LOVELY PROFESSIONAL UNIVERSITY

Transforming Education Transforming India

ACADAMIC TASK – 2

CSE 316

(OPERATING SYSTEM)

Submitted By:

Ashna Jain

Reg No. 12304693

Roll no. 29

Section: K23GN

Submitted To:

Dr. Parvinder Singh

Introduction to Operating System

An Operating System (OS) is system software that acts as an interface between the computer hardware and the user. It manages hardware resources, provides essential services for application software, and ensures smooth execution of programs.

Purpose of an Operating System

The main objectives of an operating system are:

- **Resource Management:** Allocates CPU time, memory, storage, and peripheral devices efficiently.
- **Process Management:** Controls execution of multiple processes and provides multitasking capabilities.
- **File Management:** Organizes, stores, retrieves, and secures data in file systems.
- **Security and Access Control:** Ensures data protection through user authentication and permissions.
- **User Interface:** Provides command-line and graphical interfaces for interaction.

Project Overview:

Introduction to Advanced Disk Scheduling:

The Advanced Disk Scheduler project aims to create an intelligent system that optimizes disk I/O operations in computer systems. The project will develop a scheduler that improves upon traditional disk scheduling algorithms (like FCFS, SSTF, SCAN) by incorporating machine learning to predict access patterns and adapt dynamically to workloads. Advanced disk scheduling techniques are designed to improve efficiency, minimize delays, and enhance system responsiveness.

Need for Advanced Disk Scheduling:

With increasing data storage demands and real-time processing requirements, traditional scheduling methods often lead to higher seek times and inefficient disk utilization. Advanced disk scheduling techniques aim to:

- Reduce the total head movement to minimize seek time.
- Improve overall system throughput and response time.
- Handle priority-based requests efficiently.
- Adapt to dynamic workloads in modern storage systems.

Goals:

- Create a more efficient disk scheduling mechanism that reduces seek time and rotational latency
- Minimize overall I/O wait times in various workload scenarios
- Provide analytical tools to visualize and understand disk access patterns
- Demonstrate measurable performance improvements over standard schedulers

Expected Outcomes:

- A functional disk scheduler that can be deployed in real systems
- Performance metrics showing improvements in throughput and latency
- Visualization tools for analyzing disk access patterns
- Documentation and research findings on scheduling optimization

Scope:

- Development of core scheduling algorithm and logic
- Implementation of ML-based prediction capabilities
- Creation of visualization and monitoring tools
- Performance testing and comparison with existing schedulers
- User interface for configuration and monitoring

Module-Wise Breakdown:

1. Data Structures

- **DiskRequest Struct**:
  - Represents a disk request with comprehensive attributes
  - Includes request_id, track_number, arrival_time, priority, deadline flag, and deadline
- **PredictionResult Struct**:
  - Simulates a machine learning prediction
  - Contains confidence level and predicted track number

2. Core Scheduling Algorithms

First Come First Serve:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics of FCFS:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

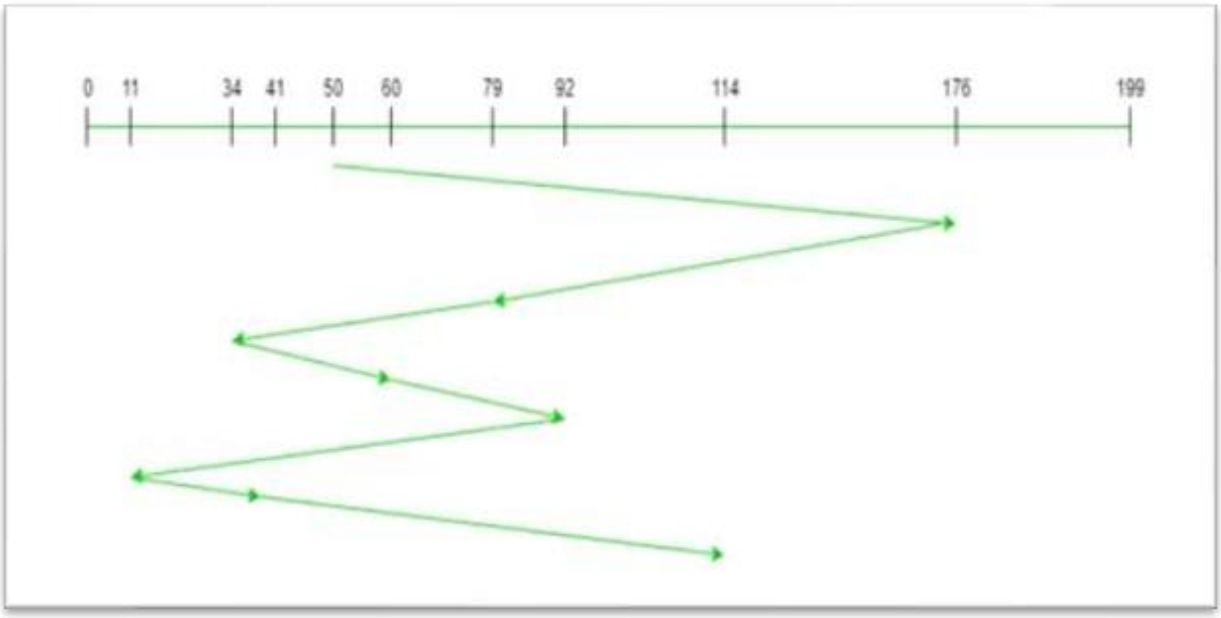
Advantages of FCFS:

- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

- FCFS suffers from Convoy effect.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

```
Input:
Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}
Initial head position = 50
```



Shortest Seek Time First (SSTF) :

The basic idea is the tracks that are closer to the current disk head position should be serviced first in order to minimize the seek operations is basically known as **Shortest Seek Time First (SSTF)**.

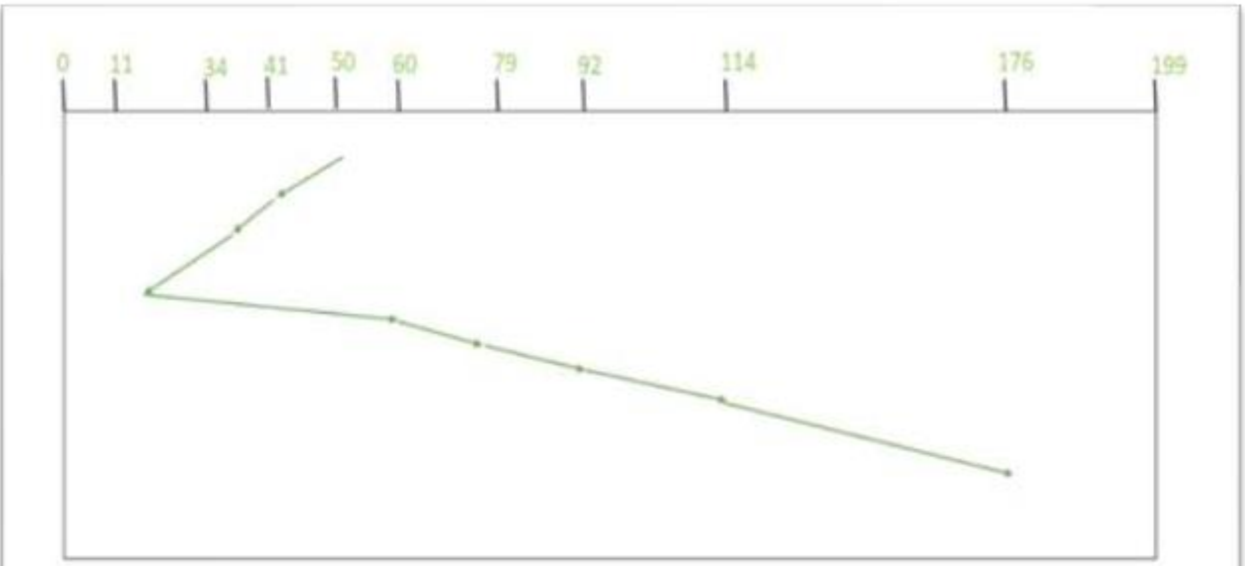
Advantages of Shortest Seek Time First (SSTF):

- Better performance than the FCFS
- It provides better throughput.
- This algorithm is used in Batch Processing systems where throughput is more important.
- It has a less average response and waiting time.

Disadvantages of Shortest Seek Time First (SSTF):

- Starvation is possible for some requests as it favours easy-to-reach requests and ignores the far-away processes.
- There is a lack of predictability because of the high variance of response time.
- Switching direction slows things down.

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}
Initial head position = 50



SSTF (Shortest Seek Time First)

SCAN Algorithm:

In the SCAN Disk Scheduling Algorithm, the head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reaching the other end. Then the direction of the head is reversed and the process continues as the head continuously scans back and forth to access the disk.

