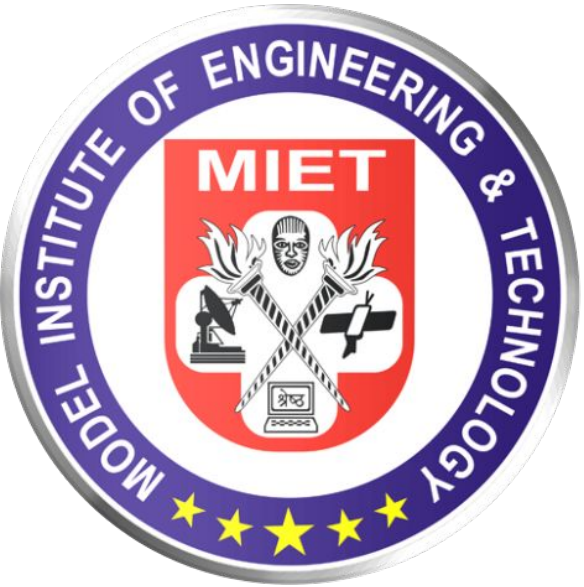


PRESENTED BY-TEAM NO. 8

JAPLEEN KAUR (2021A1R033)
GOKUL JAMWAL(2021A1R043)
REETIKESH BALI(2021A1R037)
DHRUV GUPTA(2021A1R041)

OPERATING SYSTEMS LAB
COM-312

TEACHER INCHARGE- Mr. Saurabh Sharma

PROBLEM STATEMENT 9

SIMULATE THE WORKING OF FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK DISC-SCHEDULING ALGORITHMS ON LINUX ENVIRONMENT.

PROBLEM DESCRIPTION

Disk Scheduling Algorithms are used to reduce the total seek time of any request. In operating systems, seek time is very important.

Since all device requests are linked in queues, the increased seek time causes the system to slow down.

–*Seek time* is the time for the disk to move the heads to the cylinder containing the desired sector.

Several algorithms exist to schedule the servicing of disk I/O requests they are as follows:

1. **First Come-First Serve (FCFS)**
2. **Shortest Seek Time First (SSTF)**
3. **Elevator (SCAN)**
4. **Circular SCAN (C-SCAN)**
5. **C-LOOK**
6. **LOOK**

DISC-SCHEDULING

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. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.

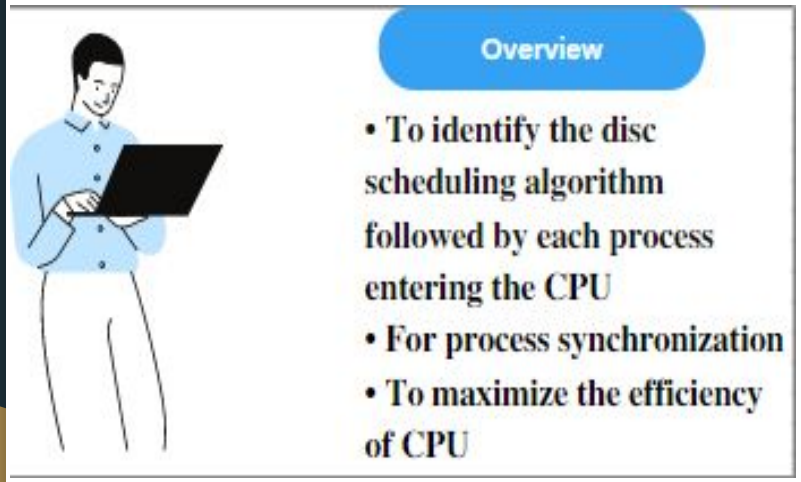
Two or more request may be far from each other so can result in greater disk arm movement.

Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

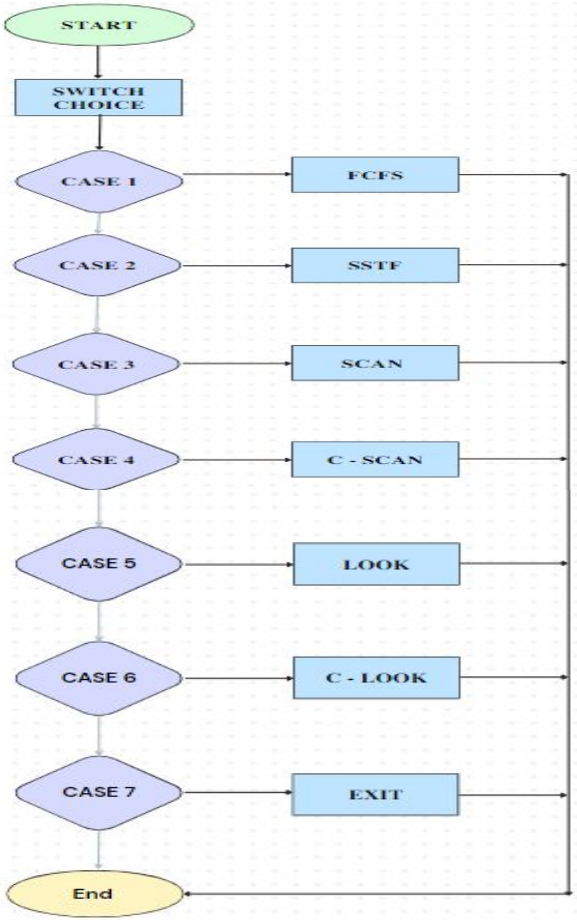
TYPES OF DISC-SCHEDULING ALGORITHMS

- FCFS (First Come First Serve)
- SSTF (Shortest Seek Time First)
- SCAN (Elevator)
- C-SCAN(Circular Scan)
- LOOK
- C-LOOK

FLOWCHART



Flowchart This visual map shows us. The working of FCFS,SSTF,SCAN,C-SCAN,LOOK,AND C-LOOK DISC -SCHEDULING ALOGORITHM


START
SWITCH CHOICE
CASE 1
FCFS
CASE 2
SSTF
CASE 3
SCAN
CASE 4
C - SCAN
CASE 5
LOOK
CASE 6
C - LOOK
CASE 7
EXIT
End

FCFS (First Come First Serve)

FCFS is the simplest disk scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

ALGORITHM -:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position
5. Go to step 2 until all tracks in request array have not been serviced.

Input:

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

Output:

Total number of seek operations = 510

Seek Sequence is

176

79

34

60

92

11

41

114

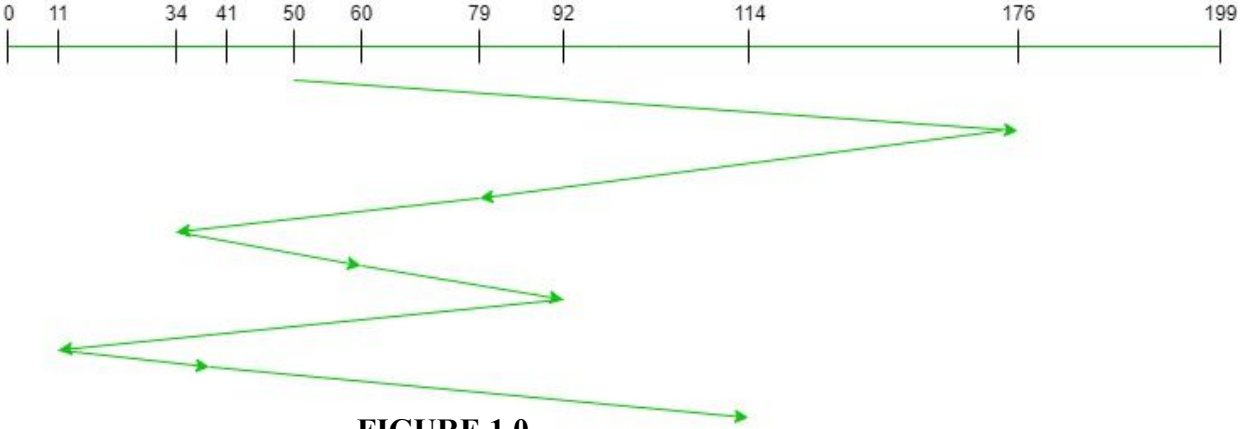


FIGURE 1.0

The following chart shows the sequence in which requested tracks are serviced using FCFS.

Therefore, the total seek count is calculated as:

= (176-50)+(176-79)+(79-34)+(60-34)+(92-60)+(92-11)+(41-11)+(114-41)

PSEUDO CODE FOR FCFS

1.

```
#Function for fcfs scheduling

  #def find_seek time(head, seek=0 max, diff;)
```

2.

```
# for calculating seek time

 queue[0]=head;

          for(j=0;j<=n-1;j++)

          {

                    diff=abs(queue[j+1]-queue[j]);

                    seek+=diff;
```

3.

