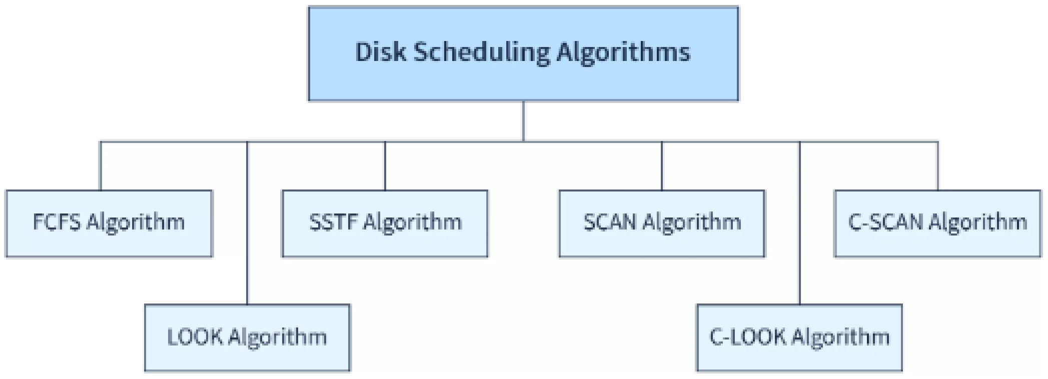
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Operating System Laboratory
Course Code:	DJ19CEL403
Experiment No.:	10

AIM: **DISK SCHEDULING ALGORITHM (FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK)**

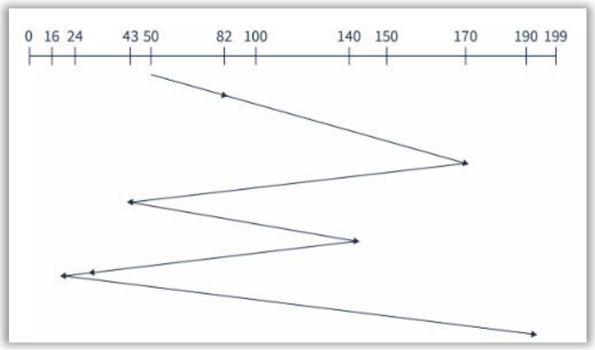
THEORY:

- 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 requests 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.



FCFS

- It stands for 'first-come-first-serve'.
- As the name suggests, the request which comes first will be processed first and so on.
- The requests coming to the disk are arranged in a proper sequence as they arrive.
- Since every request is processed in this algorithm, so there is no chance of 'starvation'.
- Example:
  Suppose a disk having 200 tracks (0-199).
  The request sequence(82,170,43,140,24,16,190) of disk are shown as in the given figure and the head start is at request 50.



Explanation: In the above image, we can see the head starts at position 50 and moves to request 82. After serving them the disk arm moves towards the second request that is 170 and then to the request 43 and so on. In this algorithm, the disk arm will serve the requests in arriving order. In this way, all the requests are served in arriving order until the process executes.

"Seek time" will be calculated by adding the head movement differences of all the requests:
Seek time = "(82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16) = 642

- Advantages:
    1. Implementation is easy.
    2. No chance of starvation.
- Disadvantages:
    1. 'Seek time' increases.
    2. Not so efficient

CODE:

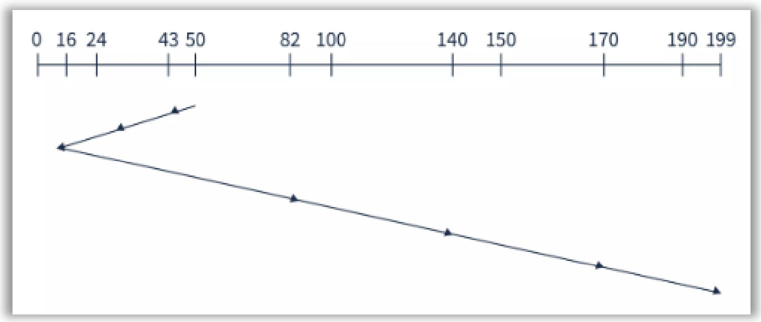
```
package Exp10;
class FCFS_Disk {
    static void FCFS(int arr[], int head) {
        int size = arr.length, seek_count = 0;
        int distance, cur_track;
        for (int i = 0; i < size; i++) {
            cur_track = arr[i];
            distance = Math.abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        System.out.println("Total number of " + "seek operations = " +
seek_count);
        System.out.println("Seek Sequence is");
        for (int i = 0; i < size; i++) {
            System.out.println(arr[i]);
        }
    }
    public static void main(String[] args) {
        int arr[] = { 82, 170, 43, 140, 24, 16, 190 };
        int head = 50;
        FCFS(arr, head);
    }
}
```