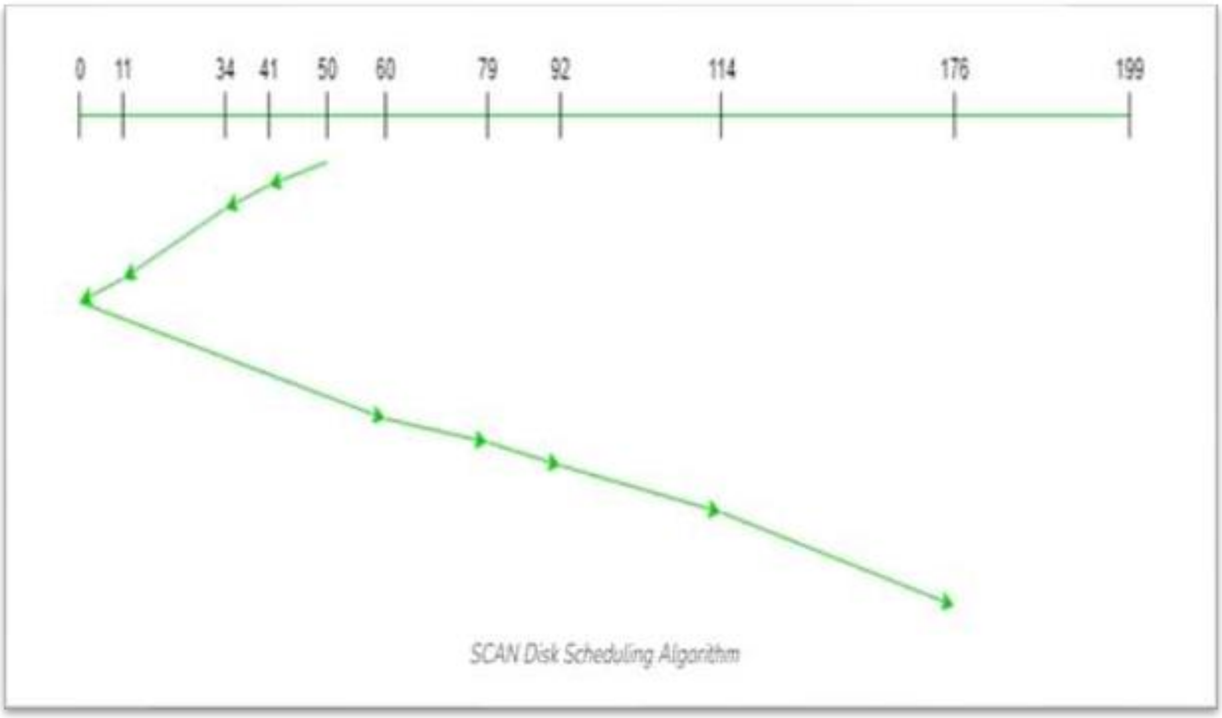
Advantages of SCAN (Elevator) Algorithm:

- This algorithm is simple and easy to understand.
- SCAN algorithm has no starvation
- This algorithm is better than the FCFS

Disadvantages of the SCAN (Elevator) Algorithm:

- More complex algorithm to implement.
- This algorithm is not fair because it causes a long waiting time for the cylinders just visited by the head.
- It causes the head to move till the end of the disk in this way the requests arriving ahead of the arm position would get immediate service but some other requests that arrive behind the arm position will have to wait for the request to complete.

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



SCAN Disk Scheduling Algorithm

C-SCAN (Circular SCAN) Algorithm:

The **Circular SCAN (C-SCAN) Scheduling Algorithm** is a modified version of the SCAN that deals with the inefficiency of the SCAN algorithm by servicing the requests more uniformly. Like SCAN , **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 and starts servicing again once reaches the beginning.

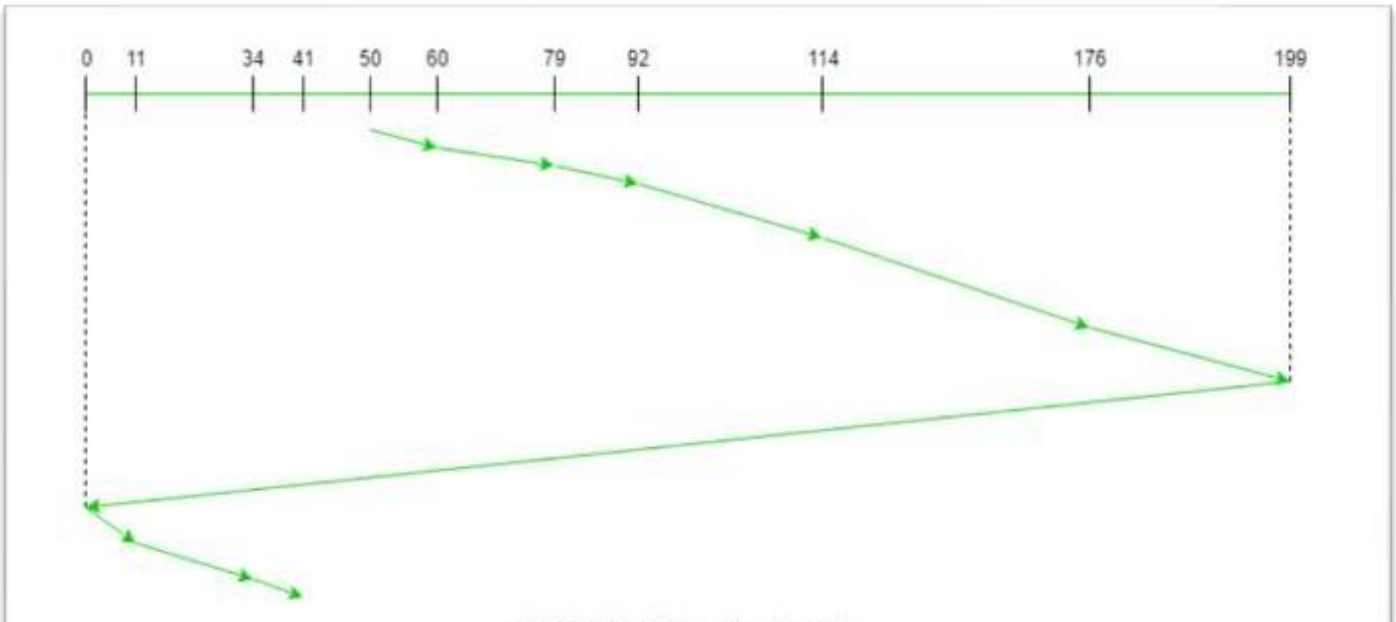
Advantages of C-SCAN (Circular Elevator) Disk Scheduling Algorithm:

- Works well with moderate to heavy loads.
- It provides better response time and uniform waiting time.

Disadvantages of C-SCAN (Circular Elevator) Disk Scheduling Algorithm:

- May not be fair to service requests for tracks at the extreme end.
- It has more seek movements as compared to the SCAN Algorithm.

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



C-SCAN Disk Scheduling Algorithm

LOOK Algorithm:

The LOOK algorithm operates by scanning the disk in a specific direction, but instead of going all the way to the end of the disk before reversing direction like the SCAN algorithm, it reverses direction as soon as it reaches the last request in the current direction. It is used to reduce the amount of time it takes to access data on a hard disk drive by minimizing the seek time between read/write operations.

Advantages of the LOOK Disk Scheduling Algorithm:

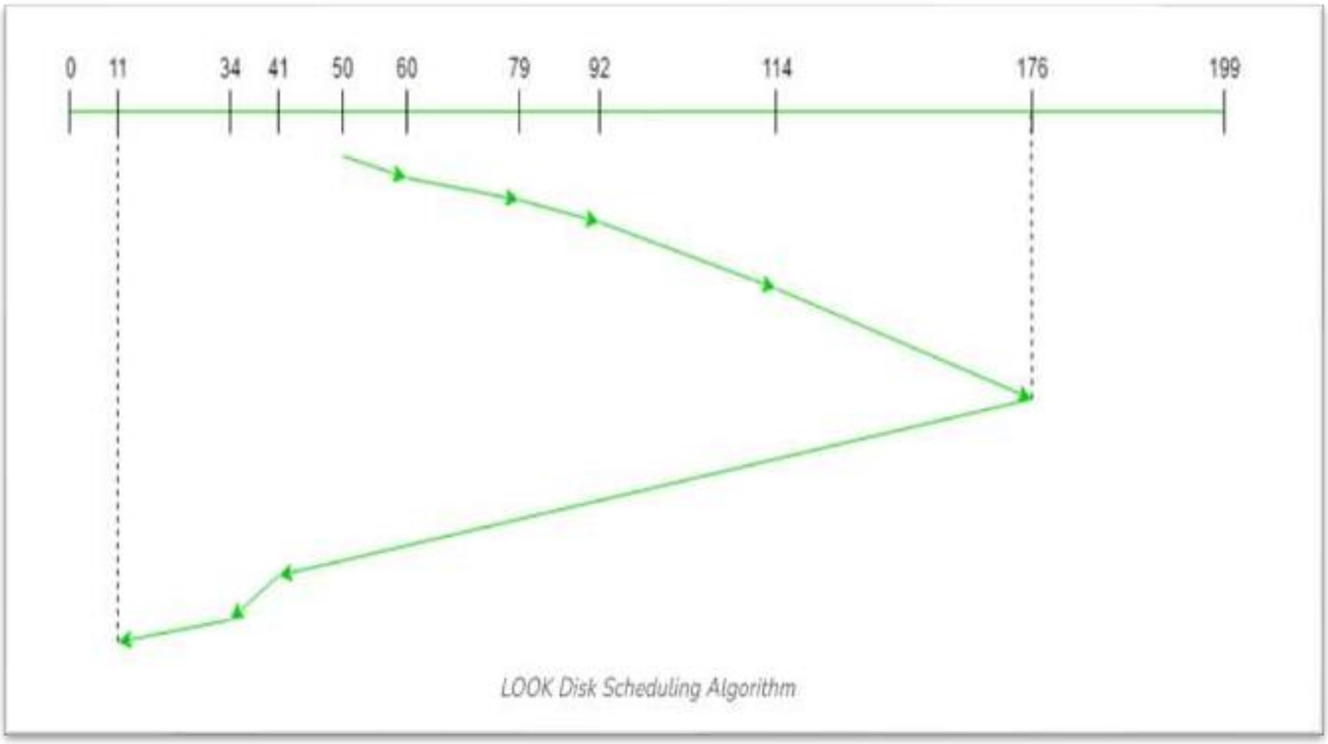
- It can provide better performance than the FCFS and SSTF algorithms because it reduces the number of head movements required to access data on the disk.
- It is relatively simple to implement and does not require a large amount of memory or processing power.
- It is efficient in terms of disk usage because it scans only the areas of the disk where data is located.

Disadvantages of the LOOK Disk Scheduling Algorithm:

- It may not be optimal in situations where there are large amounts of data to be read or written in one direction, as it could lead to a large number of requests being queued up in the opposite direction.
- It may not be suitable for real-time systems where fast response times are critical, as it does not prioritize requests based on their urgency or importance.

- It may lead to starvation of requests that are located far away from the current position of the disk head.

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



LOOK Disk Scheduling Algorithm

2.3 Prediction Engine

- Basic machine learning prediction simulation
- Predicts next likely track based on request history
- Provides a confidence metric

2.4 Performance Logging

- Logs algorithm performance in performance_log.txt
- Tracks total head movement for each scheduling strategy

3. Functionalities:

The system is designed to enhance disk scheduling efficiency by incorporating multiple advanced features. The key functionalities include:

- **Implementation of Multiple Disk Scheduling Algorithms:**

  The system supports various disk scheduling algorithms, such as **FCFS, SSTF, SCAN, C-SCAN, and LOOK**, allowing flexible selection based on system requirements.

