }
        }
        printf("ProcessID\tMemory required\tAllocated\tAllocated
Partition\n");
        for(i=0;i<no_of_processes;i++)
        {
                if(proc[i].allocated)
                {
                        printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id,
proc[i].memory_required,  proc[i].allocated_partition);
                }
                else
                {
                        printf("%d\t\t%d\t\t\tNO\t%d\n",proc[i].id,
proc[i].memory_required, proc[i].external_fragment);
                }
        }
        printf("\n");
        printf("Internal Fragmentation\n");
        for(i=0;i<no_of_partitions;i++)
        {
                if(parti[i].internal_fragment==-1)
                {
                        printf("%d ---\n",parti[i].id);
                }
                else
                        printf("%d %d\n",parti[i].id,parti[i].internal_fragment );
        }
        printf("Total internal Fragmentation: %d\nTotal external
Fragmentation: %d\n",total_internal_fragment, total_external_fragment);
}
```

OUTPUT-



```
Enter total memory: 80
Enter number of partitions: 3
Enter memory for partition1: 30
Enter memory for partition2: 20
Enter memory for partition3: 30
Enter number of processes: 4
Enter memory required for process1: 5
Enter memory required for process2: 30
Enter memory required for process3: 20
Enter memory required for process4: 28
ProcessID       Memory required Allocated       Allocated Partition
1               5                       YES     1
2               30                      YES     3
3               20                      YES     2
4               28                      NO      0

Internal Fragmentation
1 25
3 0
2 0
Total internal Fragmentation: 25
Total external Fragmentation: 0

Process returned 0 (0x0)   execution time : 77.782 s
Press ENTER to continue.
```

Q2. Implement a program for page replacement using the following:

a. FIFO

b. LRU

c. OPTIMAL

A2.

a)FIFO

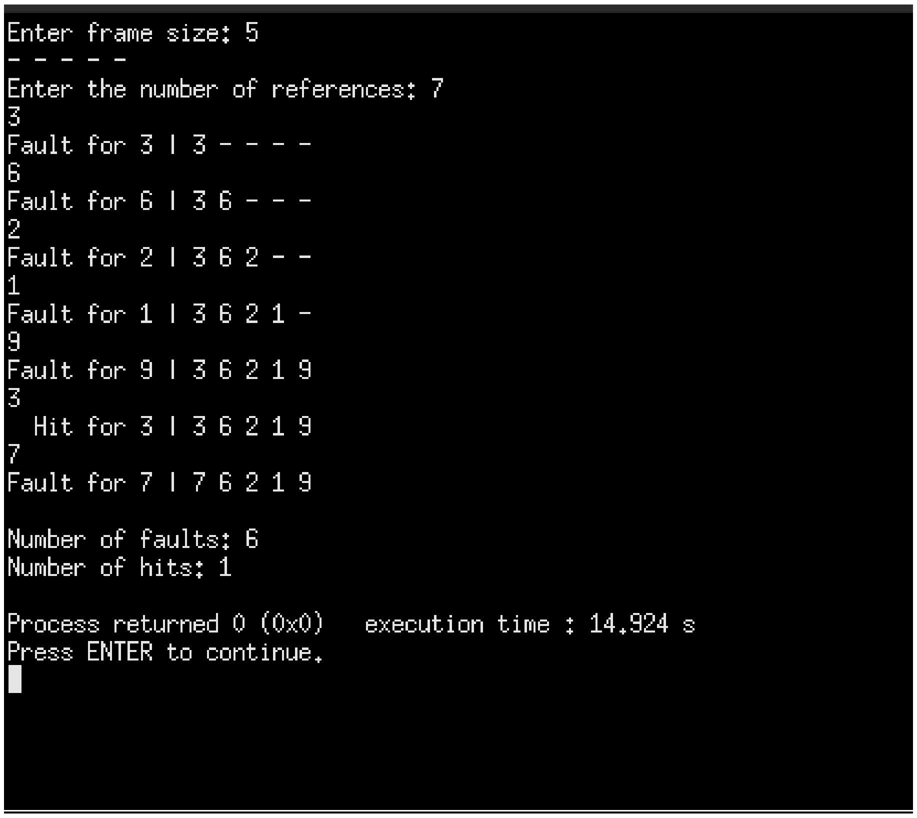
ALGORITHM-