OUTPUT:

```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.FCFS_Disk'
Total number of seek operations = 642
Seek Sequence is
82
170
43
140
24
16
190
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code> 
```

SSTF

- It stands for 'Shortest seek time first'. As the name suggests, it searches for the request having the least 'seek time' and executes them first.
- This algorithm has less 'seek time' as compared to FCFS Algorithm.
- Example:
  Suppose a disk having 200 tracks (0-199). The request sequence(82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: The disk arm searches for the request which will have the least difference in head movement. So, the least difference is (50-43). Here the difference is not about the shortest value, but it is about the shortest time the head will take to reach the nearest next request. So, after 43, the head will be nearest to 24, and from here the head will be nearest to the request 16, After 16, the nearest request is 82, so the disk arm will move to serve request 82 and so on.
Hence, Calculation of Seek Time = (50-43) + (43-24) + (24-16) + (82-16) + (140-82) + (170-140) + (190-170) = 208

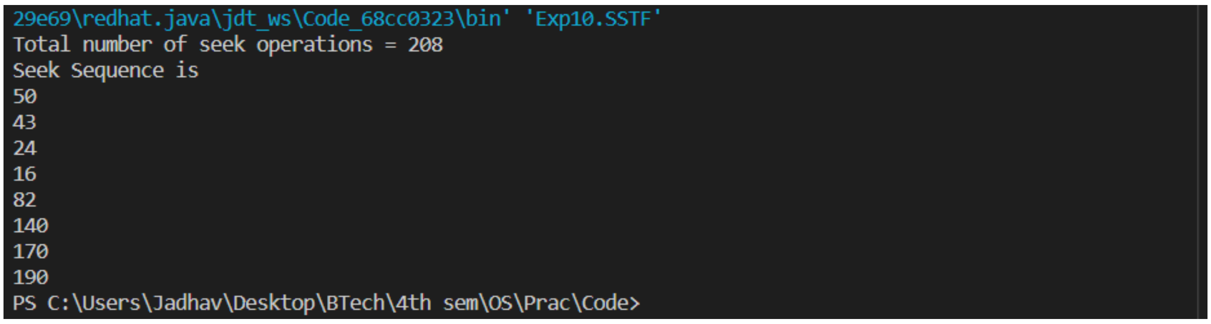
- Advantages:
  In this algorithm, disk response time is less.
  More efficient than FCFS.
- Disadvantages:
  Less speed of algorithm execution.
  Starvation can be seen.

CODE:

```
package Exp10;
class node {
    int distance = 0;
    boolean accessed = false;
}
```

```java
public class SSTF {
    public static void calculateDifference(int queue[], int head, node
diff[]){
        for (int i = 0; i < diff.length; i++)
            diff[i].distance = Math.abs(queue[i] - head);
    }
    public static int findMin(node diff[])
    {
        int index = -1, minimum = Integer.MAX_VALUE;
        for (int i = 0; i < diff.length; i++) {
            if (!diff[i].accessed && minimum > diff[i].distance) {
                minimum = diff[i].distance;
                index = i;
            }
        }
        return index;
    }
    public static void shortestSeekTimeFirst(int request[], int head){
        if (request.length == 0)
            return;
        node diff[] = new node[request.length];
        for (int i = 0; i < diff.length; i++)
            diff[i] = new node();
        int seek_count = 0;
        int[] seek_sequence = new int[request.length + 1];
        for (int i = 0; i < request.length; i++) {
            seek_sequence[i] = head;
            calculateDifference(request, head, diff);
            int index = findMin(diff);
            diff[index].accessed = true;
            seek_count += diff[index].distance;
            head = request[index];
        }
        seek_sequence[seek_sequence.length - 1] = head;
        System.out.println("Total number of seek operations = " + seek_count);
        System.out.println("Seek Sequence is");
        for (int i = 0; i < seek_sequence.length; i++)
            System.out.println(seek_sequence[i]);
    }
    public static void main(String[] args)  {
        int arr[] = { 82 , 170 , 43 , 140 , 24 , 16 , 190 };
        shortestSeekTimeFirst(arr, 50);
    }
}
```