Movement of head

(Disk head moves from %d to %d with seek %d\n", queue[j], queue[j+1], diff) }

SSTF(Shortest Seek Time First)

Basic idea is the tracks which are closer to current disk head position should be serviced first in order to minimize the seek operations.

ALGORITHM-:

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘head’ is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

PSEUDO CODE FOR SSTF

1.

```
#Function to find the seek time

  # def find_seek time (q_size, head, seek=0 , temp);
```

2.

```
  #calculating distance from head of elements in queue

 for(int i=0; i<q_size; i++){

    queue2[i] = abs(head-queue[i]);

  }
```

3.

```
# swapping elements in the queue

( if queue[i]>queue[j]){

            temp = queue2[i];

            queue2[i]=queue[j];

            queue2[j]=temp;


            temp=queue[i];

            queue[i]=queue[j];

            queue[j]=temp;

          }
```

4.

```
#Calculating seek time
for(int i=1; i<q_size; i++)
{
 seek =
 seek + abs (head-queue[i]);
    head = queue[i];

avg = seek/(float)q_size;
```

INPUT-

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

The following chart shows the sequence in which requested tracks are serviced using SSTF.

Therefore, total seek count is calculated as:

=(50-41)+(41-34)+(34-1

= 204

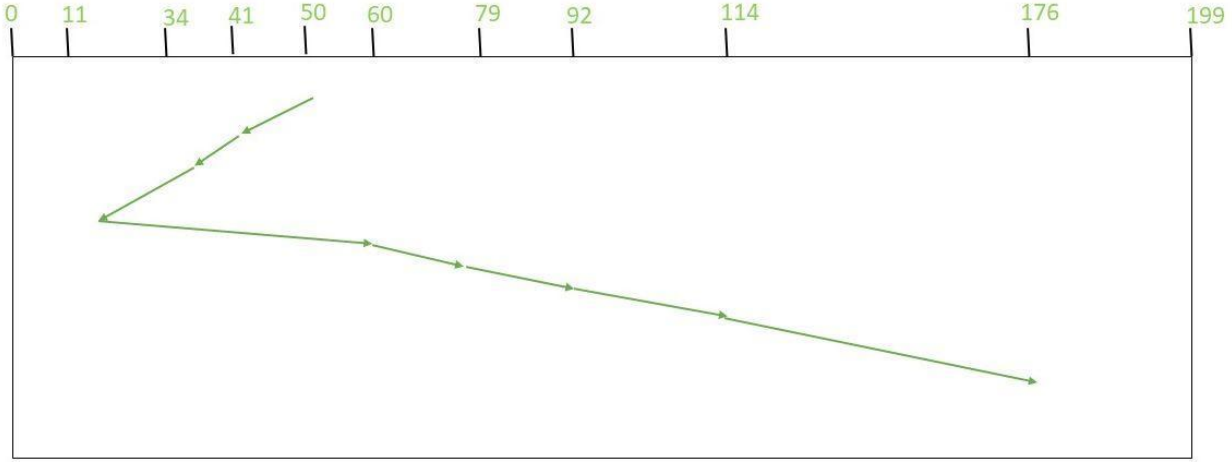


FIGURE 2.0

SCAN (ELEVATOR)