```
Step 1. Start to traverse the pages.
Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
Step 3. Push pages in the queue one at a time until the queue reaches its
maximum capacity or all page requests are fulfilled.
Step 4. If the current page is present in the memory, do nothing.
Step 5. Else, pop the topmost page from the queue as it was inserted first.
Step 6. Replace the topmost page with the current page from the string.
Step 7. Increment the page faults.
Step 8. Stop
```

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

int pointer;
int faults ,hits;
void print(int frame_size,int frame[])
{
        int i;
        for(i=0;i<frame_size;i++)
        {
                if(frame[i]==-1)
                        printf("- ");
                else
                        printf("%d ",frame[i]);
        }

        printf("\n");
}

void add_reference(int frame_size,int frame[], int reference)
{
        int i;
        bool alloted = false;
        for(i=0;i<frame_size;i++)
        {
                if(frame[i]==reference)
                {
                        alloted = true;
                        printf("  Hit for %d | ", reference);
                        hits++;
                        break;
                }
                else if(frame[i]==-1)
                {
                        alloted = true;
                        frame[i] = reference;
                        printf("Fault for %d | ", reference);
                        faults++;
                        break;
                }
        }
        if(alloted == false)
        {
                faults++;
                printf("Fault for %d | ", reference);
                frame[pointer] = reference;
                pointer = (pointer+1)%frame_size;
        }
        print(frame_size, frame);
}

int main()
{
        int frame_size,i,number_of_references;
        printf("Enter frame size: ");
        scanf("%d",&frame_size);
        int frame[frame_size];
        for(i=0;i<frame_size;i++)
        {
                frame[i] = -1;
        }

        print(frame_size,frame);
        printf("Enter the number of references: ");
        scanf("%d",&number_of_references);
        int reference[number_of_references];

        for(i=0;i<number_of_references;i++)
        {
                scanf("%d",&reference[i]);
                add_reference(frame_size,frame,reference[i]);
        }
        printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );
}
```

OUTPUT-



```
Enter frame size: 5
- - - - -
Enter the number of references: 7
3
Fault for 3 | 3 - - - -
6
Fault for 6 | 3 6 - - -
2
Fault for 2 | 3 6 2 - -
1
Fault for 1 | 3 6 2 1 -
9
Fault for 9 | 3 6 2 1 9
3
  Hit for 3 | 3 6 2 1 9
7
Fault for 7 | 7 6 2 1 9

Number of faults: 6
Number of hits: 1

Process returned 0 (0x0)   execution time : 14.924 s
Press ENTER to continue.
```

b)LRU

ALGORITHM-

Step 1. Start the process
Step 2. Declare the page size
Step 3. Determine the number of pages to be inserted.
Step 4. Get the value.
Step 5. Declare the counter and stack value.
Step 6. Choose the least recently used page by the counter value.
Step 7. Stack them as per the selection.
Step 8. Display the values.
Step 9. Terminate the process.

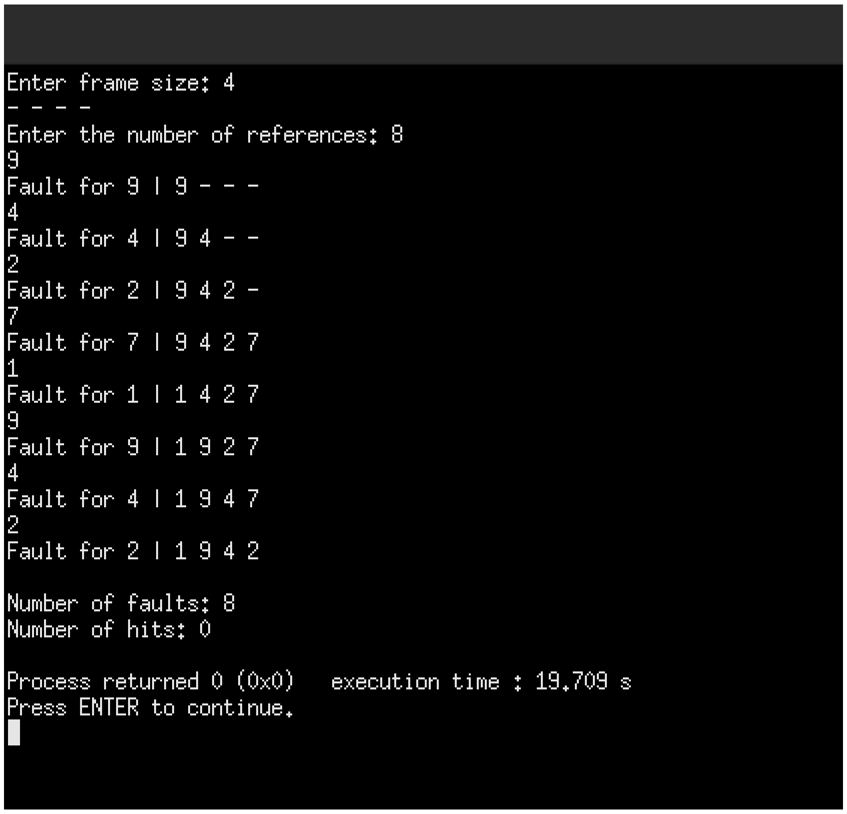
SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

int pointer;
int faults ,hits;
void print(int frame_size,int frame[])
{
        int i;
        //printf("Printing the Frames: ");
        for(i=0;i<frame_size;i++)
        {
                if(frame[i]==-1)
                        printf("- ");
                else
                        printf("%d ",frame[i]);
        }

        printf("\n");
}

int predict(int reference_length, int references[], int page_no ,int
frame_size,int frame[], int start)
{
        int pos = -1, farthest = start, i;
        for(i=0;i<frame_size;i++)
        {
                int j;
                for(j=start-1;j>=0;j--)
```

