Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavrepalanchowk

Assignment
Of
“COMP 307”

Submitted by:
Aaditya K.C (05)

Level:UNG CS(III/I)

Submitted to: Subhadra Joshi
Department of Computer Science and Engineering
Submission Date: 1/18/2026

Assignment Task 1: CPU Scheduling Algorithms

Algorithms Implemented

1. First Come First Serve (FCFS)
2. Shortest Job First (SJF)
    - Non-preemptive
    - Preemptive (SRTF)
3. Round Robin (RR)
4. Priority Scheduling (Non-preemptive)

Input Parameters

- Arrival Time (AT)
- Burst Time (BT)
- Priority (for Priority Scheduling)
- Time Quantum (for Round Robin)

Performance Metrics

- Completion Time (CT)
- Turnaround Time (TAT) $TAT=CT-AT$
- Waiting Time (WT) $WT=TAT-BT$

Observation

- FCFS is simple but may cause the convoy effect.
- SJF provides minimum average waiting time.
- Round Robin is suitable for time-sharing systems.
- Priority scheduling may cause starvation.

a) FCFS (First Come First Serve)

Source Code:

question1 > fcfs.py > ...

```python
# FCFS Scheduling

n = int(input("Enter number of processes: "))

arrival = []
burst = []

for i in range(n):
    print(f"\nProcess P{i+1}")
    arrival.append(int(input("Arrival Time: ")))
    burst.append(int(input("Burst Time: ")))

# Store original order
processes = [[i+1, arrival[i], burst[i]] for i in range(n)]

# Sort by arrival time
processes.sort(key=lambda x: x[1])

completion = [0] * n
waiting = [0] * n
turnaround = [0] * n

# First process completion time
completion[0] = processes[0][1] + processes[0][2]

# Completion time for others
for i in range(1, n):
    completion[i] = completion[i-1] + processes[i][2]

# Turnaround & Waiting time
for i in range(n):
    turnaround[i] = completion[i] - processes[i][1]
    waiting[i] = turnaround[i] - processes[i][2]

print("\nP\tAT\tBT\tCT\tWT\tTAT")
for i in range(n):
    print(f"P{processes[i][0]}\t{processes[i][1]}\t{processes[i][2]}\t{completion[i]}\t{waiting[i]}\t{turnaround[i]}")

print(f"\nAverage Waiting Time: {sum(waiting)/n}")
print(f"Average Turnaround Time: {sum(turnaround)/n}")

# Gantt Chart - Execution Order
print("\nGantt Chart (Execution Order):")
gantt = " -> ".join([f"P{processes[i][0]}" for i in range(n)])
print(gantt)
```

Sample Input:

```
Process P1
Arrival Time: 2
Burst Time: 4

Process P2
Arrival Time: 1
Burst Time: 4

Process P3
Arrival Time: 4
Burst Time: 2

Process P4
Arrival Time: 5
Burst Time: 3
Burst Time: 2

Process P4
Arrival Time: 5
Burst Time: 3

Process P5
```

Output Table:

```
P         AT        BT        CT        WT        TAT
P2        1         4         5         0         4
P1        2         4         9         3         7
P5        2         5         14        7         12
P3        4         2         16        10        12
P4        5         3         19        11        14

Average Waiting Time: 6.2
Average Turnaround Time: 9.8
```

Gantt Chart:

```
Gantt Chart (Execution Order):
P2 -> P1 -> P5 -> P3 -> P4
```

b) SJF (Preemptive) (SRTF)

Source Code:

oslab > question1 > srtf.py > ...

```python
# SJF Preemptive (SRTF)

n = int(input("Enter number of processes: "))

arrival = []
burst = []
remaining = []

for i in range(n):
    arrival.append(int(input(f"\nArrival Time of P{i+1}: ")))
    burst.append(int(input(f"Burst Time of P{i+1}: ")))
    remaining.append(burst[i])

completion = [0] * n
waiting = [0] * n
turnaround = [0] * n

time = 0
completed = 0
execution_order = []

while completed < n:
    idx = -1
    min_bt = 9999

    for i in range(n):
        if arrival[i] <= time and remaining[i] > 0 and remaining[i] < min_bt:
            min_bt = remaining[i]
            idx = i

    if idx == -1:
        time += 1
        continue

    remaining[idx] -= 1
    time += 1

    # Track only when process completes or first executes
    if remaining[idx] == 0:
        completed += 1
        completion[idx] = time
        turnaround[idx] = completion[idx] - arrival[idx]
        waiting[idx] = turnaround[idx] - burst[idx]

    if not execution_order or execution_order[-1] != f"P{idx+1}":
        execution_order.append(f"P{idx+1}")

print("\nP\tAT\tBT\tCT\tWT\tTAT")
for i in range(n):
    print(f"P{i+1}\t{arrival[i]}\t{burst[i]}\t{completion[i]}\t{waiting[i]}\t{turnaround[i]}")

print(f"\nAverage Waiting Time: {sum(waiting)/n}")
print(f"Average Turnaround Time: {sum(turnaround)/n}")

# Gantt Chart - Execution Order
print("\nGantt Chart (Execution Order):")
```

