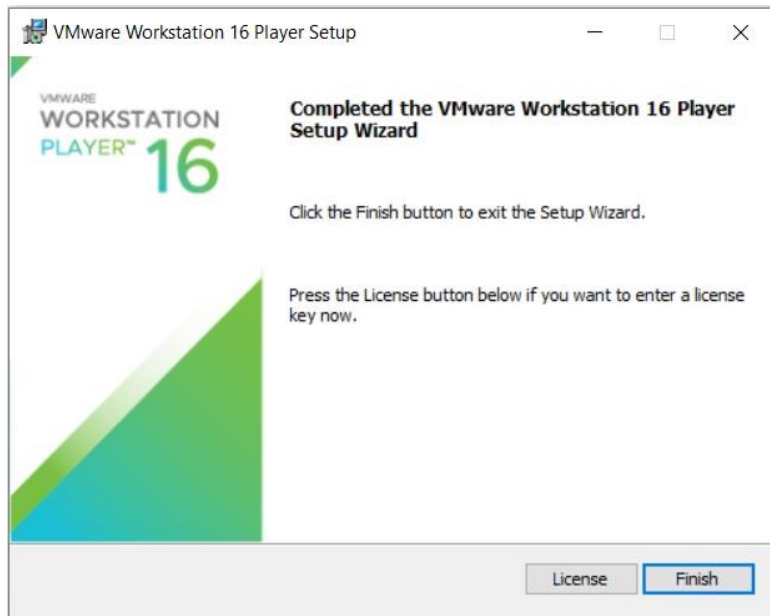


VMWARE WORKSTATION PLAYER

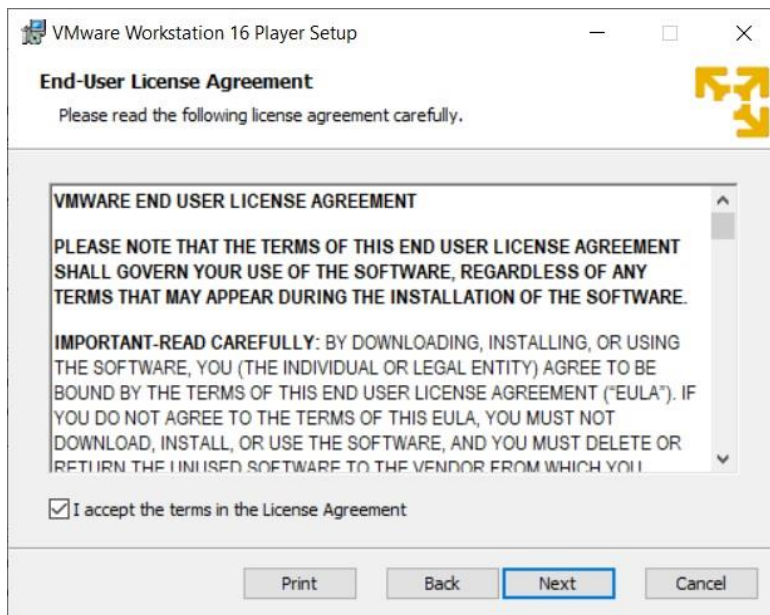
VMware Workstation Player is an ideal utility for running a single virtual machine on a Windows or Linux PC. Otherwise, we can easily run multiple operating systems as virtual machines on any Windows or Linux PC with VMware Workstation Player. Organizations use Workstation Player to deliver managed corporate desktops, while students and educators use it for learning and training.

VMWARE WORKSTATION PLAYER INSTALLATION STEPS

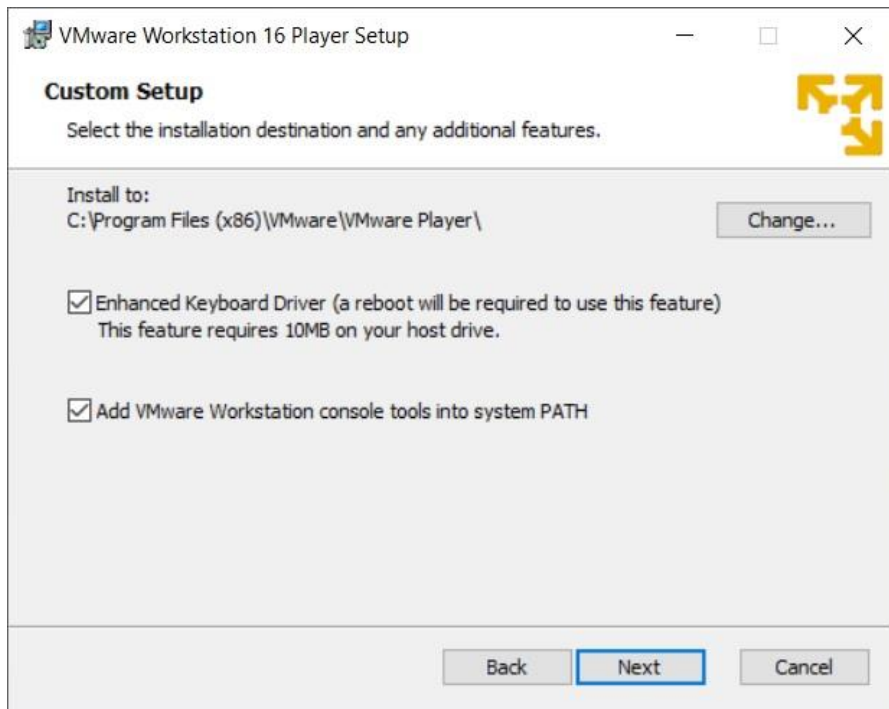
Run the installation wizard.