```
		{
			if(frame[i]==references[j])
			{
				if(j<farthest)
				{
					farthest=j;
					pos=i;
				}
				break;
			}
		}
		if(j==page_no)
			return i;
	}
	if(pos == -1)
		return 0;
	else
		return pos;
}

void add_reference(int frame_size,int frame[], int reference, int
current_position,int reference_length, int references[])
{
	int i;
	bool allocated=false;
	for(i=0;i<frame_size;i++)
	{

		if(frame[i]==reference)
		{
			printf("  Hit for %d | ", reference);
			hits++;
			allocated = true;
			break;
		}
		else if(frame[i]==-1)
		{
			frame[i] = reference;
			printf("Fault for %d | ", reference);
			faults++;
			allocated = true;
			break;
```

```
                }
        }
        if(allocated==false)
        {
                int j =
predict(reference_length,references,current_position,frame_size,frame,curren
t_position+1);

                frame[j] = reference;
                printf("Fault for %d | ", reference);
                faults++;
        }
        print(frame_size, frame);
}

int main()
{
        int frame_size,i,number_of_references;
        printf("Enter frame size: ");
        scanf("%d",&frame_size);
        int frame[frame_size];
        for(i=0;i<frame_size;i++)
        {
                frame[i] = -1;
        }

        print(frame_size,frame);

        printf("Enter the number of references: ");
        scanf("%d",&number_of_references);
        int reference[number_of_references];

        for(i=0;i<number_of_references;i++)
        {
                scanf("%d",&reference[i]);

        add_reference(frame_size,frame,reference[i],i,number_of_references,ref
erence);
        }
        printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );
}
```

OUTPUT-



```
Enter frame size: 4
- - - -
Enter the number of references: 8
9
Fault for 9 | 9 - - -
4
Fault for 4 | 9 4 - -
2
Fault for 2 | 9 4 2 -
7
Fault for 7 | 9 4 2 7
1
Fault for 1 | 1 4 2 7
9
Fault for 9 | 1 9 2 7
4
Fault for 4 | 1 9 4 7
2
Fault for 2 | 1 9 4 2

Number of faults: 8
Number of hits: 0

Process returned 0 (0x0)   execution time : 19.709 s
Press ENTER to continue.
```

c)OPTIMAL

ALGORITHM-

Step 1: Push the first page in the stack as per the memory demand.
Step 2:Push the second page as per the memory demand.
Step 3:Push the third page until the memory is full.
Step 4:As the queue is full, the page which is least recently used is popped.
Step 5:repeat step 4 until the page demand continues and until the processing is over.
Step 6:Terminate the program.

SOURCE CODE-

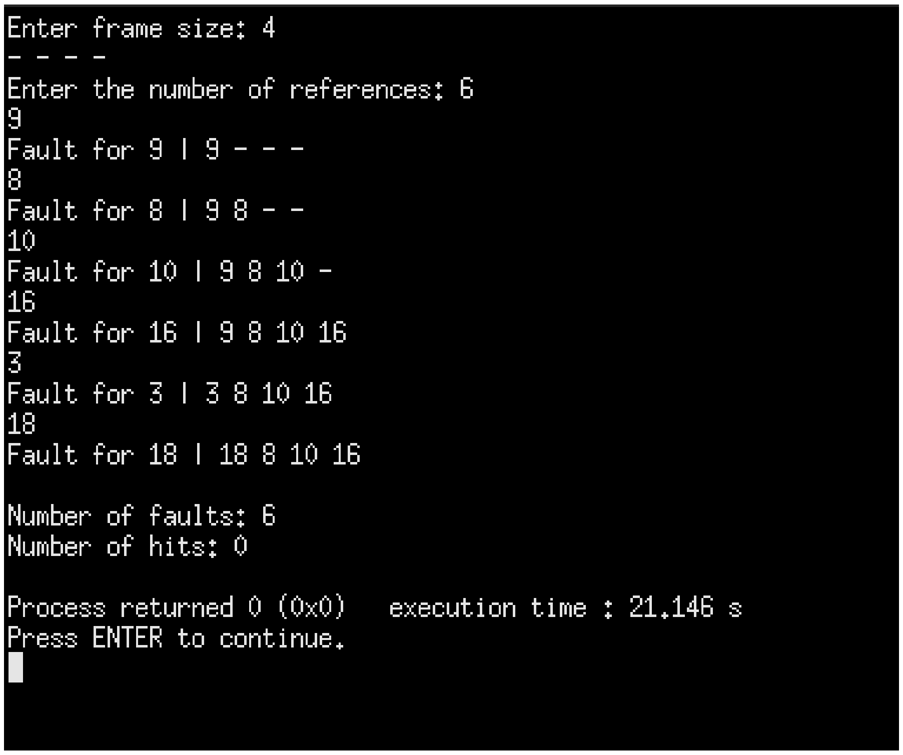
```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