In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

ALGORITHM-:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

Total number of seek operations = 226

Seek Sequence is

41

34

11

0

60

79

92

114

176

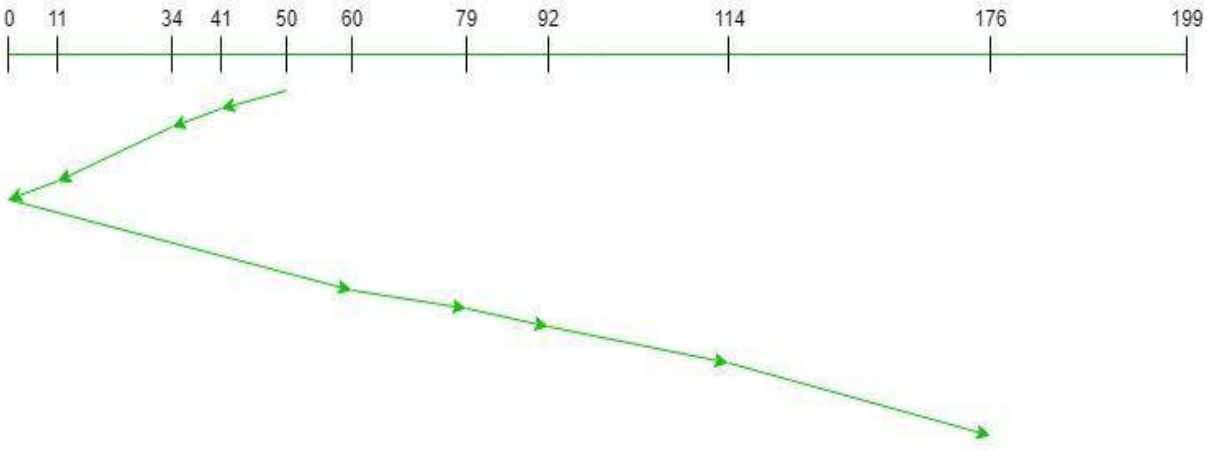


FIGURE 3.0

The following chart shows the sequence in which requested tracks are serviced using SCAN. Therefore, the total seek count is calculated as:

= (50-41)+(41-34)+(34-11)
+(11-0)+(60-0)+(79-60)
+(92-79)+(114-92)+(176-114)
= 226

PSEUDO CODE FOR SCAN DISC-SCHEDULING

1.

```
#To find seek time in scan disc scheduling
  #def  find_seek time( head, max, q_size, temp, sum;)
            store location of head(dloc;)
```

2.

```
    # sorting the array
      for(int i=0; i<q_size;i++){
   for(int j=i; j<q_size; j++){
     if(queue[i]>queue[j]){
       temp = queue[i];
       queue[i] = queue[j];
       queue[j] = temp;
     }
   }
 }
 max = queue[q_size-1];
```

3.

```
 # locate head in the queue
    for(int i=0; i<q_size; i++)
 {
  if(head == queue[i])
 {
   dloc = i;
   break;
  }
  }
```

4.

```
  #Calculating seek time
  if(abs(head-LOW) <= abs(head-HIGH)){

    for(int j=dloc; j>=0; j--){
     printf("%d --> ",queue[j]);
    }
    for(int j=dloc+1; j<q_size; j++){
     printf("%d --> ",queue[j]);
    }

    } else {

    for(int j=dloc+1; j<q_size; j++){
       printf("%d --> ",queue[j]);
    }
    for(int j=dloc; j>=0; j--){
       printf("%d --> ",queue[j]);
    }

 }
 sum  = head + max;
```

C-SCAN (CIRCULAR SCAN)

Circular SCAN (C-SCAN) scheduling algorithm is a modified version of SCAN disk scheduling algorithm that deals with the inefficiency of SCAN algorithm by servicing the requests more uniformly. Like SCAN (Elevator Algorithm) C-SCAN moves the head from one end servicing all the requests to the other end. However, as soon as the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip (see chart below) and starts servicing again once reaches the beginning. This is also known as the “Circular Elevator Algorithm” as it essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

ALGORITHM:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. The head services only in the right direction from 0 to size of the disk.
3. While moving in the left direction do not service any of the tracks.
4. When we reach at the beginning(left end) reverse the direction.
5. While moving in right direction it services all tracks one by one.
6. While moving in right direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 6 until we reach at right end of the disk.
10. If we reach at the right end of the disk reverse the direction and go to step 3 until all tracks n request array have not been serviced.

Input:
Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}
Initial head position = 50
Direction = right(We are moving from left to right)