- **Priority-Based Request Handling:**

  Requests are processed based on priority levels, ensuring that high-priority requests are serviced first, improving system responsiveness.

- **Deadline-Aware Request Management:**
  The system considers requests with deadlines and prioritizes them accordingly to prevent delays in time-sensitive operations.

- **Performance Tracking and Logging**:
  Detailed performance metrics, such as total head movement and seek time, are recorded and logged for analysis and optimization.

- **Simulated Machine Learning-Based Prediction**:
  A prediction model is integrated to analyze historical disk requests and anticipate the next track request, optimizing scheduling efficiency.

- **Flexible Request Sorting Mechanism**:
  Disk requests can be sorted based on track number, priority, or arrival time, providing adaptability to various workload scenarios.

4. Technology Used:

4.1 Programming Languages

- C (Core implementation)

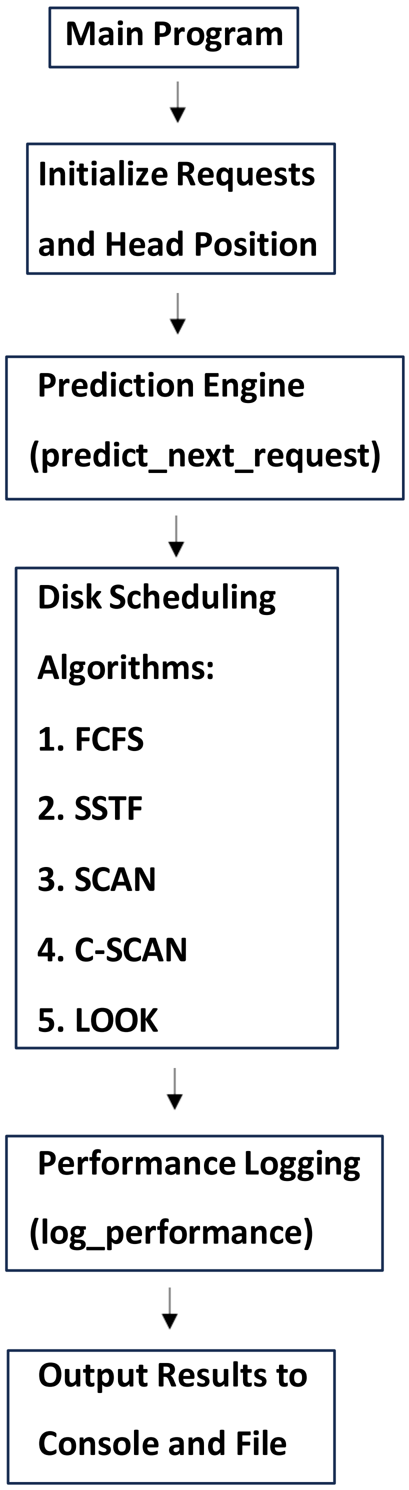
4.2 Libraries and Tools

- Standard C Libraries:
  - stdio.h: Input/Output operations
  - stdlib.h: Memory allocation, conversions
  - limits.h: Integer limits
  - stdbool.h: Boolean type support
  - string.h: String manipulation

4.3 Development Tools

- Compiler: GCC (GNU Compiler Collection)
- Version Control: Recommended GitHub
- Text Editor/IDE: Visual Studio Code, Onlinegdb

5. Flow Diagram:


Main Program
Initialize Requests and Head Position
Prediction Engine (predict_next_request)
Disk Scheduling Algorithms:
1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
Performance Logging (log_performance)
Output Results to Console and File

6. Revision Tracking

- **Repository Name:** Advanced-Disk-Scheduler
- **Recommended GitHub Link**:
- https://github.com/yashikamalik23

7. Conclusion and Future Scope:

Current Achievements:

- Implemented multiple disk scheduling strategies
- Added priority and deadline management
- Basic machine learning prediction integration

Future Enhancements:

- Advanced machine learning prediction models
- Real-time performance visualization
- Support for more complex request scenarios
- Enhanced deadline and priority handling
- Parallel processing support

8. References:

1. https://www.geeksforgeeks.org/
2. https://en.wikipedia.org/wiki/Category:Disk_scheduling_algorithms

Appendix:

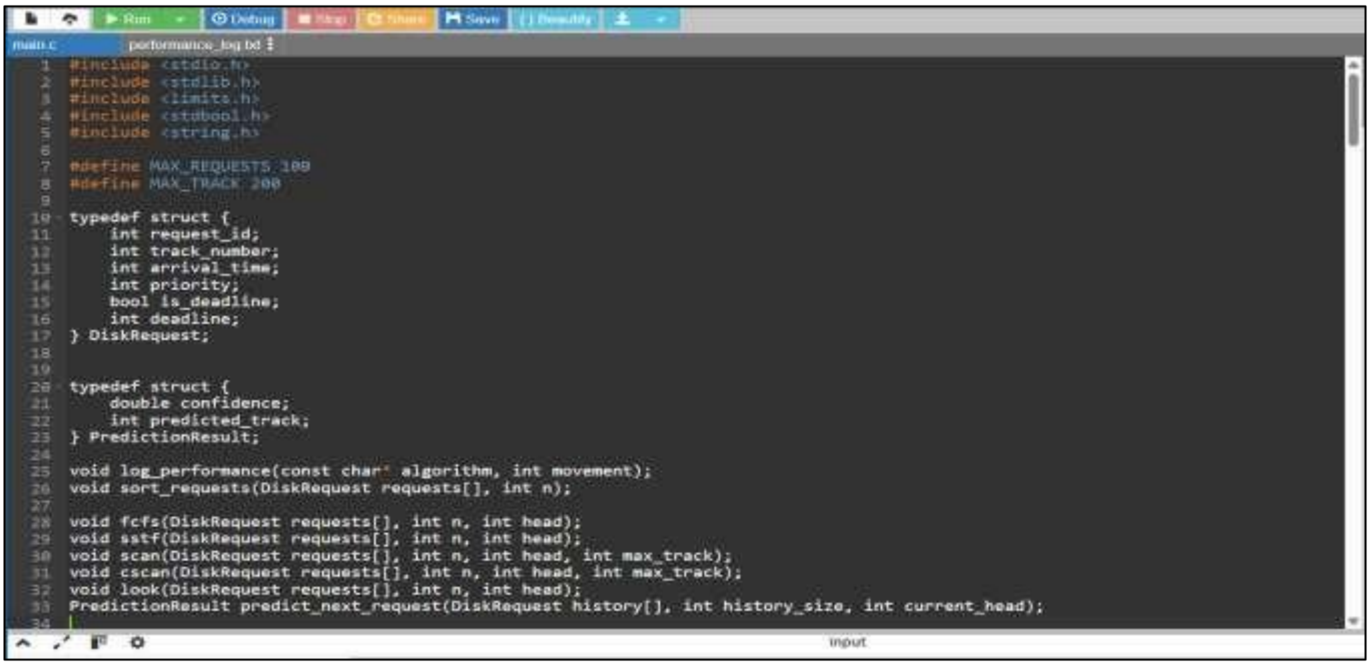
A. Problem Statement:

Disk scheduling is crucial for optimizing disk I/O performance and system efficiency. Traditional algorithms like FCFS and SSTF do not consider request prioritization, deadlines, or predictive access patterns, leading to inefficiencies.

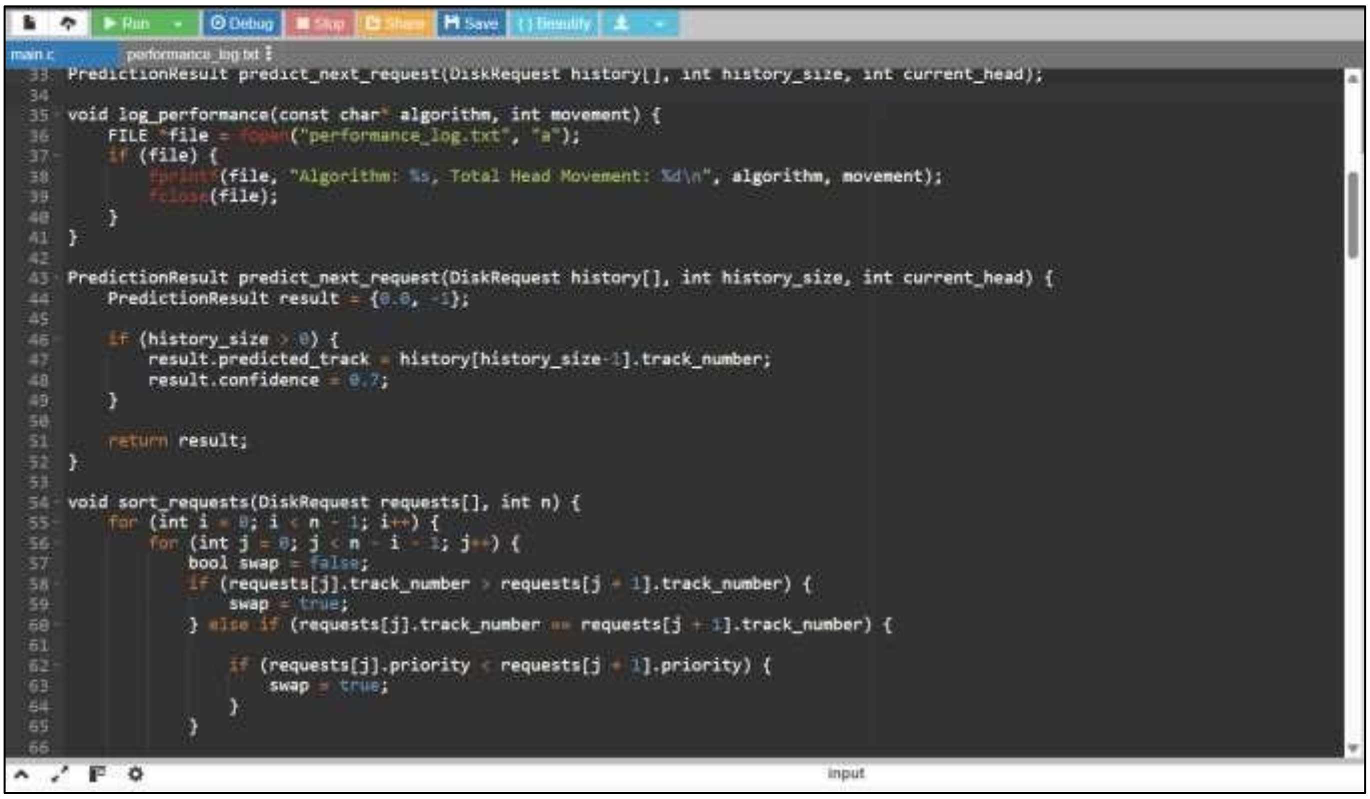
This project implements and analyzes multiple scheduling algorithms (FCFS, SSTF, SCAN, C-SCAN, and LOOK) while integrating a machine learning-based prediction engine to anticipate future requests. Additionally, request prioritization and deadline constraints are incorporated to enhance scheduling efficiency.