int pointer;
int faults ,hits;
void print(int frame_size,int frame[])
```

```
{
        int i;
        for(i=0;i<frame_size;i++)
        {
                if(frame[i]==-1)
                        printf("- ");
                else
                        printf("%d ",frame[i]);
        }

        printf("\n");
}

int predict(int reference_length, int references[], int page_no ,int
frame_size,int frame[], int start)
{
        int pos = -1, farthest = start, i;
        for(i=0;i<frame_size;i++)
        {
                int j;
                for(j=start;j<reference_length;j++)
                {
                        if(frame[i]==references[j])
                        {
                                if(j>farthest)
                                {
                                        farthest=j;
                                        pos=i;
                                }
                                break;
                        }
                }
                if(j==page_no)
                        return i;
        }
        if(pos == -1)
                return 0;
        else
                return pos;
}
```

```
void add_reference(int frame_size,int frame[], int reference, int
current_position,int reference_length, int references[])
{
	int i;
	bool allocated=false;
	for(i=0;i<frame_size;i++)
	{

		if(frame[i]==reference)
		{
			printf("  Hit for %d | ", reference);
			hits++;
			allocated = true;
			break;
		}
		else if(frame[i]==-1)
		{
			frame[i] = reference;
			printf("Fault for %d | ", reference);
			faults++;
			allocated = true;
			break;
		}
	}
	if(allocated==false)
	{
		int j =
predict(reference_length,references,current_position,frame_size,frame,curren
t_position+1);

		frame[j] = reference;
		printf("Fault for %d | ", reference);
		faults++;
	}
	print(frame_size, frame);
}

int main()
{
	int frame_size,i,number_of_references;
	printf("Enter frame size: ");
	scanf("%d",&frame_size);
```

```
        int frame[frame_size];
        for(i=0;i<frame_size;i++)
        {
                frame[i] = -1;
        }

        print(frame_size,frame);

        printf("Enter the number of references: ");
        scanf("%d",&number_of_references);
        int reference[number_of_references];

        for(i=0;i<number_of_references;i++)
        {
                scanf("%d",&reference[i]);

        add_reference(frame_size,frame,reference[i],i,number_of_references,ref
erence);
        }
        printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );
}
```

OUTPUT-



```
Enter frame size: 4
- - - -
Enter the number of references: 6
9
Fault for 9 | 9 - - -
8
Fault for 8 | 9 8 - -
10
Fault for 10 | 9 8 10 -
16
Fault for 16 | 9 8 10 16
3
Fault for 3 | 3 8 10 16
18
Fault for 18 | 18 8 10 16

Number of faults: 6
Number of hits: 0

Process returned 0 (0x0)   execution time : 21.146 s
Press ENTER to continue.
```

Q3.Simulate with a program to schedule disk in seek optimization.

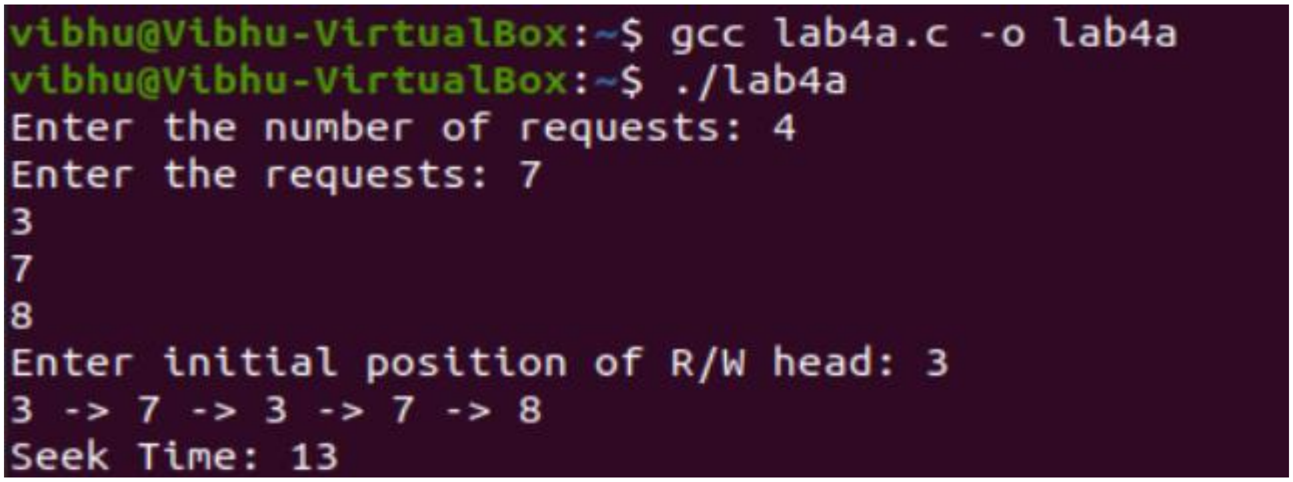
A3.

FCFS-

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"
int main()
{
	int i,no_of_requests,initial_head;
	printf("Enter the number of requests: ");
	scanf("%d",&no_of_requests);
	int request[no_of_requests];
	printf("Enter the requests: ");
	for (i = 0; i < no_of_requests; ++i)
	{
		scanf("%d",&request[i]);
	}
	printf("Enter initial position of R/W head: ");
	scanf("%d",&initial_head);
	int seek_time=0;
	printf("%d -> ",initial_head );
	for(i=0;i<no_of_requests;i++)
	{
		if(i == no_of_requests-1)
			printf("%d\n", request[i] );
		else
			printf("%d -> ", request[i] );
		seek_time += abs(request[i] - initial_head);
		initial_head = request[i];
	}
	printf("Seek Time: %d\n", seek_time);
}
```