Click next and accept the license terms and click next again to move on to the next screen.



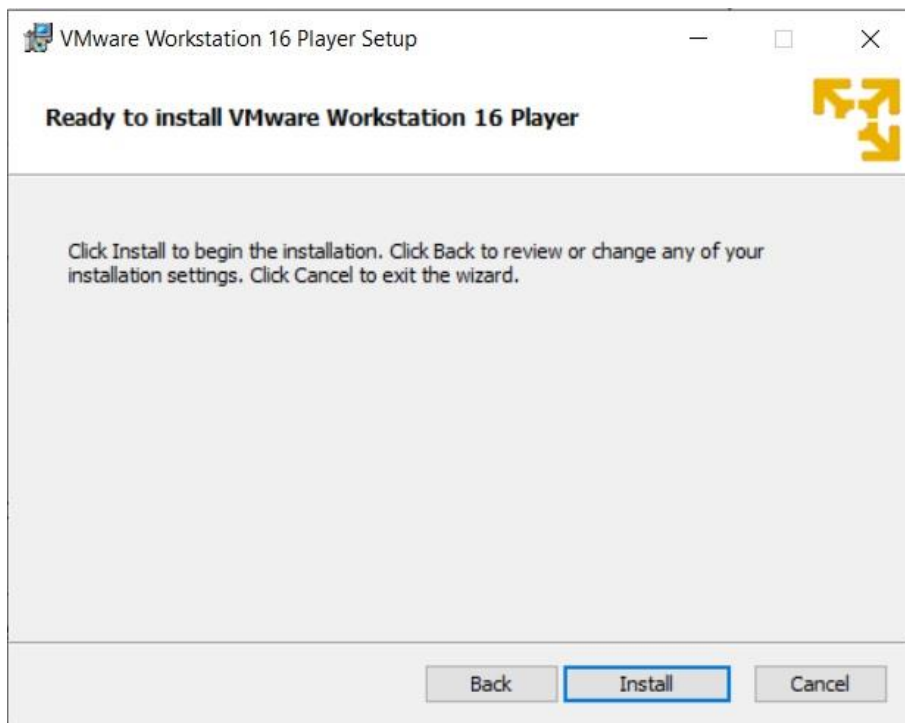
Custom setup – Enhanced Keyboard driver and Installation directory