Output:
Initial position of head: 50
Total number of seek operations = 389

Seek Sequence is
60
79
92
114
176
199
0
11
34
41

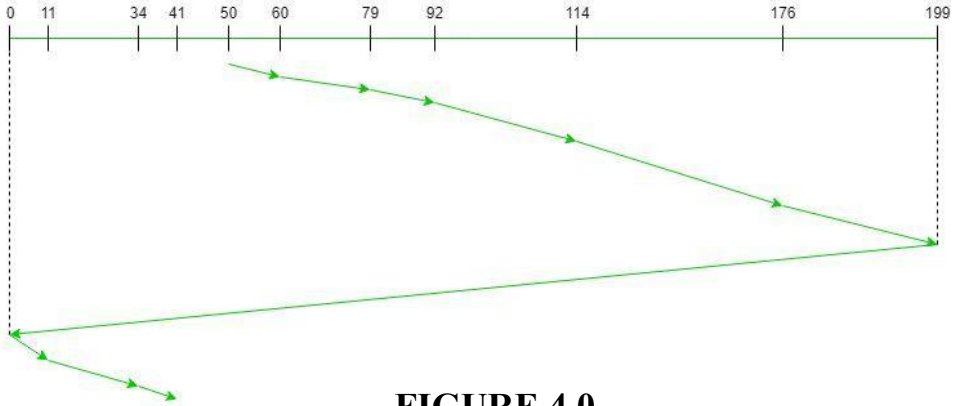


FIGURE 4.0

Therefore, the total seek count is calculated as:

= (60-50)+(79-60)+(92-79)
+(114-92)+(176-114)+(199-176)+(199-0)
+(11-0)+(34-11)+(41-34)
= 389

PSEUDO CODE FOR C-SCAN DISC SCHEDULING

```
C-SCAN(requests, start)
    requests = sort(requests)  # sort requests in ascending order
    current_position = start  # set current position to start
    for request in requests:
        if request >= current_position:  # check if request is ahead of current position
            service_request(request)  # service request
            current_position = request  # update current position
    current_position = 0  # jump to beginning of disk
    for request in requests:
        if request < current_position:  # check if request is behind current position
            service_request(request)  # service request
            current_position = request
```

LOOK DISC SCHEDULING

LOOK is the advanced version of SCAN (elevator) disk scheduling algorithm which gives slightly better seek time than any other algorithm in the hierarchy (FCFS-> SRTF-> SCAN-> C-SCAN-> LOOK). TheLOOK algorithm services request similarly as SCAN algorithm meanwhile it also “looks” ahead as if there are more tracks that are needed to be serviced in the same direction. If there are no pending requests in the moving direction the head reverses the direction and start servicing requests in the opposite direction.The main reason behind the better performance of LOOK algorithm in comparison to SCAN is because in this algorithm the head is not allowed to move till the end of the disk.

ALGORITHM-:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. The initial direction in which head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction head is moving.
4. The head continues to move in the same direction until all the request in this direction are not finished.
5. While moving in this direction calculate the absolute distance of the track from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach at last request in this direction.
9. If we reach where no requests are needed to be serviced in this direction reverse the direction and go to step 3 until all tracks in request array have not been serviced.

Input:

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

Direction = right (We are moving from left to right)

Output:

Initial position of head: 50

Total number of seek operations = 291

Seek Sequence is

60

79

92

114

176

41

34

11

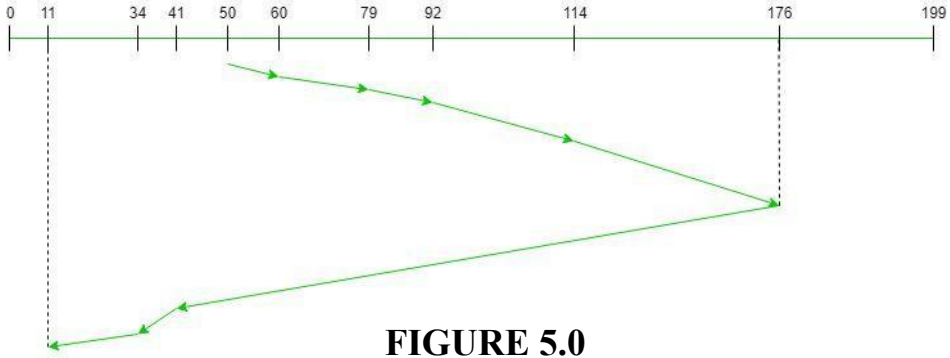


FIGURE 5.0

Therefore, the total seek count is calculated as:

= (60-50)+(79-60)+(92-79)
+(114-92)+(176-114)
+(176-41)+(41-34)+(34-11

PSEUDO CODE FOR LOOK DISC SCHEDULING

```
LOOK(queue, head)
  // queue: list of request blocks to be scheduled
  // head: current position of the disk head

  // Find the direction of the head movement
  if (head is moving toward the end of the queue)
    direction = 1
  else
    direction = -1

  // Initialize the closest block to be the current head position
  closest_block = head

  // Iterate through the queue in the appropriate direction
  for each request block in queue in the direction of head movement:
    if (request block is closer to the head than the closest block)
      closest_block = request block
    else
      // We have passed the closest block, so we can stop searching
      break

  // Return the closest block
  return closest_block
```

C-LOOK(CIRCULAR LOOK)

C-LOOK is an enhanced version of both SCAN as well as LOOK disk scheduling algorithms. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm. We know that C-SCAN is used to avoid starvation and services all the requests more uniformly, the same goes for C-LOOK. In this algorithm, the head services requests only in one direction(either left or right) until all the requests in this direction are not serviced and then jumps back to the farthest request on the other direction and service the remaining requests which gives a better uniform servicing as well as avoids wasting seek time for going till the end of the disk.