OUTPUT-



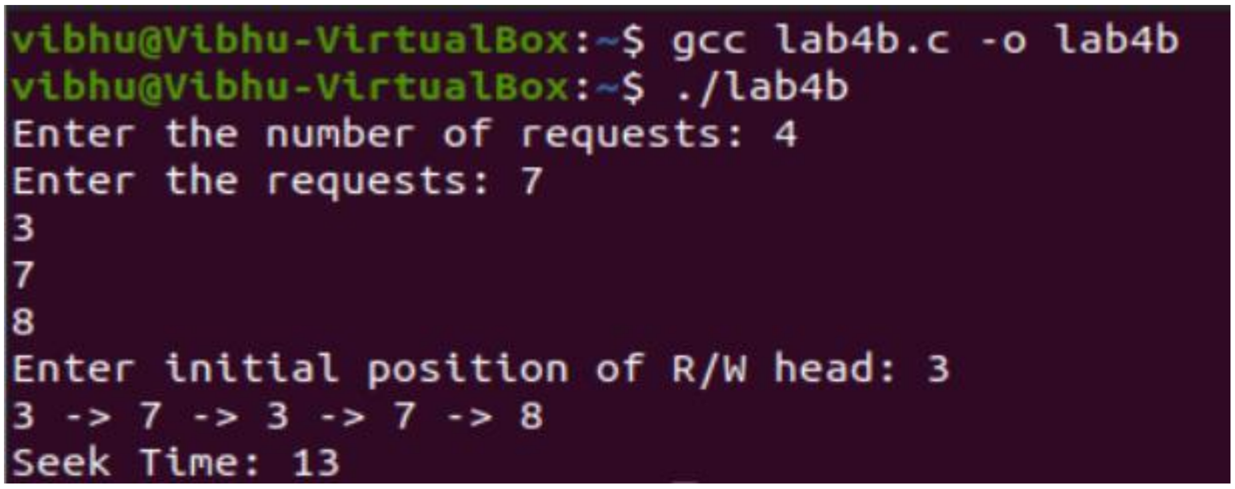
```
vibhu@Vibhu-VirtualBox:~$ gcc lab4a.c -o lab4a
vibhu@Vibhu-VirtualBox:~$ ./lab4a
Enter the number of requests: 4
Enter the requests: 7
3
7
8
Enter initial position of R/W head: 3
3 -> 7 -> 3 -> 7 -> 8
Seek Time: 13
```

SSTF-

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct request
{
        int request_track_number;
        bool visited;
};

int main()
{
        int i,no_of_requests,initial_head,limit,j,choice,previous_head;
        printf("Enter the number of requests: ");
        scanf("%d",&no_of_requests);
        struct request req[no_of_requests];
        printf("Enter the requests: ");
        for (i = 0; i < no_of_requests; ++i)
        {
                scanf("%d",&req[i].request_track_number);
                req[i].visited = false;
        }
        printf("Enter initial position of R/W head: ");
        scanf("%d",&initial_head);

        int seek_time=0;
        printf("%d -> ",initial_head );
        int n = no_of_requests;
        while(n)
        {
                int min = 1e9;
                int min_track_number, position;
                for(i=0;i<no_of_requests;i++)
                {
                        if(abs(initial_head - req[i].request_track_number) < min &&
req[i].visited == false)
                        {
                                min = abs(initial_head - req[i].request_track_number);
                                min_track_number = req[i].request_track_number;
                                position = i;
                        }
                }
                initial_head = req[position].request_track_number;
                req[position].visited = true;
                printf("%d ->",min_track_number);
                seek_time += min;
                n--;
        }