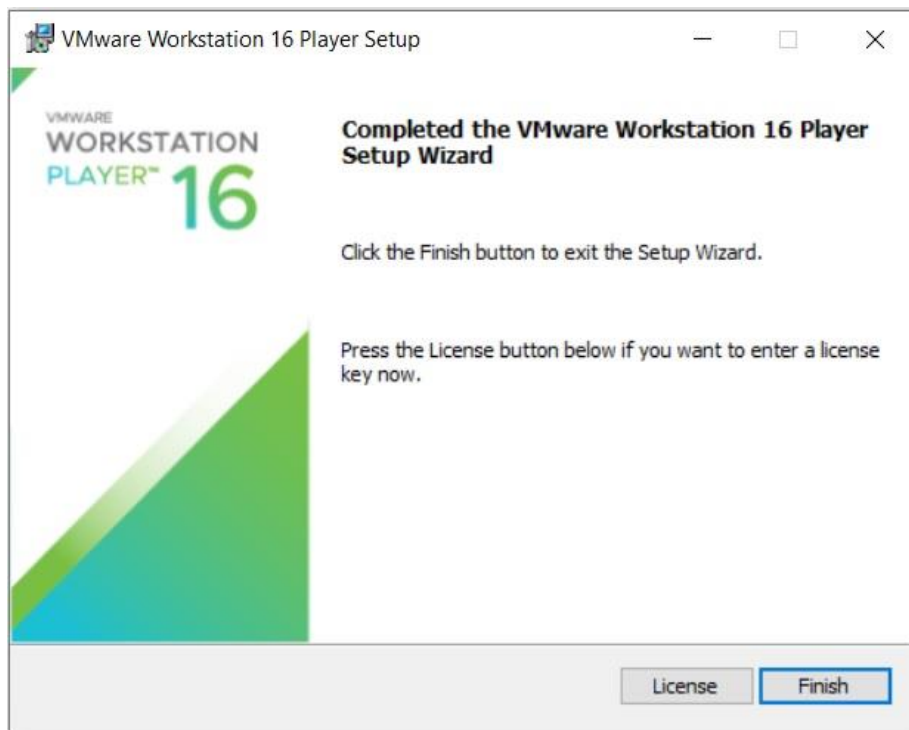
Ready to install

Now the installation wizard is ready to install. Click on install to begin the installation.

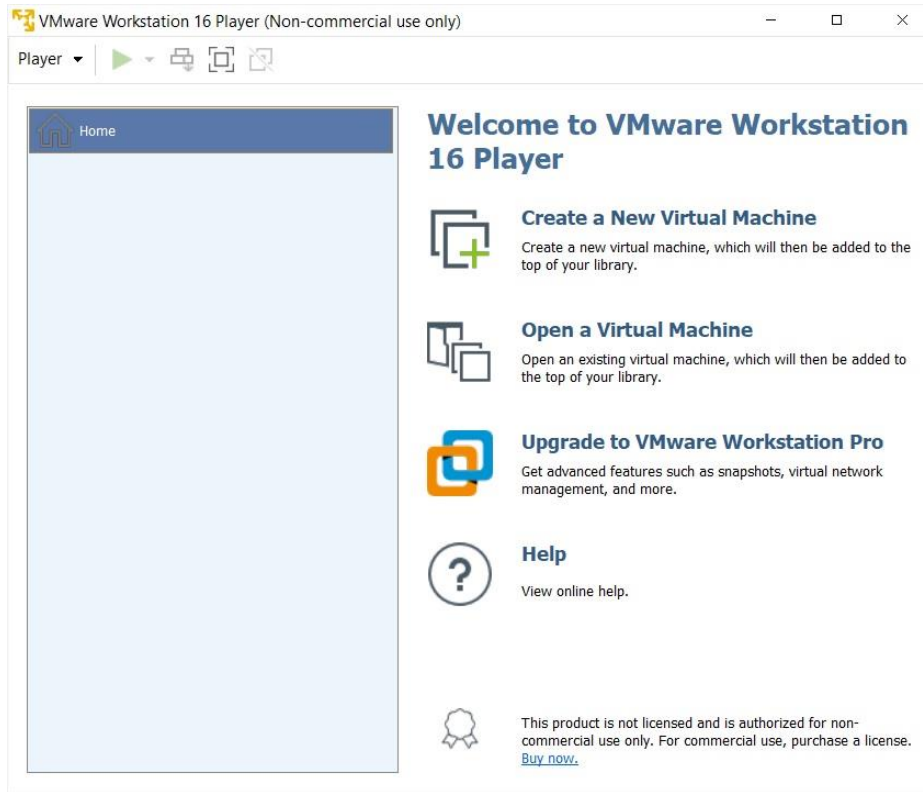
Installation begins, wait for it to complete.



After sometime, you will see installation complete message.



Now you will see VMware Workstation Player 16 ready to be used for free for non-commercial purpose.



VMware Workstation Player is ready to use.

LAB NO. 1

Thread in Operating System

A thread is the smallest unit of processing that can be performed in an OS. Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks. Thread is often referred to as a lightweight process.

Program for Single Threaded Process

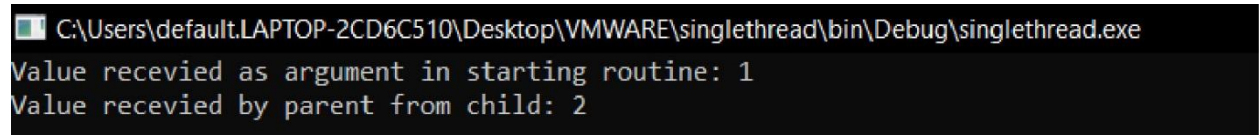
```
#include <stdio.h>
#include <string.h>
#include <pthread.h>

int i = 2;

void* foo(void* p){
    printf("Value received as argument in starting routine: ");
    printf("%i\n", * (int*)p);
    pthread_exit(&i);
}

int main(void){
    pthread_t id;
    int j = 1;
    pthread_create(&id, NULL, foo, &j);
    int* ptr;
    pthread_join(id, (void**)&ptr);
    printf("Value received by parent from child: ");
    printf("%i\n", *ptr);
}
```