Sample Input:

```
Arrival Time of P1: 1
Burst Time of P1: 3

Arrival Time of P2: 2
Burst Time of P2: 4

Arrival Time of P3: 1
Burst Time of P3: 4

Arrival Time of P4: 3
Burst Time of P4: 4
```

Output Table:

```
P       AT      BT      CT      WT      TAT
P1      1       3       4       0       3
P2      2       4       8       2       6
P3      1       4       12      7       11
P4      3       4       16      9       13

Average Waiting Time: 4.5
Average Turnaround Time: 8.25
```

Gantt Chart:

```
Gantt Chart (Execution Order):
P1 -> P2 -> P3 -> P4
```

ii. SJF (non Preemptive)

Source Code:

oslab > question1 > sjf.py > ...

```python
# SJF Non-Preemptive Scheduling

n = int(input("Enter number of processes: "))

processes = []

for i in range(n):
    at = int(input(f"\nArrival Time of P{i+1}: "))
    bt = int(input(f"Burst Time of P{i+1}: "))
    processes.append([i+1, at, bt])

# Store original process data
original_processes = [p[:] for p in processes]

ct = [0] * n
wt = [0] * n
tat = [0] * n

processed = []
current_time = 0
execution_order = []  # Track execution order

for _ in range(n):
    # Find the process with minimum burst time that has arrived
    best = -1
    min_bt = float('inf')

    for i in range(n):
        if i not in processed and original_processes[i][1] <= current_time and original_processes[i][2] < min_bt:
            min_bt = original_processes[i][2]
            best = i

    # If no process has arrived, jump to the next arrival time
    if best == -1:
        best = min([i for i in range(n) if i not in processed], key=lambda x: original_processes[x][1])
        current_time = original_processes[best][1]

    # Execute the process
    current_time += original_processes[best][2]
    ct[best] = current_time
    tat[best] = ct[best] - original_processes[best][1]
    wt[best] = tat[best] - original_processes[best][2]
    processed.append(best)
    execution_order.append(f"P{original_processes[best][0]}")

print("\nP\tAT\tBT\tCT\tWT\tTAT")
for i in range(n):
    print(f"P{original_processes[i][0]}\t{original_processes[i][1]}\t{original_processes[i][2]}\t{ct[i]}\t{wt[i]}\t{ta

print(f"\nAverage Waiting Time: {sum(wt)/n}")
print(f"Average Turnaround Time: {sum(tat)/n}")

# Gantt Chart - Execution Order
print("\nGantt Chart (Execution Order):")
```

Sample Input:

```
Enter number of processes: 5

Arrival Time of P1: 1
Burst Time of P1: 4

Arrival Time of P2: 2
Burst Time of P2: 3

Arrival Time of P3: 5
Burst Time of P3: 1

Arrival Time of P4: 7
Burst Time of P4: 3

Arrival Time of P5: 3
Burst Time of P5: 6
```

Output Table:

P	AT	BT	CT	WT	TAT
P1	1	4	5	0	4
P2	2	3	9	4	7
P3	5	1	6	0	1
P4	7	3	12	2	5
P5	3	6	18	9	15

```
Average Waiting Time: 3.0
Average Turnaround Time: 6.4
```

Gantt Chart:

```
Gantt Chart (Execution Order):
P1 -> P3 -> P2 -> P4 -> P5
```

c) Round Robin

Source Code:

rr.py U ×

oslab > question1 > rr.py > ...

```python
n = int(input("Enter number of processes: "))
quantum = int(input("Enter Time Quantum: "))

arrival = []
burst = []
remaining = []

for i in range(n):
    arrival.append(int(input(f"\nArrival Time of P{i+1}: ")))
    burst.append(int(input(f"Burst Time of P{i+1}: ")))
    remaining.append(burst[i])

completion = [0] * n
waiting = [0] * n
turnaround = [0] * n

time = 0
completed = 0
queue = collections.deque()
processed_list = [False] * n
execution_order = []

while completed < n:
    # Add newly arrived processes to queue
    for i in range(n):
        if arrival[i] <= time and not processed_list[i] and remaining[i] > 0 and i not in queue:
            queue.append(i)

    if not queue:
        # No process ready, jump to next arrival
        next_arrival = min([arrival[i] for i in range(n) if remaining[i] > 0])
        time = next_arrival
        for i in range(n):
            if arrival[i] == next_arrival and remaining[i] > 0:
                queue.append(i)
                break
    else:
        # Execute process from front of queue
        i = queue.popleft()

        if remaining[i] > quantum:
            time += quantum
            remaining[i] -= quantum
            execution_order.append(f"P{i+1}")
            queue.append(i)  # Put back in queue
        else:
            time += remaining[i]
            execution_order.append(f"P{i+1}")
            remaining[i] = 0
            completion[i] = time
            turnaround[i] = completion[i] - arrival[i]
            waiting[i] = turnaround[i] - burst[i]
            completed += 1
            processed_list[i] = True
```