        printf("\nSeek Time: %d\n", seek_time);
}
```

OUTPUT-



```
vibhu@Vibhu-VirtualBox:~$ gcc lab4b.c -o lab4b
vibhu@Vibhu-VirtualBox:~$ ./lab4b
Enter the number of requests: 4
Enter the requests: 7
3
7
8
Enter initial position of R/W head: 3
3 -> 7 -> 3 -> 7 -> 8
Seek Time: 13
```

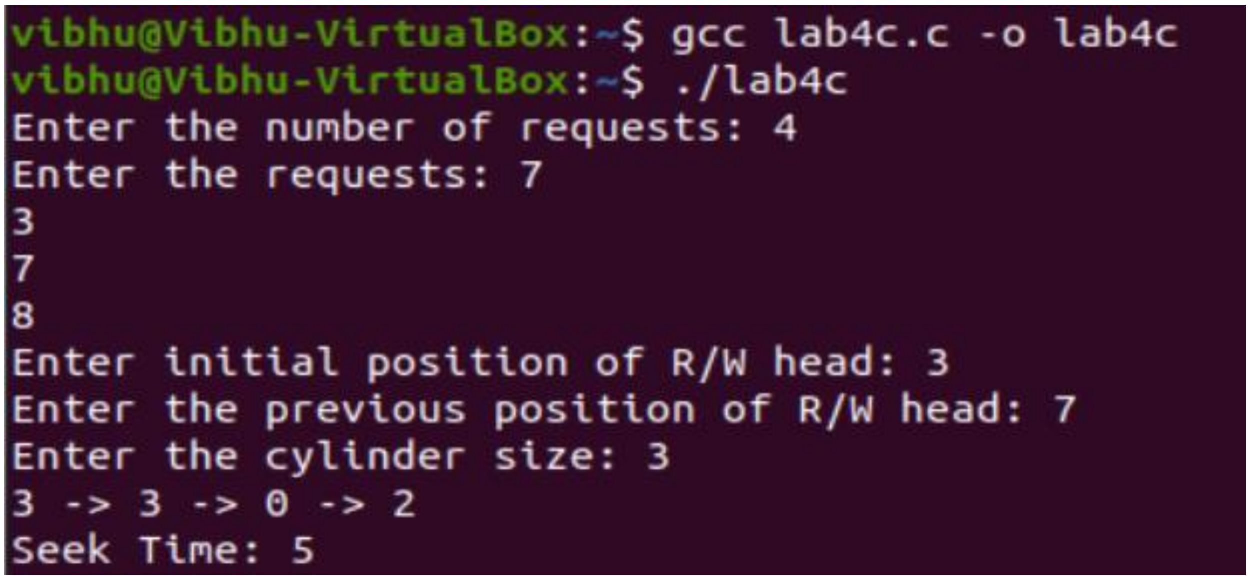
SCAN-

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct request
{
	int request_track_number;
	bool visited;
};

int main()
{
	int i,no_of_requests,initial_head,limit,j,choice,previous_head;
	printf("Enter the number of requests: ");
	scanf("%d",&no_of_requests);
	struct request req[no_of_requests];
	printf("Enter the requests: ");
	for (i = 0; i < no_of_requests; ++i)
	{
		scanf("%d",&req[i].request_track_number);
		req[i].visited = false;
	}
	printf("Enter initial position of R/W head: ");
	scanf("%d",&initial_head);

	printf("Enter the previous position of R/W head: ");
	scanf("%d",&previous_head);

	printf("Enter the cylinder size: ");
	scanf("%d",&limit);

	if(previous_head - initial_head > 0 )
	{
		choice = 2;
	}
	else
		choice = 1;
	//scanf("%d",&choice);
	int seek_time=0;
	printf("%d -> ",initial_head );
```

```
        if(choice == 1)
        {
                for(i=initial_head;i<limit;i++)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("%d -> ", limit-1);
                seek_time += abs(limit-1 - initial_head);
                initial_head = limit-1;
                for(i=initial_head;i>=0;i--)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                seek_time += abs(initial_head - 0);
                printf("0 \n");
        }
        else if(choice == 2)
        {
                for(i=initial_head;i>=0;i--)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("%d -> ", 0);
                seek_time += abs(0 - initial_head);
                initial_head = 0;
                for(i=initial_head;i<limit;i++)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                seek_time += abs(limit-1 - initial_head );
                printf("%d \n", limit-1);

        }
        printf("Seek Time: %d\n", seek_time);
}
```

OUTPUT-



```
vibhu@Vibhu-VirtualBox:~$ gcc lab4c.c -o lab4c
vibhu@Vibhu-VirtualBox:~$ ./lab4c
Enter the number of requests: 4
Enter the requests: 7
3
7
8
Enter initial position of R/W head: 3
Enter the previous position of R/W head: 7
Enter the cylinder size: 3
3 -> 3 -> 0 -> 2
Seek Time: 5
```