OUTPUT:



```
C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\singlethread\bin\Debug\singlethread.exe
Value received as argument in starting routine: 1
Value received by parent from child: 2
```

Program for Multithreaded Process

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

int g = 0;

void *myThreadFun(void *vargp){
    int *myid = (int *)vargp;
    static int s = 0;
    ++s; ++g;
}
```

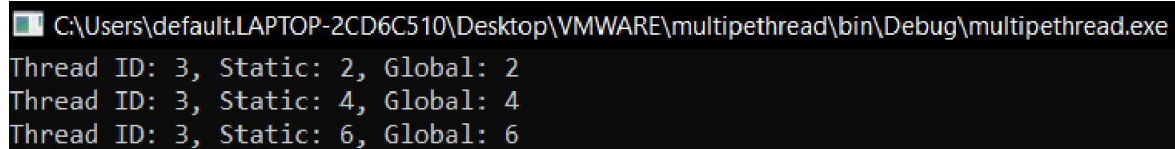
```

        printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
    }

int main(){
    int i;
    pthread_t tid;
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
    pthread_exit(NULL);
    return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\multipethread\bin\Debug\multipethread.exe
Thread ID: 3, Static: 2, Global: 2
Thread ID: 3, Static: 4, Global: 4
Thread ID: 3, Static: 6, Global: 6

```

LAB NO. 2

Process Scheduling in Operating System

Process Scheduling is the process of removing the running task from the processor and selecting another task for processing. It schedules a process into different states like ready, waiting, and running. This allows you to get the minimum response time for programs.

Program for Process Scheduling

```

#include<stdio.h>

void findWaitingTime(
int processes[], int n,int bt[], int wt[]){
    wt[0] = 0;
    for (int i = 1; i < n ; i++ ){
        wt[i] = bt[i-1] + wt[i-1];
    }
}

void findTurnAroundTime( int processes[], int n,
    int bt[], int wt[], int tat[]){
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

void findavgTime( int processes[], int n, int bt[]){
    int wt[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];
    }
}

```

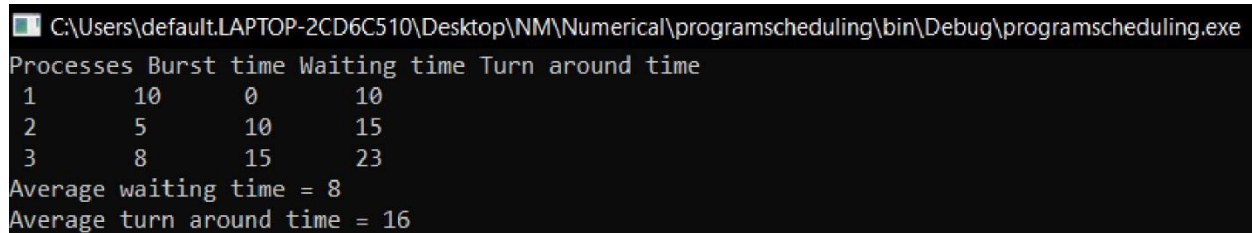
```

        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];
    int burst_time[] = {10, 5, 8};
    findavgTime(processes, n, burst_time);
    return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\NM\Numerical\program scheduling\bin\Debug\program scheduling.exe
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

```

LAB NO. 3

Dining Philosophers Problem in OS

The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Program for Dining Philosophers Problem

```

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1

```

```

#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];

void test(int phnum){
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n", phnum + 1,
LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        sem_post(&S[phnum]);
    }
}

void take_fork(int phnum){
    sem_wait(&mutex);
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    test(phnum);
    sem_post(&mutex);
    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum){
    sem_wait(&mutex);
    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n", phnum + 1,
LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void* philosopher(void* num){
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

```

```

int main(){
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);
}

```

OUTPUT:

```

C:\Users\default.LAPTOP-2CD6C510\Desktop\NM\Numerical\PhilosopherProblem\bin\Debug\PhilosopherProblem.exe
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 2 is Hungry
Philosopher 4 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking

```


LAB NO. 4

Banker's Algorithm in OS

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Program for Banker's Algorithm

```
#include <stdio.h>

int main(){
    int n, m, i, j, k;
    n = 5;
    m = 3;
    int alloc[5][3] = { { 0, 1, 0 },
                        { 2, 0, 0 },
                        { 3, 0, 2 },
                        { 2, 1, 1 },
                        { 0, 0, 2 } };

    int max[5][3] = { { 7, 5, 3 },
                     { 3, 2, 2 },
                     { 9, 0, 2 },
                     { 2, 2, 2 },
                     { 4, 3, 3 } };

    int avail[3] = { 3, 3, 2 };
    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                        break;
                    }
                }
                if (flag == 0) {
                    ans[ind++] = i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
}
```

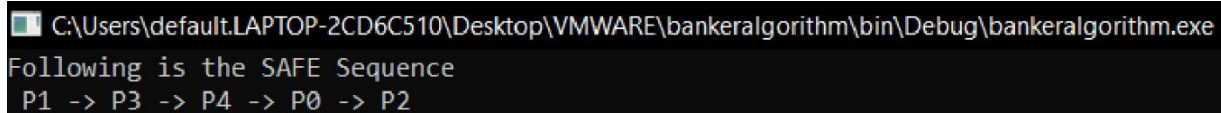
```