Sample Input:

```
Enter number of processes: 5
Enter Time Quantum: 3

Arrival Time of P1: 1
Burst Time of P1: 2

Arrival Time of P2: 2
Burst Time of P2: 6

Arrival Time of P3: 3
Burst Time of P3: 6

Arrival Time of P4: 4
Burst Time of P4: 3

Arrival Time of P5: 6
Burst Time of P5: 2
```

Output Table:

P	AT	BT	CT	WT	TAT
P1	1	2	3	0	2
P2	2	6	12	4	10
P3	3	6	20	11	17
P4	4	3	15	8	11
P5	6	2	17	9	11

```
Average Waiting Time: 6.4
Average Turnaround Time: 10.2
```

Gantt Chart:

```
Gantt Chart (Execution Order):
P1 -> P2 -> P3 -> P2 -> P4 -> P5 -> P3
```

d) Priority scheduling

```
ritysch.py U ×
> question1 > prioritysch.py > ...
# Priority Scheduling (Lower number = Higher Priority)

n = int(input("Enter number of processes: "))

processes = []

for i in range(n):
    at = int(input(f"\nArrival Time of P{i+1}: "))
    bt = int(input(f"Burst Time of P{i+1}: "))
    pr = int(input(f"Priority of P{i+1}: "))
    processes.append([i+1, at, bt, pr])

# Store original process data
original_processes = [p[:] for p in processes]

ct = [0] * n
wt = [0] * n
tat = [0] * n

processed = []
current_time = 0
execution_order = []  # Track execution order

for _ in range(n):
    # Find the process with highest priority (lowest priority number) that has arrived
    best = -1
    min_pr = float('inf')

    for i in range(n):
        if i not in processed and original_processes[i][1] <= current_time and original_processes[i][3] < min_
            min_pr = original_processes[i][3]
            best = i

    # If no process has arrived, jump to the next arrival time
    if best == -1:
        best = min([i for i in range(n) if i not in processed], key=lambda x: original_processes[x][1])
        current_time = original_processes[best][1]

    # Execute the process
    current_time += original_processes[best][2]
    ct[best] = current_time
    tat[best] = ct[best] - original_processes[best][1]
    wt[best] = tat[best] - original_processes[best][2]
    processed.append(best)
    execution_order.append(f"P{original_processes[best][0]}")

print("\nP\tAT\tBT\tPR\tCT\tWT\tTAT")
for i in range(n):
    print(f"P{original_processes[i][0]}\t{original_processes[i][1]}\t{original_processes[i][2]}\t{original_proc

print(f"\nAverage Waiting Time: {sum(wt)/n}")
print(f"Average Turnaround Time: {sum(tat)/n}")
```

Sample Input:

```
Arrival Time of P1: 2
Burst Time of P1: 3
Priority of P1: 1

Arrival Time of P2: 4
Burst Time of P2: 2
Priority of P2: 3

Arrival Time of P3: 5
Burst Time of P3: 2
Priority of P3: 2

Arrival Time of P4: 6
Burst Time of P4: 3
Priority of P4: 1

Arrival Time of P5: 6
Burst Time of P5: 5
```

Output Table:

```
P       AT      BT      PR      CT      WT      TAT
P1      2       3       1       5       0       3
P2      4       2       3       12      6       8
P3      5       2       2       7       0       2
P4      6       3       1       10      1       4
P5      6       5       4       17      6       11

Average Waiting Time: 2.6
Average Turnaround Time: 5.6
```

Gantt Chart:

```
Gantt Chart (Execution Order):
P1 -> P3 -> P4 -> P2 -> P5
```

Assignment Task 3: Page Replacement Algorithms

Aim

To simulate different page replacement algorithms and analyze their performance in terms of page faults and hit ratio.

Algorithms Implemented

1. First In First Out (FIFO)
2. Optimal (OPT)
3. Least Recently Used (LRU)

Input Description

- Reference String: A sequence of page numbers referenced by the CPU.
- Number of Frames: Total frames available in physical memory.