ALGORITHM-:

1. Let Request array represents an array storing indexes of the tracks that have been requested in ascending order of their time of arrival and head is the position of the disk head.
2. The initial direction in which the head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction it is moving.
4. The head continues to move in the same direction until all the requests in this direction have been serviced.
5. While moving in this direction, calculate the absolute distance of the tracks from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach the last request in this direction.
9. If we reach the last request in the current direction then reverse the direction and move the head in this direction until we reach the last request that is needed to be serviced in this direction without servicing the intermediate requests.
10. Reverse the direction and go to step 3 until all the requests have not been serviced.

Input:

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

Direction = right (Moving from left to right)

Output:

Initial position of head: 50

Total number of seek operations = 156

Seek Sequence is

60

79

92

114

176

11

34

41

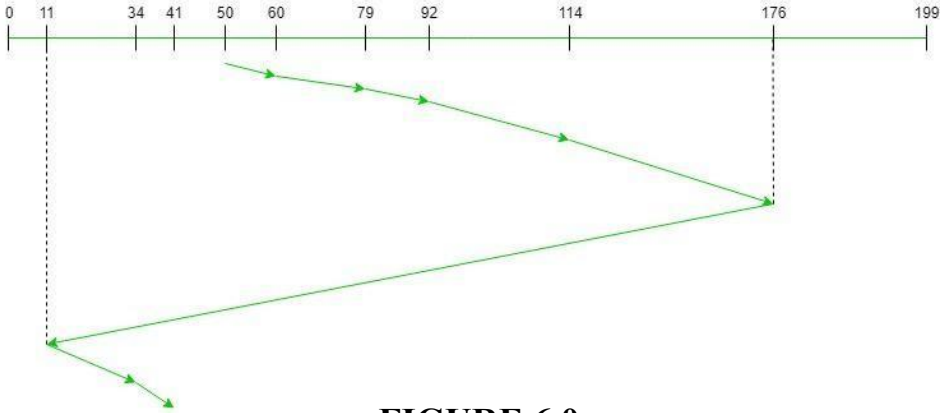


FIGURE 6.0

Therefore, the total seek count = (60 – 50) + (79 – 60) + (92 – 79) + (114 – 92) + (176 – 114) + (176 – 11) + (34 – 11) + (41 – 34) = 321

PSEUDO CODE FOR C-LOOK DISC SCHEDULING

1.

```
#Function to find seek time
# def  find_seek time( head,
item, dst;0
        dst[i]=(head-item[i]);
```

2.

```
#Selection Sort
 for(i=0;i<n-1;i++)
   {
      for(j=i+1;j<n;j++)
      {
         if(dst[j]>dst[i])
         {
            int temp=dst[j];
            dst[j]=dst[i];
            dst[i]=temp;

            temp=item[i];
            item[i]=item[j];
            item[j]=temp}}}  for(i=0;i<n;i++)
                              { if(item[i]>=head)
                              { j=i;
                               break; }}
```

3.

```
# calculating seek time

cyl+= abs(head-item[i]);
        head=item[i];
```

LINK OF THE CODES USED-

- https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/
- https://www.educative.io/answers/what-is-the-sstf-disk-scheduling-algorithm
- https://www.thecompletecodez.com/2019/08/c-program-for-scan-disk-scheduling.html
- https://codewithsudeep.com/sudeep/c-program/program-to-simulate-c-scan-disk-scheduling-algorithm/
- https://www.nanogalaxy.org/2021/05/c-program-to-simulate-look-disk.html
- https://github.com/onyxe/Disk-Scheduling-Algorithms/blob/master/cLOOK.c
- https://drive.google.com/file/d/1m3wvHj2eS6yR6A6_RiVdpBgofYvqfAvI/view

THANK YOU!!!