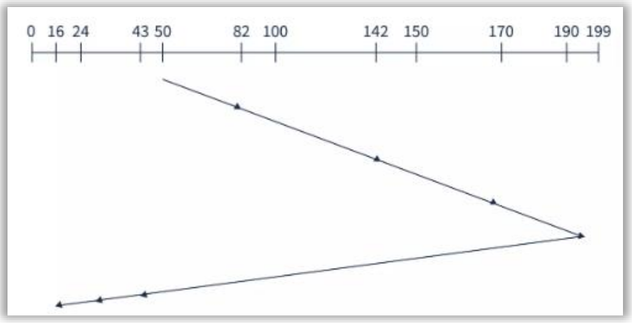
OUTPUT:



```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.SSTF'
Total number of seek operations = 208
Seek Sequence is
50
43
24
16
82
140
170
190
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

<u>SCAN</u>

- In this algorithm, the head starts to scan all the requests in a direction and reaches the end of the disk.
- After that, it reverses its direction and starts to scan again the requests in its path and serves them.
- Due to this feature, this algorithm is also known as the "Elevator Algorithm".
- Example:
  Suppose a disk having 200 tracks (0-199).
  The request sequence(82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.
  The 'disk arm' will first move to the larger values.



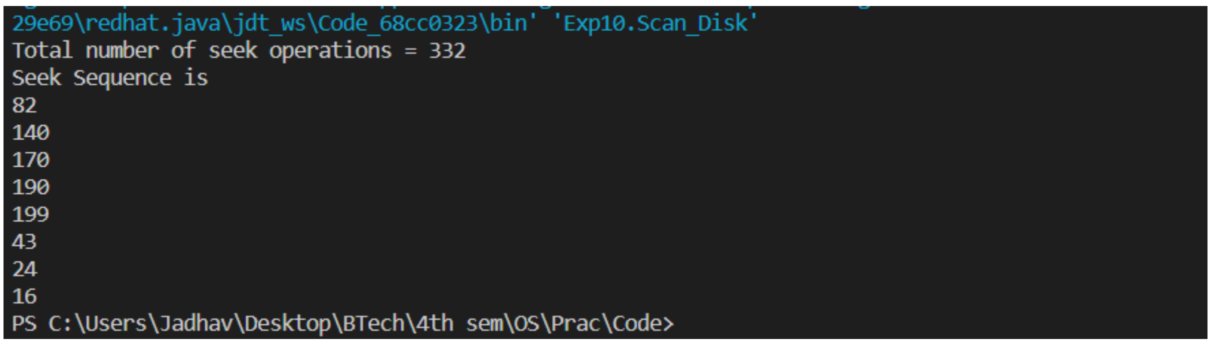
- Explanation: In the above image, we can see that the disk arm starts from position 50 and goes in a single direction until it reaches the end of the disk i.e.- request position 199. After that, it reverses and starts servicing in the opposite direction until reached the other end of the disk. This process keeps going on until the process is executed. Hence, Calculation of 'Seek Time' will be like: (199-50) + (199-16) =332

- Advantages:
  Implementation is easy.
  Requests do not have to wait in queue.
- Disadvantage:
  The head keeps going on to the end even if there are no requests in that direction.

CODE:

```
package Exp10;
import java.util.*;

class Scan_Disk {
    static int disk_size = 200;
    static void SCAN(int arr[], int head, String direction) {

        int size = arr.length;
        int seek_count = 0;
        int distance, cur_track;

        Vector<Integer> left = new Vector<Integer>(), right = new
Vector<Integer>();
        Vector<Integer> seek_sequence = new Vector<Integer>();

        if (direction == "left")
            left.add(0);
        else if (direction == "right")
            right.add(disk_size - 1);
        for (int i = 0; i < size; i++) {
            if (arr[i] < head)
                left.add(arr[i]);
            if (arr[i] > head)
                right.add(arr[i]);
        }