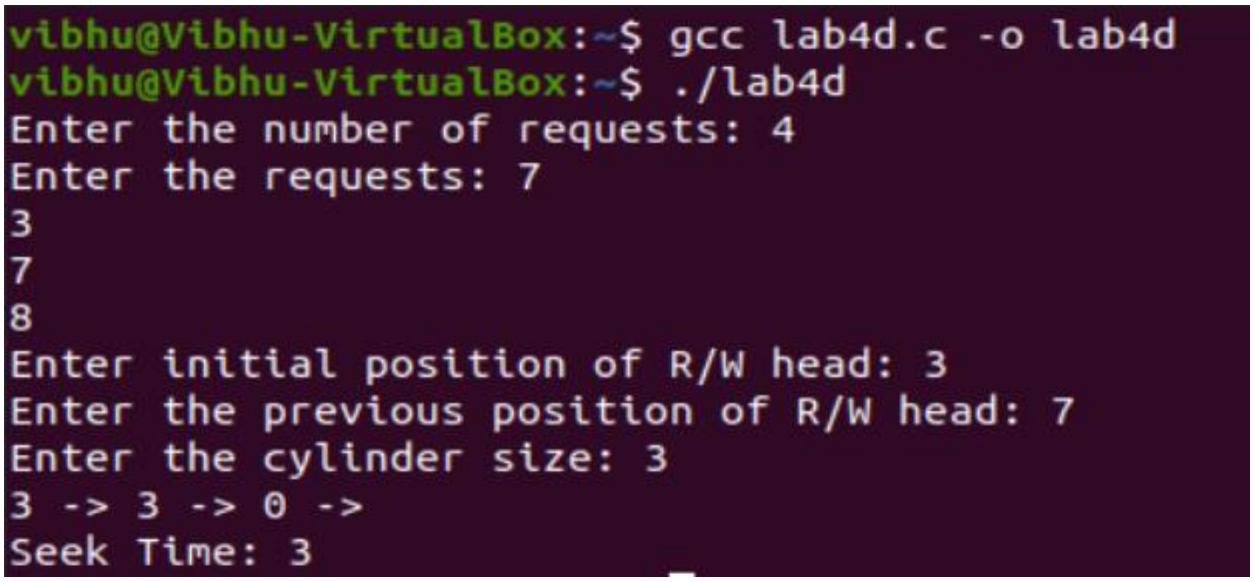
C-SCAN-

SOURCE CODE-

```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct request
{
	int request_track_number;
	bool visited;
};

int main()
{
	int i,no_of_requests,initial_head,limit,j,choice,previous_head;
	printf("Enter the number of requests: ");
	scanf("%d",&no_of_requests);
	struct request req[no_of_requests];
	printf("Enter the requests: ");
	for (i = 0; i < no_of_requests; ++i)
	{
		scanf("%d",&req[i].request_track_number);
		req[i].visited = false;
	}
	printf("Enter initial position of R/W head: ");
	scanf("%d",&initial_head);
```

```
        printf("Enter the previous position of R/W head: ");
        scanf("%d",&previous_head);

        printf("Enter the cylinder size: ");
        scanf("%d",&limit);

        if(previous_head - initial_head > 0 )
        {
                choice = 2;
        }
        else
                choice = 1;
        //scanf("%d",&choice);
        int seek_time=0;
        printf("%d -> ",initial_head );
        int cp_initial_head = initial_head;
        if(choice == 1)
        {
                for(i=initial_head;i<limit;i++)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("%d -> \n", limit-1);
                seek_time += abs(limit-1 - initial_head);
                initial_head = 0;
                for(i=0;i<cp_initial_head;i++)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("\n");
        }
        else if(choice == 2)
        {
                for(i=initial_head;i>=0;i--)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("%d -> ", 0 );
                seek_time += abs(initial_head - 0);
                initial_head = limit-1;
                for(i=limit;i>cp_initial_head;i--)
                {
                        for(j=0;j<no_of_requests;j++)
                        {
                                if(req[j].request_track_number == i && req[j].visited
== false)
                                {
                                        printf("%d -> ", req[j].request_track_number);
                                        req[j].visited = true;
                                        seek_time += abs(req[j].request_track_number -
initial_head);
                                        initial_head = req[j].request_track_number;
                                }
                        }
                }
                printf("\n");
        }
        printf("Seek Time: %d\n", seek_time);
}
```

OUTPUT-



```
vibhu@Vibhu-VirtualBox:~$ gcc lab4d.c -o lab4d
vibhu@Vibhu-VirtualBox:~$ ./lab4d
Enter the number of requests: 4
Enter the requests: 7
3
7
8
Enter initial position of R/W head: 3
Enter the previous position of R/W head: 7
Enter the cylinder size: 3
3 -> 3 -> 0 ->
Seek Time: 3
```