The goal is to evaluate and compare these algorithms based on total head movements and performance metrics, providing insights for optimizing disk scheduling in operating systems and large-scale storage systems.

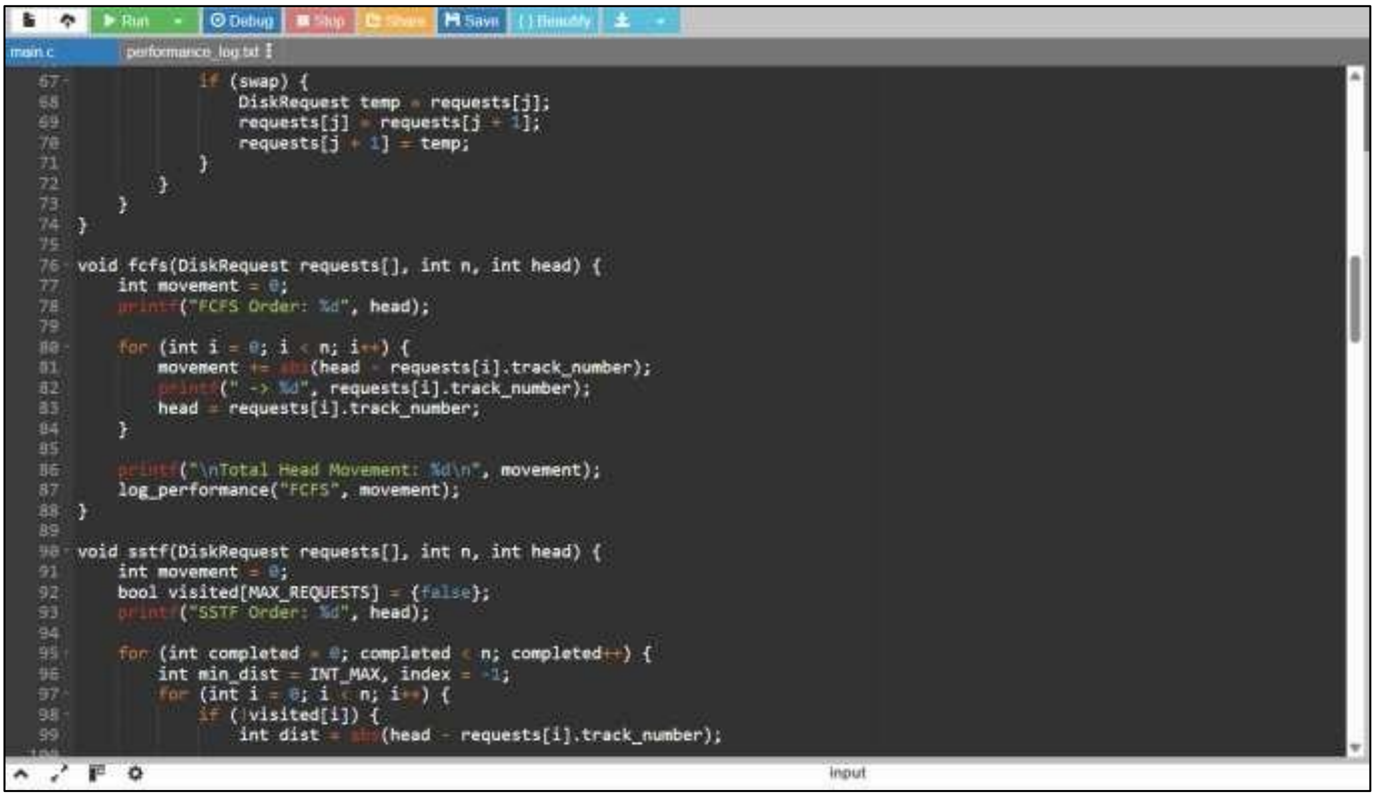
B. Code:



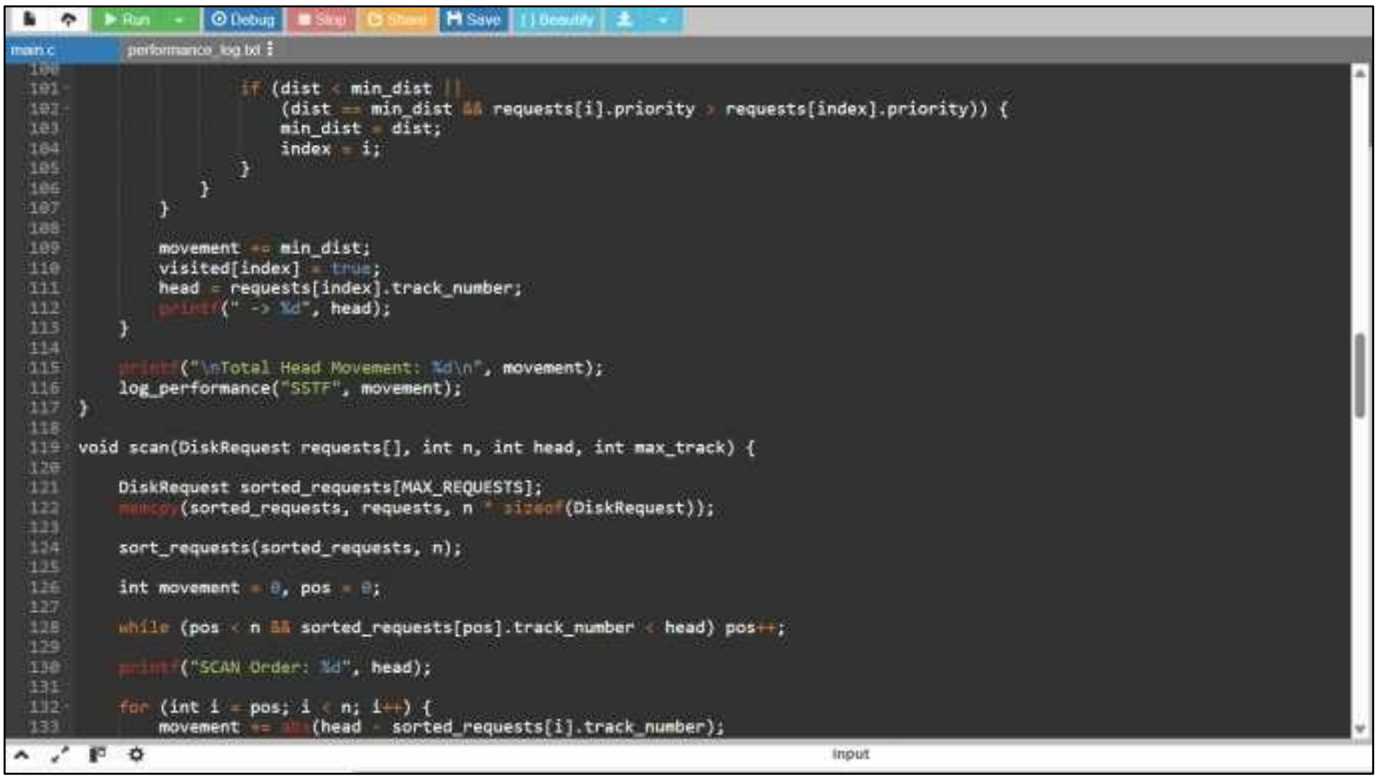
```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
#include <string.h>

#define MAX_REQUESTS 100
#define MAX_TRACK 200

typedef struct {
    int request_id;
    int track_number;
    int arrival_time;
    int priority;
    bool is_deadline;
    int deadline;
} DiskRequest;


typedef struct {
    double confidence;
    int predicted_track;
} PredictionResult;

void log_performance(const char* algorithm, int movement);
void sort_requests(DiskRequest requests[], int n);

void fcfs(DiskRequest requests[], int n, int head);
void sstf(DiskRequest requests[], int n, int head);
void scan(DiskRequest requests[], int n, int head, int max_track);
void cscan(DiskRequest requests[], int n, int head, int max_track);
void look(DiskRequest requests[], int n, int head);
PredictionResult predict_next_request(DiskRequest history[], int history_size, int current_head);
```