        }
    }
}

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

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\bankeralgorithm\bin\Debug\bankeralgorithm.exe
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2

```

LAB NO. 5

Page Replacement in Operating System

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).

Program for Page Replacement in Operating System

```

#include <stdio.h>

int main(){
    int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
    printf("\nEnter the number of Pages: ");
    scanf("%d", &pages);
    printf("\nEnter reference string values:\n");
    for( m = 0; m < pages; m++){
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &referenceString[m]);
    }
}

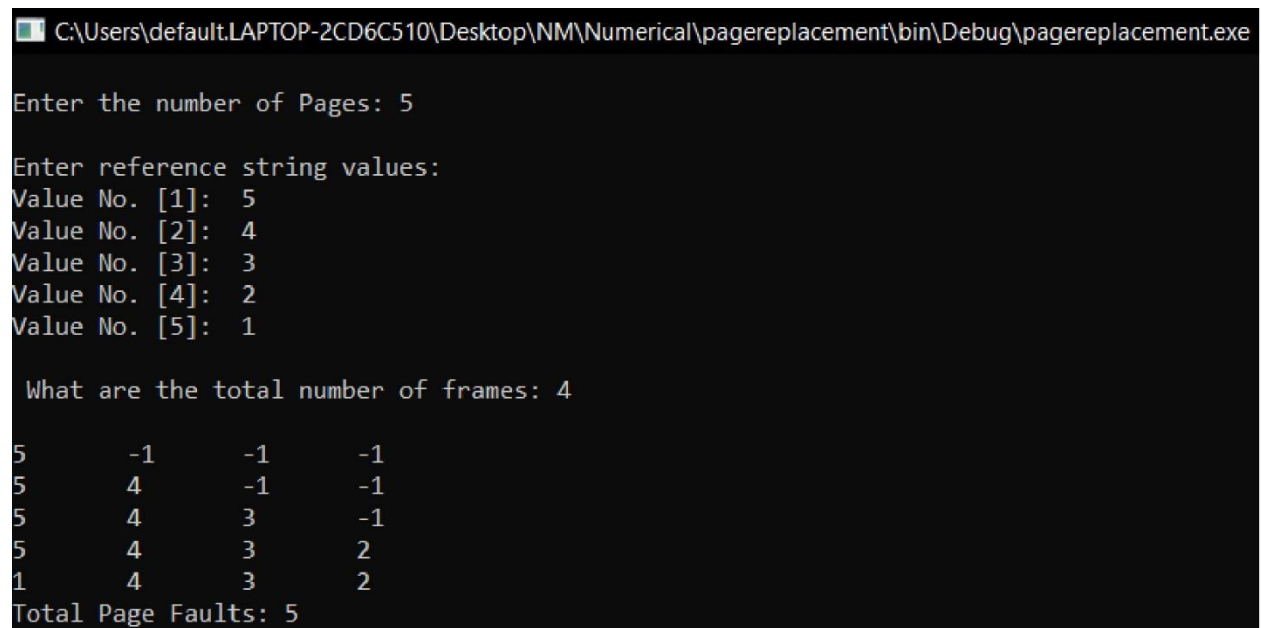
```

```

printf("\n What are the total number of frames: ");
{
    scanf("%d", &frames);
}
int temp[frames];
for(m = 0; m < frames; m++){
    temp[m] = -1;
}
for(m = 0; m < pages; m++){
    s = 0;
    for(n = 0; n < frames; n++){
        if(referenceString[m] == temp[n])
        {
            s++;
            pageFaults--;
        }
    }
    pageFaults++;
    if((pageFaults <= frames) && (s == 0)){
        temp[m] = referenceString[m];
    }
    else if(s == 0){
        temp[(pageFaults - 1) % frames] = referenceString[m];
    }
    printf("\n");
    for(n = 0; n < frames; n++){
        printf("%d\t", temp[n]);
    }
}
printf("\nTotal Page Faults: %d\n", pageFaults);
return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\NM\Numerical\pagereplacement\bin\Debug\pagereplacement.exe
Enter the number of Pages: 5

Enter reference string values:
Value No. [1]: 5
Value No. [2]: 4
Value No. [3]: 3
Value No. [4]: 2
Value No. [5]: 1