C-LOOK-

SOURCE CODE-

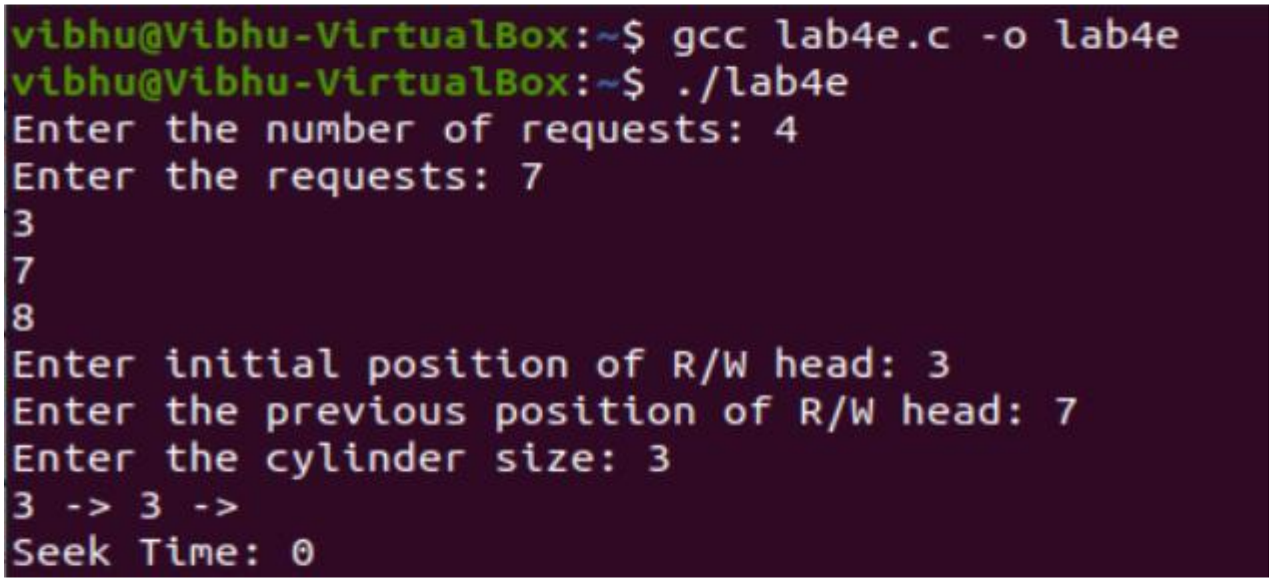
```
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

struct request
{
        int request_track_number;
        bool visited;
};

int main()
{
        int i,no_of_requests,initial_head,limit,j,choice,previous_head;
```

```
	printf("Enter the number of requests: ");
	scanf("%d",&no_of_requests);
	struct request req[no_of_requests];
	printf("Enter the requests: ");
	for (i = 0; i < no_of_requests; ++i)
	{
		scanf("%d",&req[i].request_track_number);
		req[i].visited = false;
	}
	printf("Enter initial position of R/W head: ");
	scanf("%d",&initial_head);

	printf("Enter the previous position of R/W head: ");
	scanf("%d",&previous_head);

	printf("Enter the cylinder size: ");
	scanf("%d",&limit);

	if(previous_head - initial_head > 0 )
	{
		choice = 2;
	}
	else
		choice = 1;
	int seek_time=0;
	printf("%d -> ",initial_head );
	int cp_initial_head = initial_head;
	if(choice == 1)
	{
		for(i=initial_head;i<limit;i++)
		{
			for(j=0;j<no_of_requests;j++)
			{
				if(req[j].request_track_number == i && req[j].visited
== false)
				{
					printf("%d -> ", req[j].request_track_number);
					req[j].visited = true;
					seek_time += abs(req[j].request_track_number -
initial_head);
					initial_head = req[j].request_track_number;
				}
```

```
                    }
                }
                initial_head = 0;
                for(i=0;i<cp_initial_head;i++)
                {
                    for(j=0;j<no_of_requests;j++)
                    {
                        if(req[j].request_track_number == i && req[j].visited
== false)
                        {
                            printf("%d -> ", req[j].request_track_number);
                            req[j].visited = true;
                            seek_time += abs(req[j].request_track_number -
initial_head);
                            initial_head = req[j].request_track_number;
                        }
                    }
                }
                printf("\n");
            }
            else if(choice == 2)
            {
                for(i=initial_head;i>=0;i--)
                {
                    for(j=0;j<no_of_requests;j++)
                    {
                        if(req[j].request_track_number == i && req[j].visited
== false)
                        {
                            printf("%d -> ", req[j].request_track_number);
                            req[j].visited = true;
                            seek_time += abs(req[j].request_track_number -
initial_head);
                            initial_head = req[j].request_track_number;
                        }
                    }
                }
                initial_head = limit-1;
                for(i=limit;i>cp_initial_head;i--)
                {
                    for(j=0;j<no_of_requests;j++)
                    {
```

```
                                        if(req[j].request_track_number == i && req[j].visited
== false)
                                        {
                                                printf("%d -> ", req[j].request_track_number);
                                                req[j].visited = true;
                                                seek_time += abs(req[j].request_track_number -
initial_head);
                                                initial_head = req[j].request_track_number;
                                        }
                                }
                        }
                        printf("\n");
                }
                printf("Seek Time: %d\n", seek_time);
        }
```

OUTPUT-



```
vibhu@Vibhu-VirtualBox:~$ gcc lab4e.c -o lab4e
vibhu@Vibhu-VirtualBox:~$ ./lab4e
Enter the number of requests: 4
Enter the requests: 7
3
7
8
Enter initial position of R/W head: 3
Enter the previous position of R/W head: 7
Enter the cylinder size: 3
3 -> 3 ->
Seek Time: 0
```