        Collections.sort(left);
        Collections.sort(right);
        int run = 2;

        while (run-- > 0) {
            if (direction == "left") {
                for (int i = left.size() - 1; i >= 0; i--) {
                    cur_track = left.get(i);
                    seek_sequence.add(cur_track);
                    distance = Math.abs(cur_track - head);
                    seek_count += distance;
                    head = cur_track;
                }
                direction = "right";
            } else if (direction == "right") {
                for (int i = 0; i < right.size(); i++) {
                    cur_track = right.get(i);
                    seek_sequence.add(cur_track);
                    distance = Math.abs(cur_track - head);
                    seek_count += distance;
                    head = cur_track;
                }
                direction = "left";
            }
        }

        System.out.print("Total number of seek operations = " + seek_count +
"\n");
        System.out.print("Seek Sequence is" + "\n");
        for (int i = 0; i < seek_sequence.size(); i++) {
            System.out.print(seek_sequence.get(i) + "\n");
        }
    }

    public static void main(String[] args) {
        int arr[] = { 82, 170, 43, 140, 24, 16, 190 };
        int head = 50;
        String direction = "right";
        SCAN(arr, head, direction);
    }
}
```

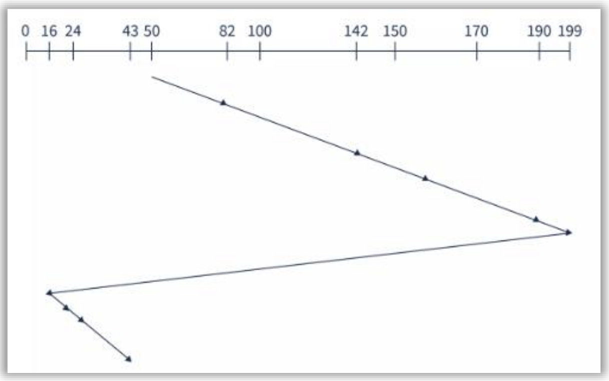
OUTPUT:



```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.Scan_Disk'
Total number of seek operations = 332
Seek Sequence is
82
140
170
190
199
43
24
16
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

C-SCAN

- It stands for "Circular-Scan".
- This algorithm is almost the same as the Scan disk algorithm but one thing that makes it different is that 'after reaching the one end and reversing the head direction, it starts to come back.
- The disk arm moves toward the end of the disk and serves the requests coming into its path.
- After reaching the end of the disk it reverses its direction and again starts to move to the other end of the disk but while going back it does not serve any requests.
- Example:
  Suppose a disk having 200 tracks (0-199).
  The request sequence(82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: In the above figure, the disk arm starts from position 50 and reached the end(199), and serves all the requests in the path. Then it reverses the direction and moves to the other end of the disk i.e.- 0 without serving any task in the path. After reaching 0, it will again go move towards the largest remaining value which is 43. So, the head will start from 0 and moves to request 43 serving all the requests coming in the path. And this process keeps going.

Hence, Seek Time will be =(199-50) + (199-0) + (43-0) =391

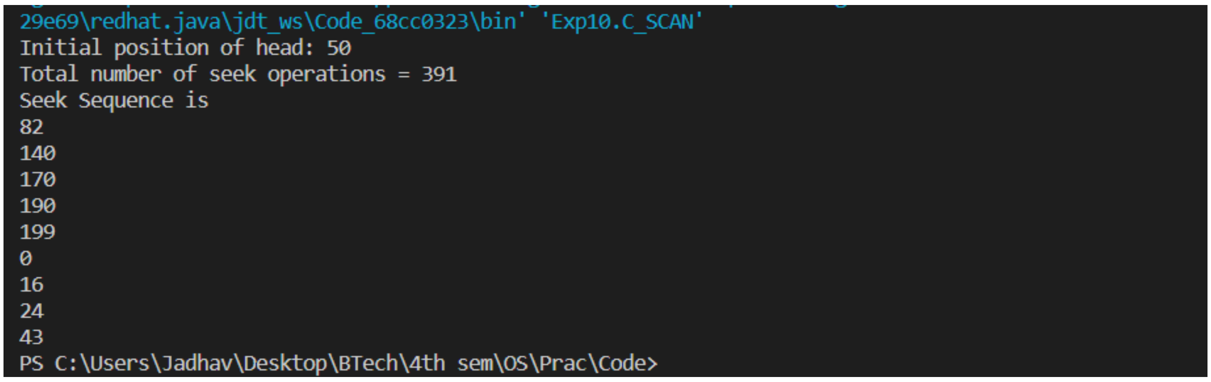
- Advantages:
  - The waiting time is uniformly distributed among the requests.
  - Response time is good in it.
- Disadvantages:
  - The time taken by the disk arm to locate a spot is increased here.
  - The head keeps going to the end of the disk.