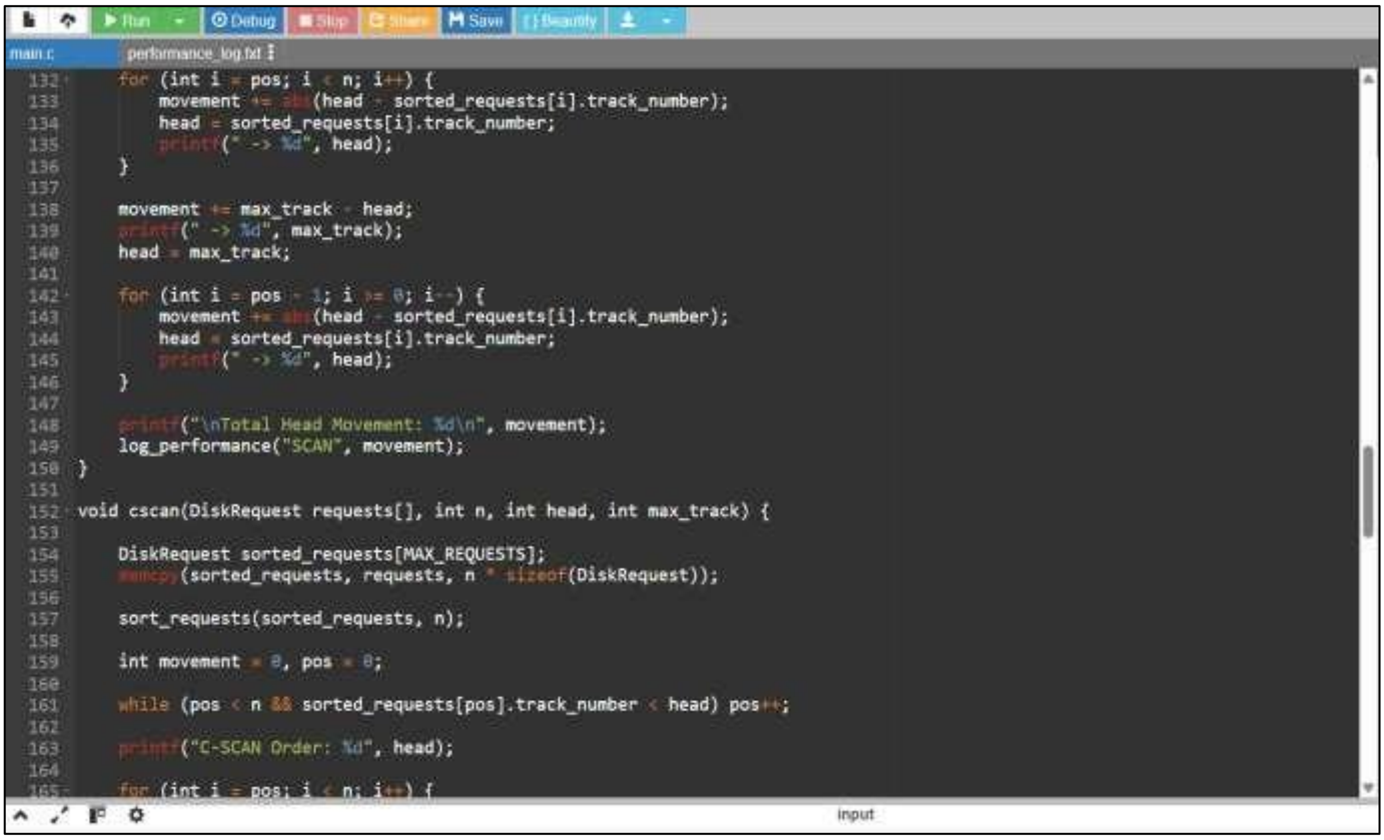
```
PredictionResult predict_next_request(DiskRequest history[], int history_size, int current_head);

void log_performance(const char* algorithm, int movement) {
    FILE *file = fopen("performance_log.txt", "a");
    if (file) {
        fprintf(file, "Algorithm: %s, Total Head Movement: %d\n", algorithm, movement);
        fclose(file);
    }
}

PredictionResult predict_next_request(DiskRequest history[], int history_size, int current_head) {
    PredictionResult result = {0.0, -1};

    if (history_size > 0) {
        result.predicted_track = history[history_size-1].track_number;
        result.confidence = 0.7;
    }

    return result;
}

void sort_requests(DiskRequest requests[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            bool swap = false;
            if (requests[j].track_number > requests[j + 1].track_number) {
                swap = true;
            } else if (requests[j].track_number == requests[j + 1].track_number) {

                if (requests[j].priority < requests[j + 1].priority) {
                    swap = true;
                }
            }

```



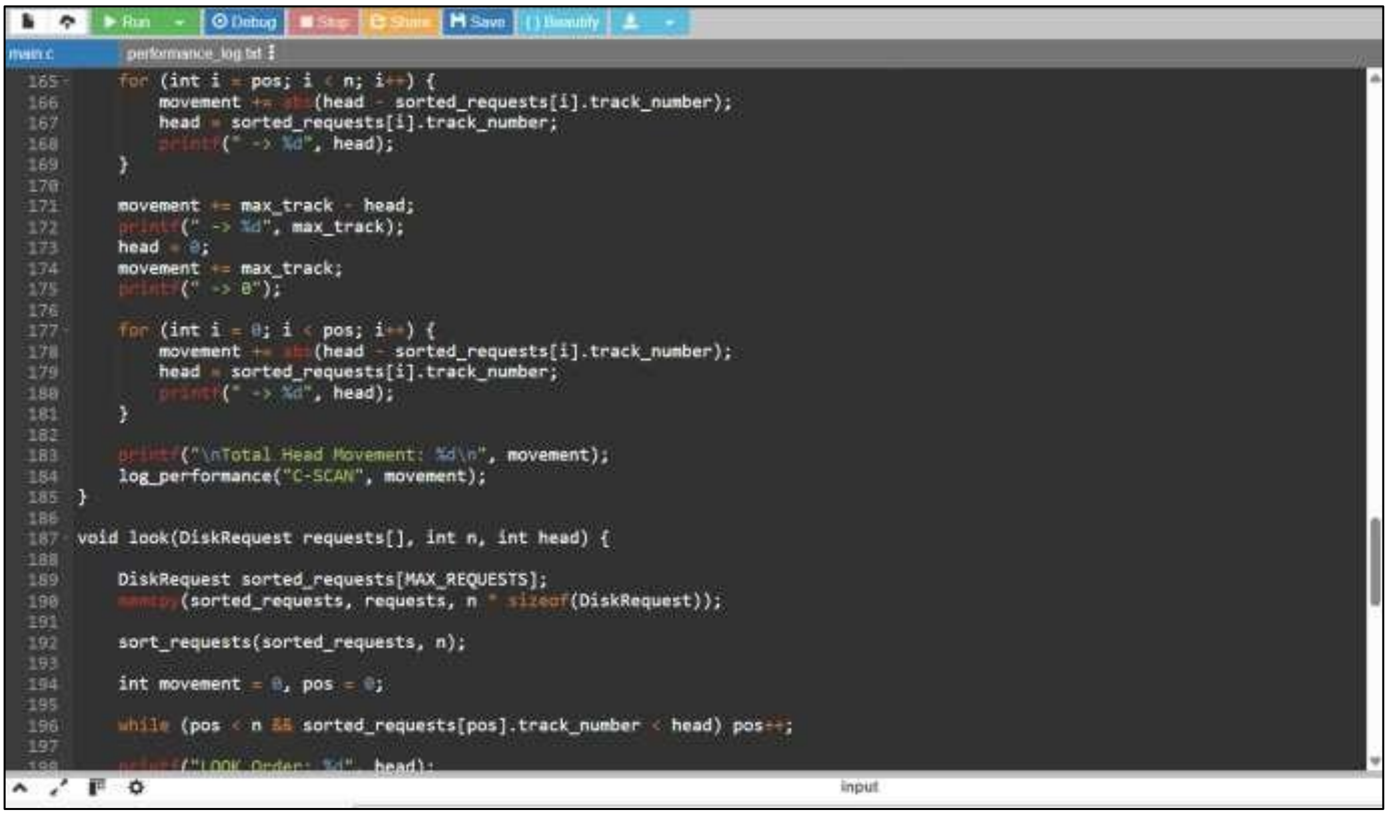
```
            if (swap) {
                DiskRequest temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }
}

void fcfs(DiskRequest requests[], int n, int head) {
    int movement = 0;
    printf("FCFS Order: %d", head);

    for (int i = 0; i < n; i++) {
        movement += abs(head - requests[i].track_number);
        printf(" -> %d", requests[i].track_number);
        head = requests[i].track_number;
    }

    printf("\nTotal Head Movement: %d\n", movement);
    log_performance("FCFS", movement);
}

void sstf(DiskRequest requests[], int n, int head) {
    int movement = 0;
    bool visited[MAX_REQUESTS] = {false};
    printf("SSTF Order: %d", head);

    for (int completed = 0; completed < n; completed++) {
        int min_dist = INT_MAX, index = -1;
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                int dist = abs(head - requests[i].track_number);
```