What are the total number of frames: 4

5      -1      -1      -1
5       4      -1      -1
5       4       3      -1
5       4       3       2
1       4       3       2
Total Page Faults: 5

```

LAB NO. 6

Disk Scheduling in Operating System

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling. Disk scheduling is important because: Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller.

1. First Come First Serve (FCFS) Scheduling in OS

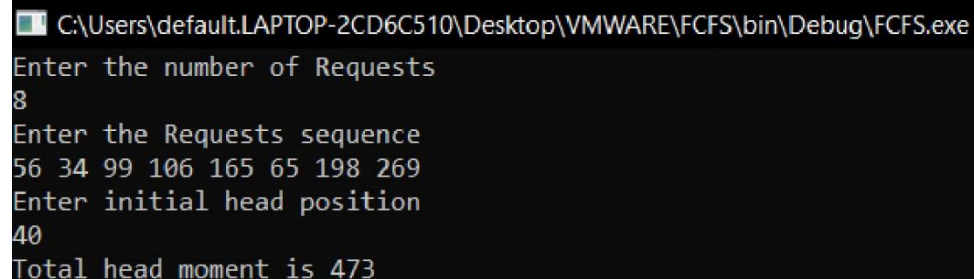
FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Program for First Come First Serve (FCFS)

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

int main(){
    int RQ[100],i,n,TotalHeadMovement=0,initial;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++){
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    for(i=0;i<n;i++){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    printf("Total head moment is %d",TotalHeadMovement);
    return 0;
}
```

OUTPUT:



```
C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\FCFS\bin\Debug\FCFS.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
40
Total head moment is 473
```

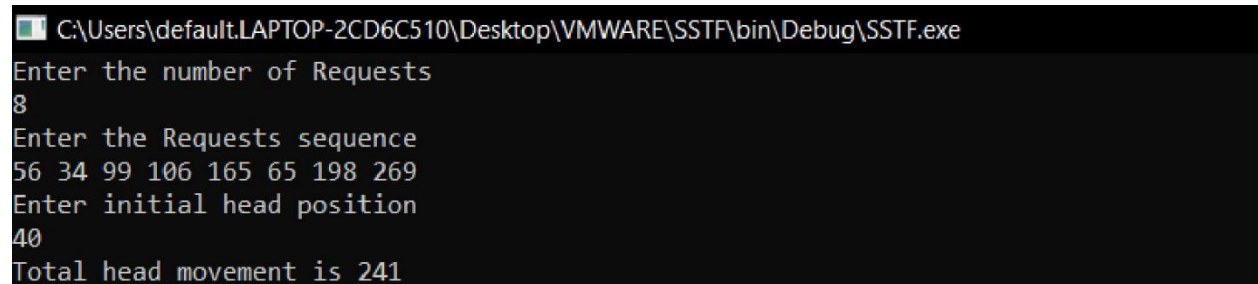
2. Shortest Seek Time First (SSTF) Scheduling in OS

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

Program for Shortest Seek Time First (SSTF)

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int RQ[100], i, n, TotalHeadMovement=0, initial, count=0;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for(i=0; i<n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);
    while(count!=n) {
        int min=1000, d, index;
        for(i=0; i<n; i++) {
            d=abs(RQ[i]-initial);
            if(min>d) {
                min=d;
                index=i;
            }
        }
        TotalHeadMovement=TotalHeadMovement+min;
        initial=RQ[index];
        RQ[index]=1000;
        count++;
    }
    printf("Total head movement is %d", TotalHeadMovement);
    return 0;
}
```

OUTPUT:



```
C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\SSTF\bin\Debug\SSTF.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
40
Total head movement is 241
```

3. Scan/ Elevator Disk Scheduling in OS

In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.

Program for Scan/ Elevator Disk Scheduling

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

int main(){
    int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++){
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);
    for(i=0;i<n;i++){
        {
            for(j=0;j<n-i-1;j++){
                if(RQ[j]>RQ[j+1]){
                    int temp;
                    temp=RQ[j];
                    RQ[j]=RQ[j+1];
                    RQ[j+1]=temp;
                }
            }
        }
        int index;
        for(i=0;i<n;i++){
            if(initial<RQ[i]){
                index=i;
                break;
            }
        }

        if(move==1){
            for(i=index;i<n;i++){
                TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
                initial=RQ[i];
            }
            TotalHeadMovement=TotalHeadMovement+abs(size-RQ[i-1]-1);
            initial = size-1;
            for(i=index-1;i>=0;i--){
                TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
                initial=RQ[i];
            }
        }
    }
}
```

```

    }
}

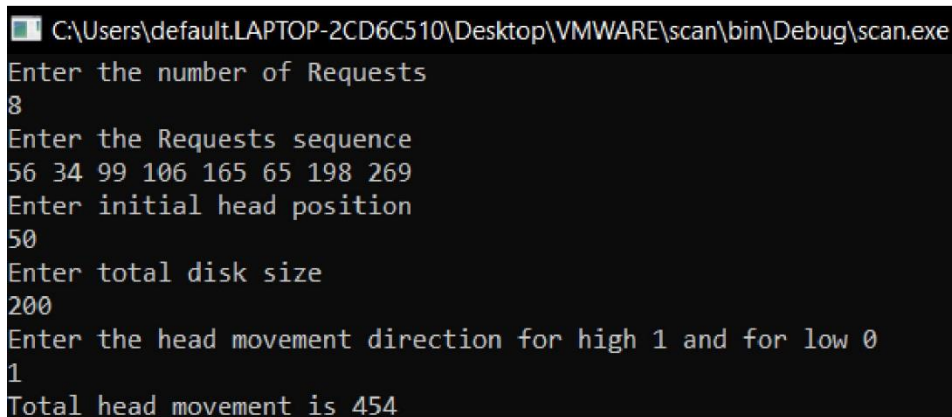
else{
    for(i=index-1;i>=0;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    TotalHeadMovement=TotalHeadMovement+abs(RQ[i+1]-0);
    initial =0;
    for(i=index;i<n;i++){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}

printf("Total head movement is %d",TotalHeadMovement);
return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\scan\bin\Debug\scan.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 454

```

4. C-SCAN Disk Scheduling in OS

In this algorithm, disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Program for C-SCAN Disk Scheduling

```

#include<stdio.h>
#include<stdlib.h>
int main(){
    int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");

```

```

for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
printf("Enter initial head position\n");
scanf("%d",&initial);
printf("Enter total disk size\n");
scanf("%d",&size);
printf("Enter the head movement direction for high 1 and for low
0\n");
scanf("%d",&move);
for(i=0;i<n;i++){
    for( j=0;j<n-i-1;j++){
        if(RQ[j]>RQ[j+1]){
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
        }
    }
}

int index;
for(i=0;i<n;i++){
    if(initial<RQ[i]){
        index=i;
        break;
    }
}
if(move==1){
    for(i=index;i<n;i++){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    TotalHeadMovement=TotalHeadMovement+abs(size-RQ[i-1]-1);
    TotalHeadMovement=TotalHeadMovement+abs(size-1-0);
    initial=0;
    for( i=0;i<index;i++){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
else{
    for(i=index-1;i>=0;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    TotalHeadMovement=TotalHeadMovement+abs(RQ[i+1]-0);
    TotalHeadMovement=TotalHeadMovement+abs(size-1-0);
    initial =size-1;
    for(i=n-1;i>=index;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
}

```

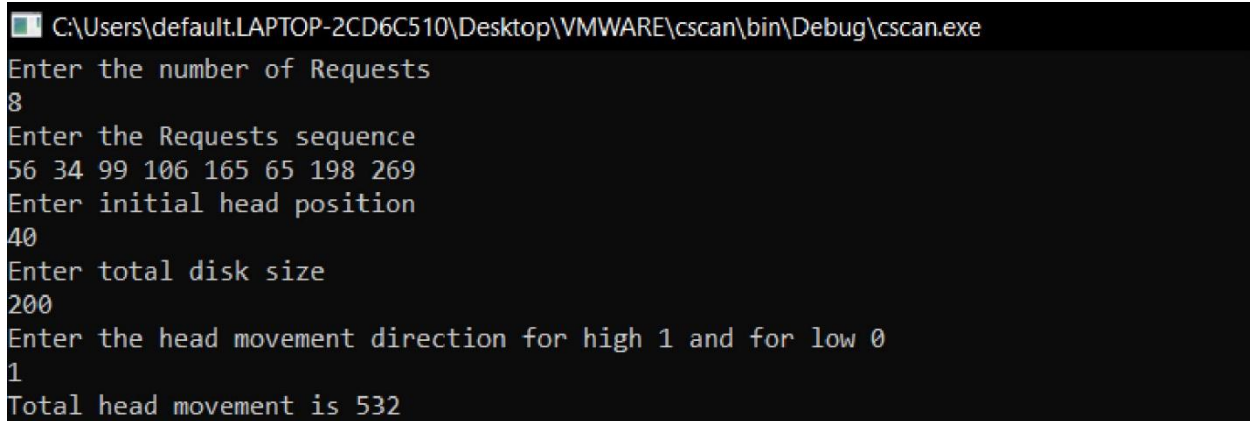


```

    printf("Total head movement is %d",TotalHeadMovement);
    return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\cscan\bin\Debug\cscan.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
40
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 532