CODE:

```
package Exp10;

import java.util.*;

class C_SCAN {

    static int disk_size = 200;

    public static void CSCAN(int arr[], int head) {

        int size = arr.length;
        int seek_count = 0;
        int distance, cur_track;

        Vector<Integer> left = new Vector<Integer>();
        Vector<Integer> right = new Vector<Integer>();
        Vector<Integer> seek_sequence = new Vector<Integer>();

        left.add(0);
        right.add(disk_size - 1);

        for (int i = 0; i < size; i++) {
            if (arr[i] < head)
                left.add(arr[i]);
            if (arr[i] > head)
```

```
                right.add(arr[i]);
        }

        Collections.sort(left);
        Collections.sort(right);

        for (int i = 0; i < right.size(); i++) {
            cur_track = right.get(i);
            seek_sequence.add(cur_track);
            distance = Math.abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }

        head = 0;
        seek_count += (disk_size - 1);

        for (int i = 0; i < left.size(); i++) {
            cur_track = left.get(i);
            seek_sequence.add(cur_track);
            distance = Math.abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }

        System.out.println("Total number of seek " + "operations = " +
seek_count);
        System.out.println("Seek Sequence is");

        for (int i = 0; i < seek_sequence.size(); i++) {
            System.out.println(seek_sequence.get(i));
        }
    }

    public static void main(String[] args) throws Exception {
        int arr[] = { 82, 170, 43, 140, 24, 16, 190 };
        int head = 50;
        System.out.println("Initial position of head: " + head);
        CSCAN(arr, head);
    }
}
```

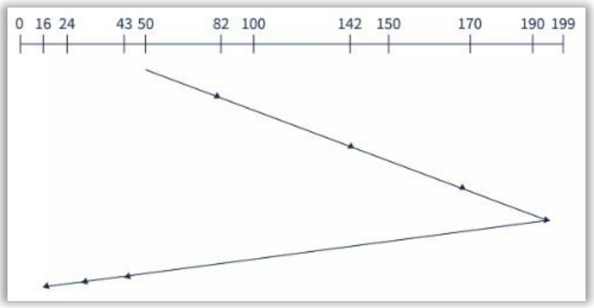
OUTPUT:



```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.C_SCAN'
Initial position of head: 50
Total number of seek operations = 391
Seek Sequence is
82
140
170
190
199
0
16
24
43
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

<ins>**LOOK**</ins>

- In this algorithm, the disk arm moves to the 'last request' present and services them. After reaching the last requests, it reverses its direction and again comes back to the starting point. It does not go to the end of the disk, in spite, it goes to the end of requests.
- Example a disk having 200 tracks (0-199). The request sequence(82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



Explanation: The disk arm is starting from 50 and starts to serve requests in one direction only but in spite of going to the end of the disk, it goes to the end of requests i.e.-190. Then comes back to the last request of other ends of the disk and serves them. And again starts from here and serves till the last request of the first side.
Hence, Seek time =(190-50) + (190-16) =314

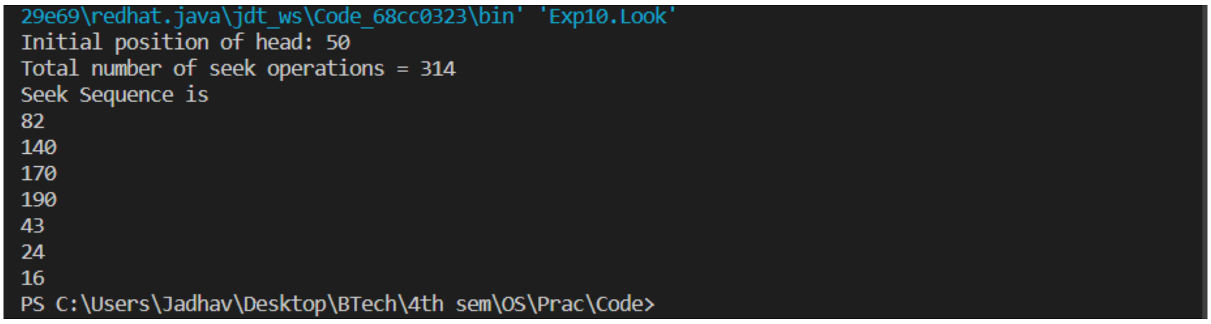
- Advantages:
  Starvation does not occur.
  Since the head does not go to the end of the disk, the time is not wasted here.
- Disadvantage:
  The arm has to be conscious to find the last request.

CODE:

```
package Exp10;

import java.util.*;

class Look {
    static int disk_size = 200;
    public static void LOOK(int arr[], int head, String direction) {
        int size = arr.length;
        int seek_count = 0;
        int distance, cur_track;
```

```java
        Vector<Integer> left = new Vector<Integer>();
        Vector<Integer> right = new Vector<Integer>();
        Vector<Integer> seek_sequence = new Vector<Integer>();
        for (int i = 0; i < size; i++) {
            if (arr[i] < head)
                left.add(arr[i]);
            if (arr[i] > head)
                right.add(arr[i]);
        }
        Collections.sort(left);
        Collections.sort(right);
        int run = 2;
        while (run-- > 0) {
            if (direction == "left") {
                for (int i = left.size() - 1; i >= 0; i--) {
                    cur_track = left.get(i);
                    seek_sequence.add(cur_track);
                    distance = Math.abs(cur_track - head);
                    seek_count += distance;
                    head = cur_track;
                }
                direction = "right";
            } else if (direction == "right") {
                for (int i = 0; i < right.size(); i++) {
                    cur_track = right.get(i);
                    seek_sequence.add(cur_track);
                    distance = Math.abs(cur_track - head);
                    seek_count += distance;
                    head = cur_track;
                }
                direction = "left";
            }
        }
        System.out.println("Total number of seek " + "operations = " +
seek_count);
        System.out.println("Seek Sequence is");
        for (int i = 0; i < seek_sequence.size(); i++) {
            System.out.println(seek_sequence.get(i));
        }
    }

    public static void main(String[] args) throws Exception {
        int arr[] = { 82 , 170 , 43 , 140 , 24 , 16 , 190 };
        int head = 50;
        String direction = "right";
        System.out.println("Initial position of head: " +head);
        LOOK(arr, head, direction);
    }
}
```

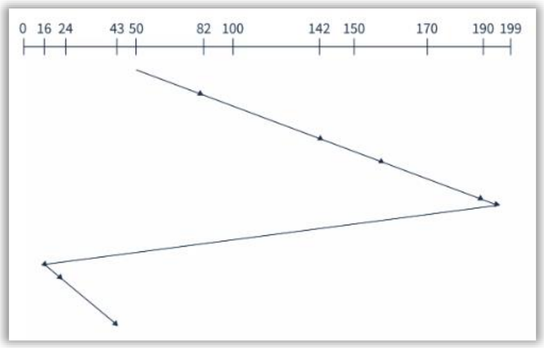
OUTPUT:



```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.Look'
Initial position of head: 50
Total number of seek operations = 314
Seek Sequence is
82
140
170
190
43
24
16
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

C-LOOK

- The C-Look algorithm is almost the same as the Look algorithm.
- The only difference is that after reaching the end requests, it reverses the direction of the head and starts moving to the initial position.
- But in moving back, it does not serve any requests.
- Example:
  Suppose a disk having 200 tracks (0-199).
  The request sequence(82,170,43,140,24,16,190) are shown in the given figure and the head position is at 50.