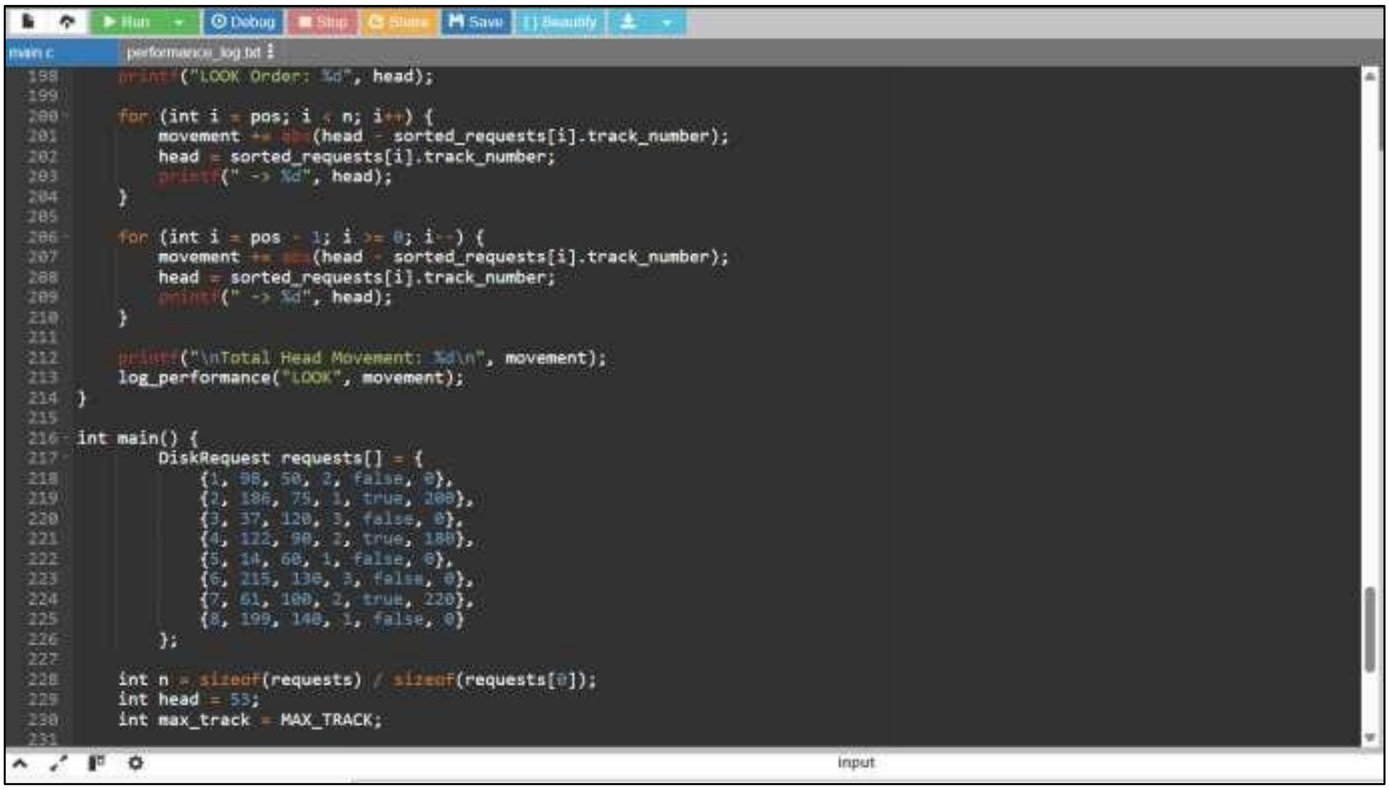
```c
                if (dist < min_dist ||
                    (dist == min_dist && requests[i].priority > requests[index].priority)) {
                    min_dist = dist;
                    index = i;
                }
            }
        }

        movement += min_dist;
        visited[index] = true;
        head = requests[index].track_number;
        printf(" -> %d", head);
    }

    printf("\nTotal Head Movement: %d\n", movement);
    log_performance("SSTF", movement);
}

void scan(DiskRequest requests[], int n, int head, int max_track) {

    DiskRequest sorted_requests[MAX_REQUESTS];
    memcpy(sorted_requests, requests, n * sizeof(DiskRequest));

    sort_requests(sorted_requests, n);

    int movement = 0, pos = 0;

    while (pos < n && sorted_requests[pos].track_number < head) pos++;

    printf("SCAN Order: %d", head);

    for (int i = pos; i < n; i++) {
        movement += abs(head - sorted_requests[i].track_number);
```



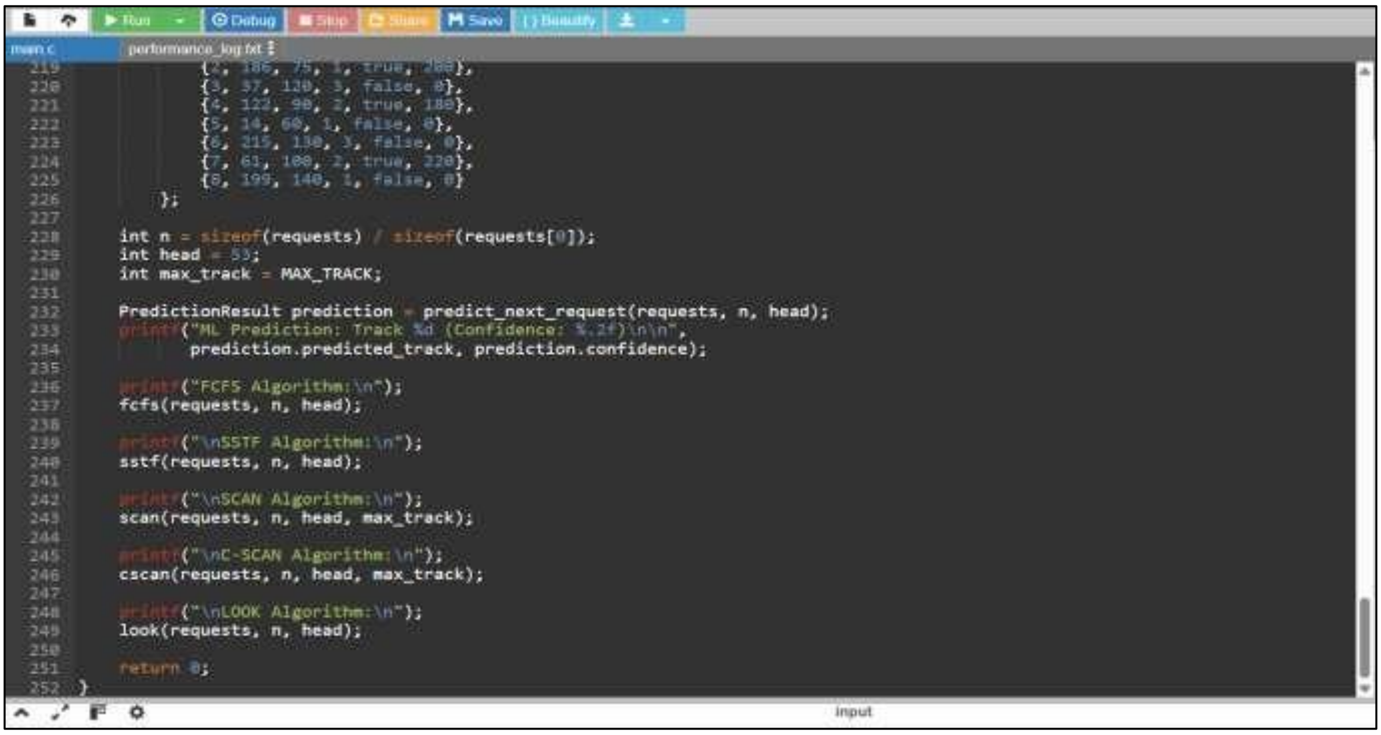
```c
    for (int i = pos; i < n; i++) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    movement += max_track - head;
    printf(" -> %d", max_track);
    head = max_track;

    for (int i = pos - 1; i >= 0; i--) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    printf("\nTotal Head Movement: %d\n", movement);
    log_performance("SCAN", movement);
}

void cscan(DiskRequest requests[], int n, int head, int max_track) {

    DiskRequest sorted_requests[MAX_REQUESTS];
    memcpy(sorted_requests, requests, n * sizeof(DiskRequest));

    sort_requests(sorted_requests, n);

    int movement = 0, pos = 0;

    while (pos < n && sorted_requests[pos].track_number < head) pos++;

    printf("C-SCAN Order: %d", head);

    for (int i = pos; i < n; i++) {
```



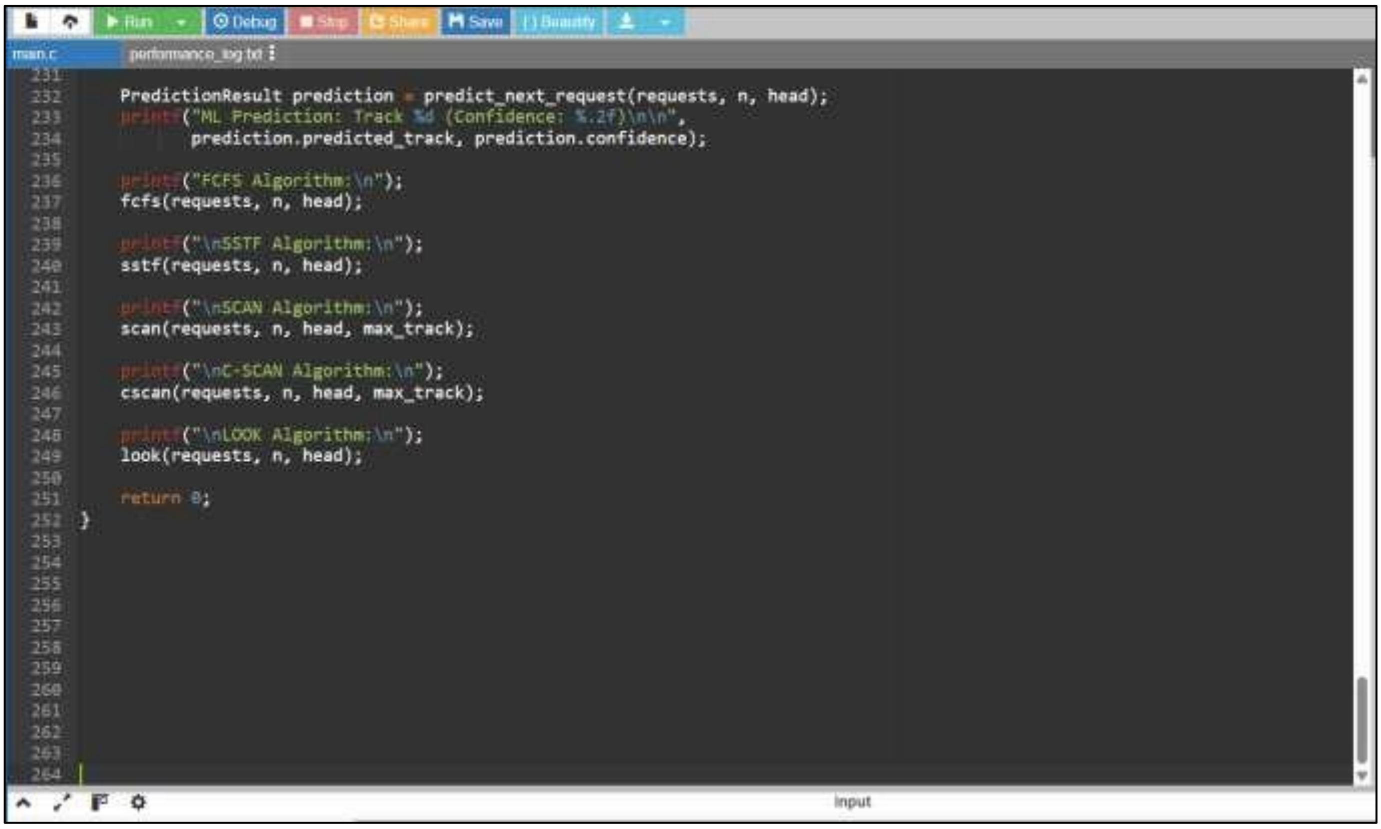
```c
    for (int i = pos; i < n; i++) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    movement += max_track - head;
    printf(" -> %d", max_track);
    head = 0;
    movement += max_track;
    printf(" -> 0");

    for (int i = 0; i < pos; i++) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    printf("\nTotal Head Movement: %d\n", movement);
    log_performance("C-SCAN", movement);
}

void look(DiskRequest requests[], int n, int head) {

    DiskRequest sorted_requests[MAX_REQUESTS];
    memcpy(sorted_requests, requests, n * sizeof(DiskRequest));

    sort_requests(sorted_requests, n);

    int movement = 0, pos = 0;

    while (pos < n && sorted_requests[pos].track_number < head) pos++;
```