```

5. Look Disk Scheduling in OS

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Program for Look Disk Scheduling

```

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

int main(){
    int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++){
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);
    for(i=0;i<n;i++){
        {
            for(j=0;j<n-i-1;j++){
                if(RQ[j]>RQ[j+1]){
                    int temp;

```

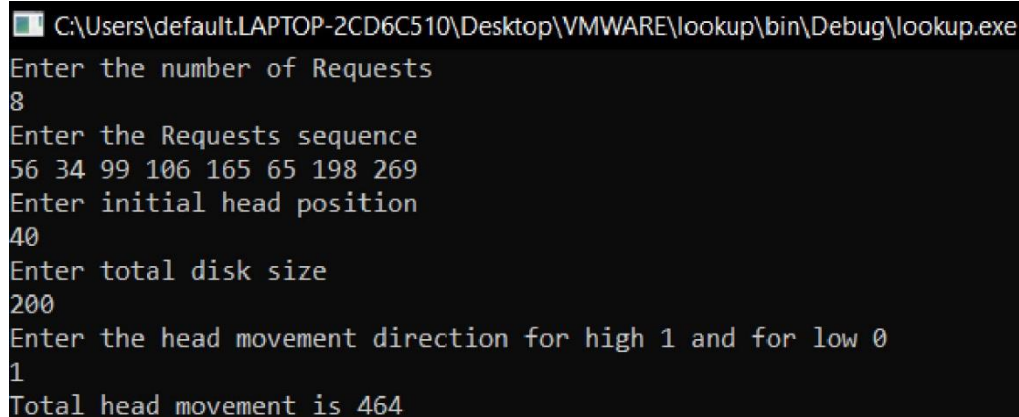
```

        temp=RQ[j];
        RQ[j]=RQ[j+1];
        RQ[j+1]=temp;
    }
}
int index;
for(i=0;i<n;i++){
    if(initial<RQ[i]){
        index=i;
        break;
    }
}
if(move==1){
    for(i=index;i<n;i++){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    for(i=index-1;i>=0;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
else{
    for(i=index-1;i>=0;i--){
        {
            TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        for(i=index;i<n;i++){
            TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }
}

printf("Total head movement is %d",TotalHeadMovement);
return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\lookup\bin\Debug\lookup.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
40
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 464

```

6. C-LOOK Disk Scheduling in OS

In C-LOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Program for C-LOOK Disk Scheduling

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

int main(){
    int RQ[100],i,j,n,TotalHeadMovement=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++){
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);
    for(i=0;i<n;i++){
        for(j=0;j<n-i-1;j++){
            if(RQ[j]>RQ[j+1]){
                int temp;
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }
        }
    }

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

    if(move==1){
        for(i=index;i<n;i++){
            TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        for(i=0;i<index;i++){
            TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }
}
```

```

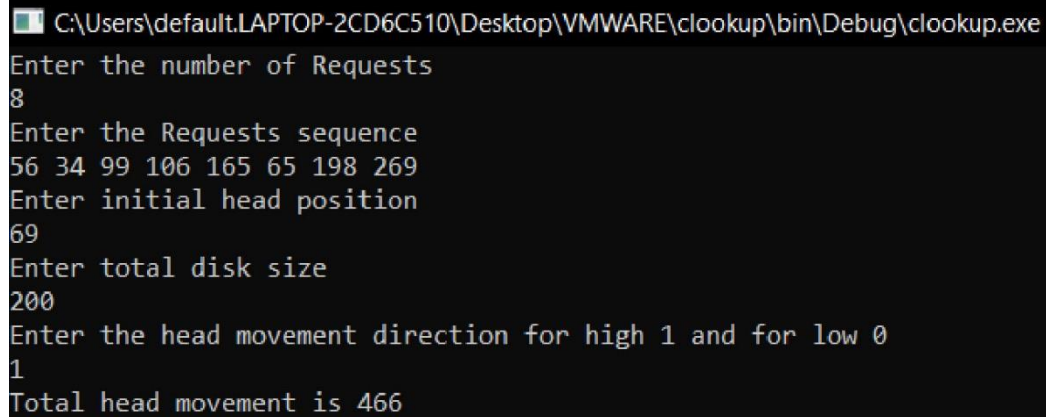
else{
    for(i=index-1;i>=0;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    for(i=n-1;i>=index;i--){
        TotalHeadMovement=TotalHeadMovement+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}

printf("Total head movement is %d",TotalHeadMovement);
return 0;
}

```

OUTPUT:



```

C:\Users\default.LAPTOP-2CD6C510\Desktop\VMWARE\clookup\bin\Debug\clookup.exe
Enter the number of Requests
8
Enter the Requests sequence
56 34 99 106 165 65 198 269
Enter initial head position
69
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 466

```