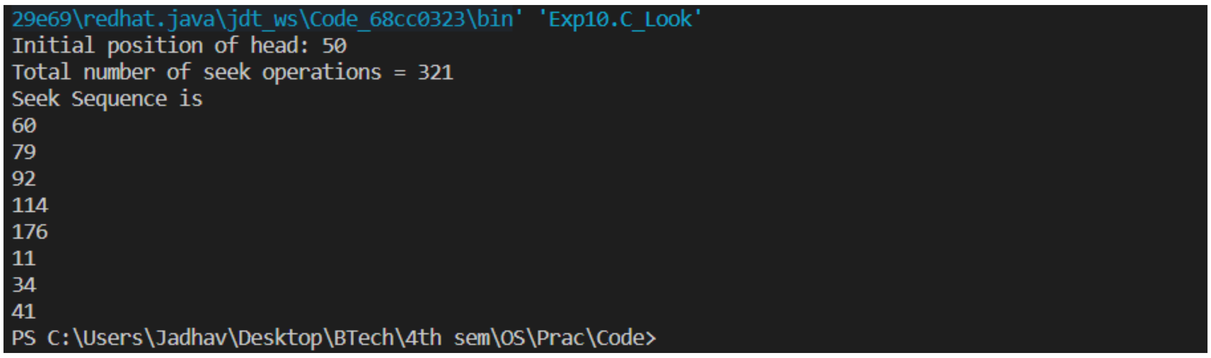
Explanation: The disk arm is starting from 50 and starts to serve requests in one direction only but in spite of going to the end of the disk, it goes to the end of requests i.e.-190. Then comes back to the last request of other ends of a disk without serving them. And again starts from the other end of the disk and serves requests of its path. Hence, Seek Time = (190-50) + (190-16) + (43-16) =341

- Advantages:
  The waiting time is decreased.
  If there are no requests till the end, it reverses the head direction immediately.
  Starvation does not occur.
  The time taken by the disk arm to find the desired spot is less.
- Disadvantage:
  The arm has to be conscious about finding the last request.

CODE:

```
package Exp10;

import java.util.*;

class C_Look {
    static int disk_size = 200;

    public static void CLOOK(int arr[], int head) {
        int size = arr.length;
        int seek_count = 0;
        int distance, cur_track;
        Vector<Integer> left = new Vector<Integer>();
        Vector<Integer> right = new Vector<Integer>();
        Vector<Integer> seek_sequence = new Vector<Integer>();
        for (int i = 0; i < size; i++) {
            if (arr[i] < head)
                left.add(arr[i]);
            if (arr[i] > head)
                right.add(arr[i]);
        }

        Collections.sort(left);
        Collections.sort(right);
        for (int i = 0; i < right.size(); i++) {
            cur_track = right.get(i);
            seek_sequence.add(cur_track);
            distance = Math.abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        seek_count += Math.abs(head - left.get(0));
        head = left.get(0);
        for (int i = 0; i < left.size(); i++) {
            cur_track = left.get(i);
            seek_sequence.add(cur_track);
            distance = Math.abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
        }
        System.out.println("Total number of seek " + "operations = " +
seek_count);
        System.out.println("Seek Sequence is");
        for (int i = 0; i < seek_sequence.size(); i++) {
            System.out.println(seek_sequence.get(i));
        }
    }

    public static void main(String[] args) {
        int arr[] = { 176, 79, 34, 60, 92, 11, 41, 114 };
        int head = 50;
        System.out.println("Initial position of head: " + head);
        CLOOK(arr, head);
    }
}
```

OUTPUT:



```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'Exp10.C_Look'
Initial position of head: 50
Total number of seek operations = 321
Seek Sequence is
60
79
92
114
176
11
34
41
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

CONCLUSION:

- The Disk Scheduling Algorithm in OS is used to manage input and output requests to the disk.
- Disk Scheduling is important as multiple requests are coming to disk simultaneously and it is also known as the "Request Scheduling Algorithm"
- Various types of scheduling algorithms are:
    - FCFS (First come first serve) algorithm
    - SSTF (Shortest seek time first) algorithm
    - Scan Disk Algorithm
    - C-Scan (Circular scan) algorithm
    - Look algorithm
    - C-Look (Circular look) algorithm
- An algorithm in which starvation is minimum is considered an efficient algorithm.