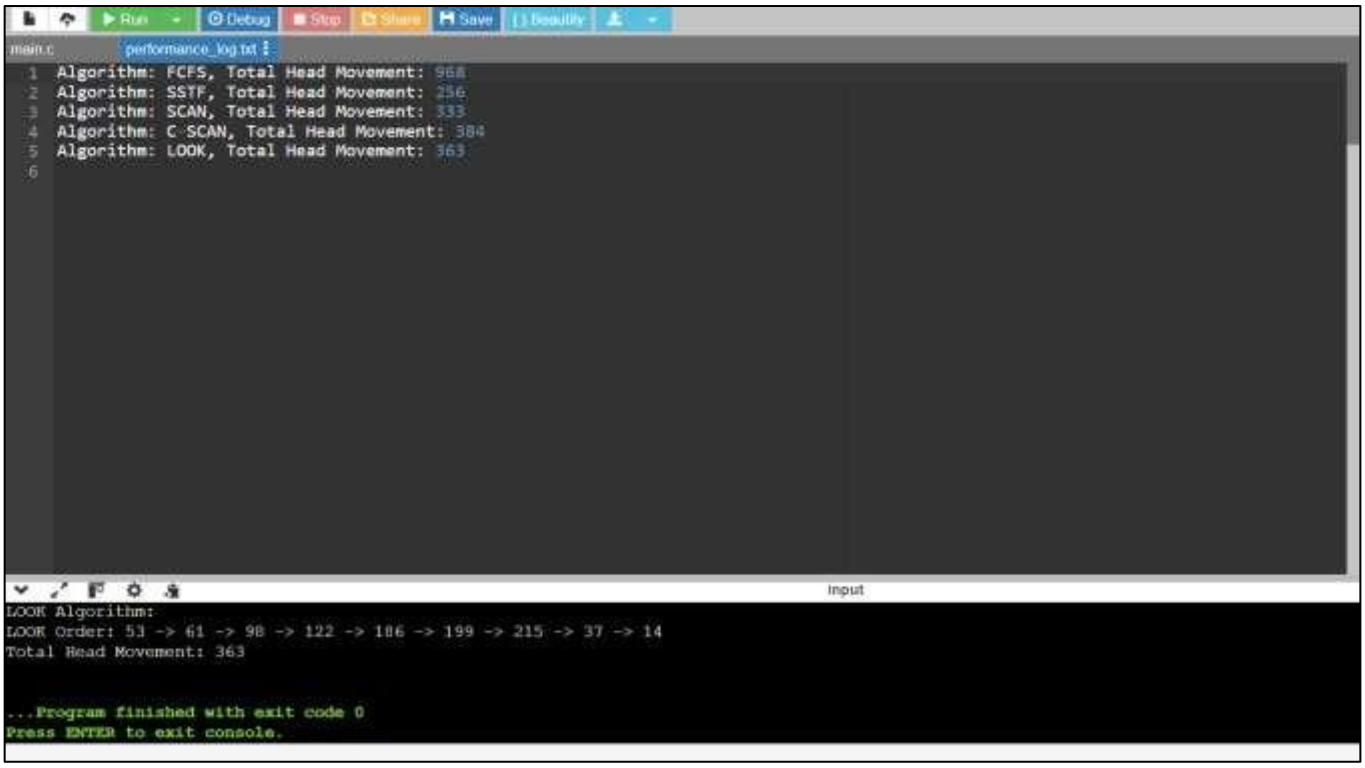
```c
    printf("LOOK Order: %d", head);

    for (int i = pos; i < n; i++) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    for (int i = pos - 1; i >= 0; i--) {
        movement += abs(head - sorted_requests[i].track_number);
        head = sorted_requests[i].track_number;
        printf(" -> %d", head);
    }

    printf("\nTotal Head Movement: %d\n", movement);
    log_performance("LOOK", movement);
}

int main() {
        DiskRequest requests[] = {
            {1, 98, 50, 2, false, 0},
            {2, 186, 75, 1, true, 200},
            {3, 37, 120, 3, false, 0},
            {4, 122, 90, 2, true, 180},
            {5, 14, 60, 1, false, 0},
            {6, 215, 130, 3, false, 0},
            {7, 61, 100, 2, true, 220},
            {8, 199, 140, 1, false, 0}
        };

    int n = sizeof(requests) / sizeof(requests[0]);
    int head = 53;
    int max_track = MAX_TRACK;
```



```c
            {2, 186, 75, 1, true, 200},
            {3, 37, 120, 3, false, 0},
            {4, 122, 90, 2, true, 180},
            {5, 14, 60, 1, false, 0},
            {6, 215, 130, 3, false, 0},
            {7, 61, 100, 2, true, 220},
            {8, 199, 140, 1, false, 0}
        };

    int n = sizeof(requests) / sizeof(requests[0]);
    int head = 53;
    int max_track = MAX_TRACK;

    PredictionResult prediction = predict_next_request(requests, n, head);
    printf("ML Prediction: Track %d (Confidence: %.2f)\n\n",
           prediction.predicted_track, prediction.confidence);

    printf("FCFS Algorithm:\n");
    fcfs(requests, n, head);

    printf("\nSSTF Algorithm:\n");
    sstf(requests, n, head);

    printf("\nSCAN Algorithm:\n");
    scan(requests, n, head, max_track);

    printf("\nC-SCAN Algorithm:\n");
    cscan(requests, n, head, max_track);

    printf("\nLOOK Algorithm:\n");
    look(requests, n, head);

    return 0;
}
```

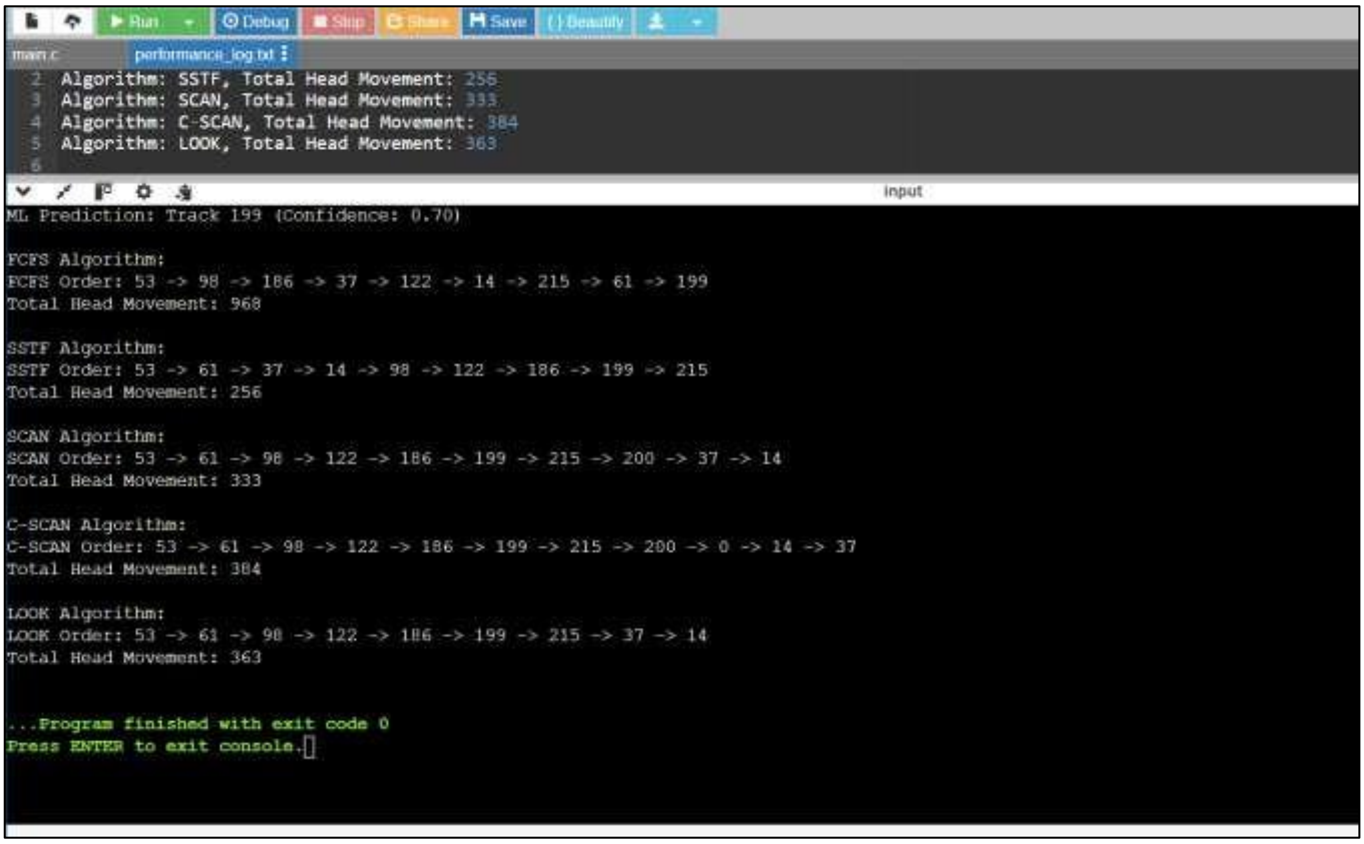


```c

    PredictionResult prediction = predict_next_request(requests, n, head);
    printf("ML Prediction: Track %d (Confidence: %.2f)\n\n",
           prediction.predicted_track, prediction.confidence);

    printf("FCFS Algorithm:\n");
    fcfs(requests, n, head);

    printf("\nSSTF Algorithm:\n");
    sstf(requests, n, head);

    printf("\nSCAN Algorithm:\n");
    scan(requests, n, head, max_track);

    printf("\nC-SCAN Algorithm:\n");
    cscan(requests, n, head, max_track);

    printf("\nLOOK Algorithm:\n");
    look(requests, n, head);

    return 0;
}
```



```
Algorithm: FCFS, Total Head Movement: 968
Algorithm: SSTF, Total Head Movement: 256
Algorithm: SCAN, Total Head Movement: 333
Algorithm: C-SCAN, Total Head Movement: 384
Algorithm: LOOK, Total Head Movement: 363
```

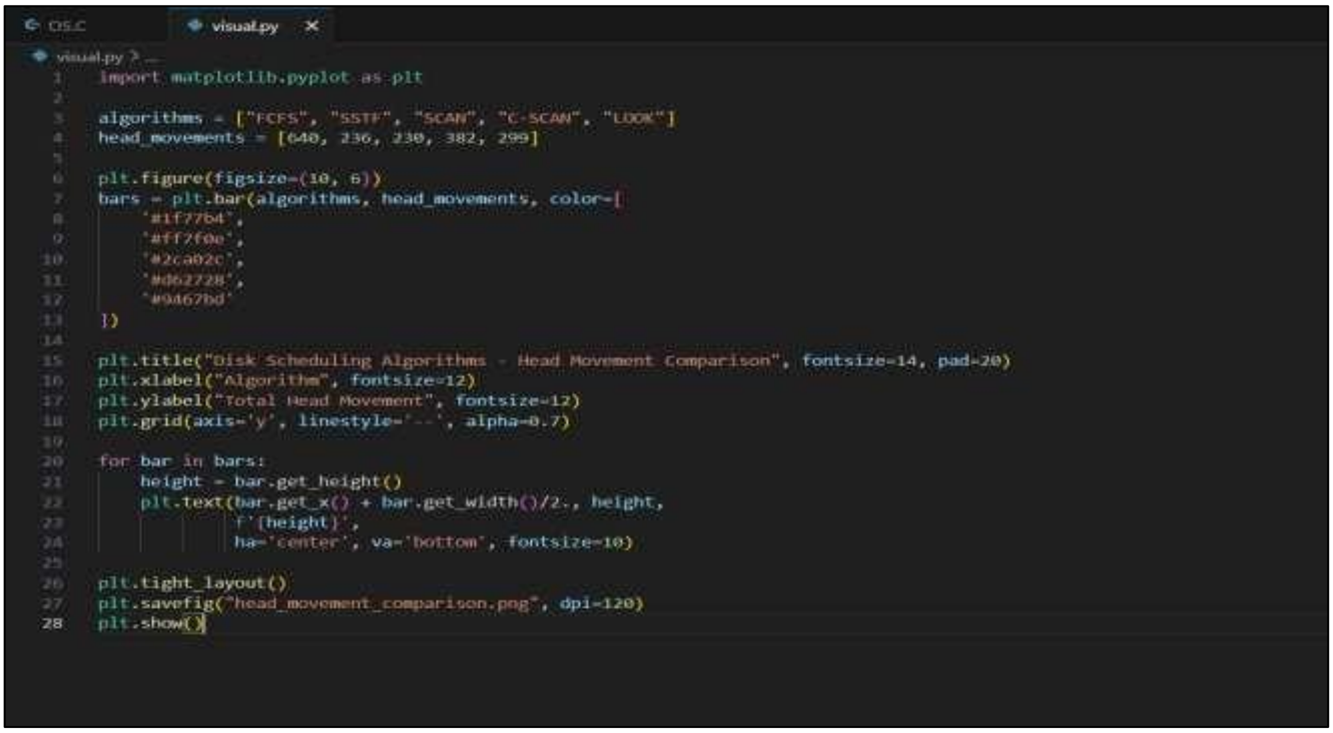
```
LOOK Algorithm:
LOOK Order: 53 -> 61 -> 98 -> 122 -> 186 -> 199 -> 215 -> 37 -> 14
Total Head Movement: 363


...Program finished with exit code 0
Press ENTER to exit console.
```



```
Algorithm: SSTF, Total Head Movement: 256
Algorithm: SCAN, Total Head Movement: 333
Algorithm: C-SCAN, Total Head Movement: 384
Algorithm: LOOK, Total Head Movement: 363
```

```
ML Prediction: Track 199 (Confidence: 0.70)

FCFS Algorithm:
FCFS Order: 53 -> 98 -> 186 -> 37 -> 122 -> 14 -> 215 -> 61 -> 199
Total Head Movement: 968

SSTF Algorithm:
SSTF Order: 53 -> 61 -> 37 -> 14 -> 98 -> 122 -> 186 -> 199 -> 215
Total Head Movement: 256

SCAN Algorithm:
SCAN Order: 53 -> 61 -> 98 -> 122 -> 186 -> 199 -> 215 -> 200 -> 37 -> 14
Total Head Movement: 333

C-SCAN Algorithm:
C-SCAN Order: 53 -> 61 -> 98 -> 122 -> 186 -> 199 -> 215 -> 200 -> 0 -> 14 -> 37
Total Head Movement: 384

LOOK Algorithm:
LOOK Order: 53 -> 61 -> 98 -> 122 -> 186 -> 199 -> 215 -> 37 -> 14
Total Head Movement: 363


...Program finished with exit code 0
Press ENTER to exit console.
```

Visualization:



```python
import matplotlib.pyplot as plt

algorithms = ["FCFS", "SSTF", "SCAN", "C-SCAN", "LOOK"]
head_movements = [640, 236, 230, 382, 299]

plt.figure(figsize=(10, 6))
bars = plt.bar(algorithms, head_movements, color=[
    '#1f77b4',
    '#ff7f0e',
    '#2ca02c',
    '#d62728',
    '#9467bd'
])

plt.title("Disk Scheduling Algorithms - Head Movement Comparison", fontsize=14, pad=20)
plt.xlabel("Algorithm", fontsize=12)
plt.ylabel("Total Head Movement", fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.7)

for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height,
             f'{height}',
             ha='center', va='bottom', fontsize=10)

plt.tight_layout()
plt.savefig("head_movement_comparison.png", dpi=120)
plt.show()
```

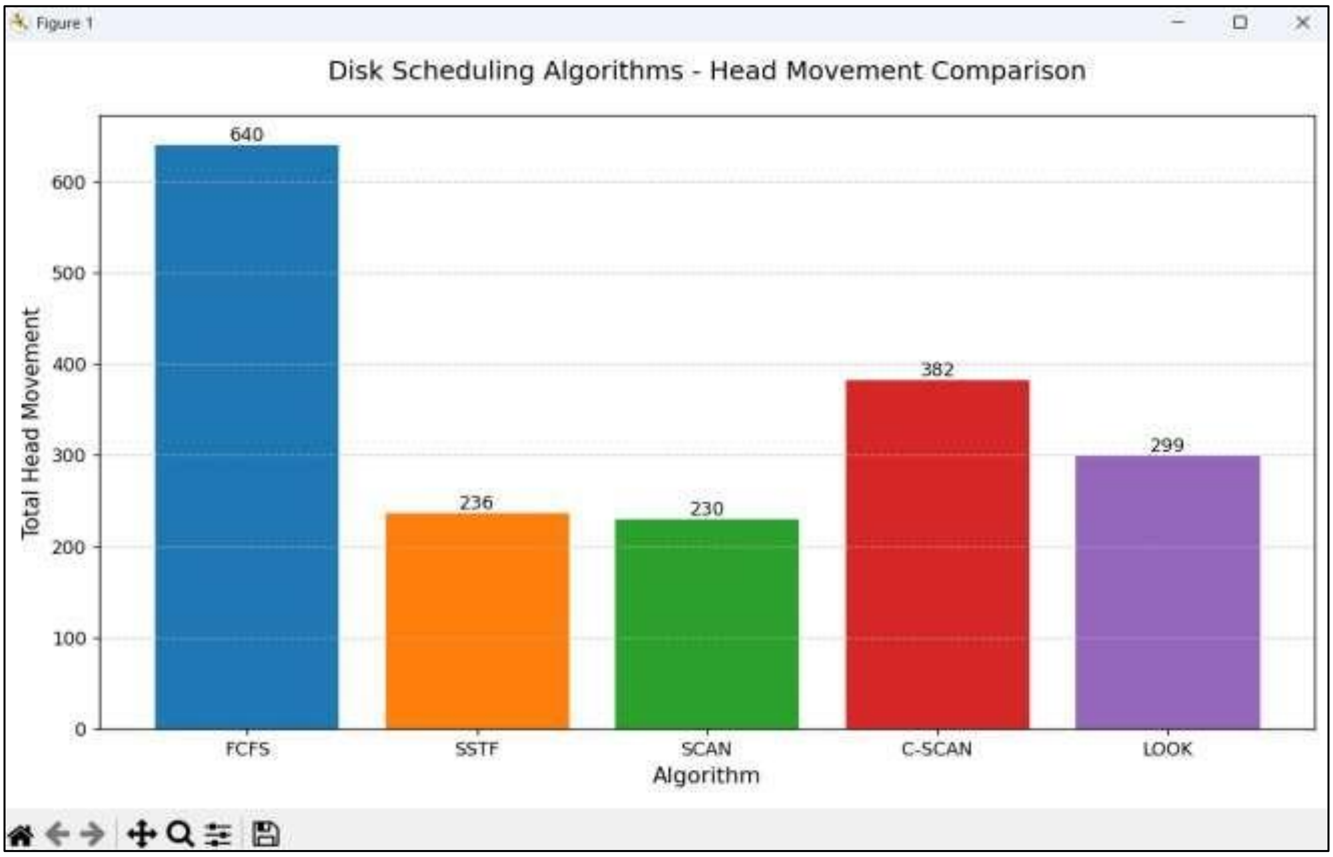


Figure 1
Disk Scheduling Algorithms - Head Movement Comparison
Total Head Movement
Algorithm
FCFS
SSTF
SCAN
C-SCAN
LOOK