Performance Metrics

- **Page Fault**: Occurs when a referenced page is not present in memory.
- **Hit**: Occurs when the referenced page is already present in memory.
-

Result and Observation

- FIFO is easy but inefficient in some cases.
- OPT produces the least number of page faults.
- LRU balances performance and practicality.

a) FIFO (First in First Out)

Source Code:

oslab > question3 > fifo.py > ...

```python
ref = list(map(int, input("Enter reference string: ").split()))
frames = int(input("Enter number of frames: "))

memory = []
faults = 0
hits = 0

print("\nRef\tFrames\t\tStatus")
print("--------------------------------")

for page in ref:
    if page in memory:
        hits += 1
        status = "HIT"
    else:
        faults += 1
        status = "FAULT"
        if len(memory) < frames:
            memory.append(page)
        else:
            memory.pop(0)
            memory.append(page)

    print(f"{page}\t{memory}\t{status}")

print("\nTotal Page Faults =", faults)
print("Hit Ratio =", hits / len(ref))
```

Output:

```
Enter reference string: 1 2 3 1 3 4 2 5 3 6 7 5 8 6 9 7 0 7 8 0
Enter number of frames: 3

Ref     Frames          Status
--------------------------------
1       [1]     FAULT
2       [1, 2]  FAULT
3       [1, 2, 3]       FAULT
1       [1, 2, 3]       HIT
3       [1, 2, 3]       HIT
4       [2, 3, 4]       FAULT
2       [2, 3, 4]       HIT
5       [3, 4, 5]       FAULT
3       [3, 4, 5]       HIT
6       [4, 5, 6]       FAULT
7       [5, 6, 7]       FAULT
5       [5, 6, 7]       HIT
8       [6, 7, 8]       FAULT
6       [6, 7, 8]       HIT
9       [7, 8, 9]       FAULT
7       [7, 8, 9]       HIT
0       [8, 9, 0]       FAULT
7       [9, 0, 7]       FAULT
8       [0, 7, 8]       FAULT
0       [0, 7, 8]       HIT

Total Page Faults = 12
Hit Ratio = 0.4
```

b) Optimal (OPT)

Source Code:

optimal.py U ×

oslab > question3 > optimal.py > ...

```python
ref = list(map(int, input("Enter reference string: ").split()))
frames = int(input("Enter number of frames: "))

memory = []
faults = 0
hits = 0

print("\nRef\tFrames\t\tStatus")
print("--------------------------------")

for i in range(len(ref)):
    page = ref[i]

    if page in memory:
        hits += 1
        status = "HIT"
    else:
        faults += 1
        status = "FAULT"

        if len(memory) < frames:
            memory.append(page)
        else:
            future = []
            for m in memory:
                if m in ref[i+1:]:
                    future.append(ref[i+1:].index(m))
                else:
                    future.append(9999)

            replace_index = future.index(max(future))
            memory[replace_index] = page

    print(f"{page}\t{memory}\t{status}")

print("\nTotal Page Faults =", faults)
print("Hit Ratio =", hits / len(ref))
```

Output:

```
Enter reference string: 1 2 3 1 3 4 2 5 3 6 7 5 8 6 9 7 0 7 8 0
Enter number of frames: 3

Ref     Frames          Status
--------------------------------
1       [1]     FAULT
2       [1, 2]  FAULT
3       [1, 2, 3]       FAULT
1       [1, 2, 3]       HIT
3       [1, 2, 3]       HIT
4       [4, 2, 3]       FAULT
2       [4, 2, 3]       HIT
5       [5, 2, 3]       FAULT
3       [5, 2, 3]       HIT
6       [5, 6, 3]       FAULT
7       [5, 6, 7]       FAULT
5       [5, 6, 7]       HIT
8       [8, 6, 7]       FAULT
6       [8, 6, 7]       HIT
9       [8, 9, 7]       FAULT
7       [8, 9, 7]       HIT
0       [8, 0, 7]       FAULT
7       [8, 0, 7]       HIT
8       [8, 0, 7]       HIT
0       [8, 0, 7]       HIT

Total Page Faults = 10
Hit Ratio = 0.5
```

c) LRU

Source Code:

lru.py U

oslab > question3 > lru.py > ...

```python
ref = list(map(int, input("Enter reference string: ").split())
frames = int(input("Enter number of frames: "))

memory = []
recent = []
faults = 0
hits = 0

print("\nRef\tFrames\t\tStatus")
print("--------------------------------")

for page in ref:
    if page in memory:
        hits += 1
        status = "HIT"
        recent.remove(page)
        recent.append(page)
    else:
        faults += 1
        status = "FAULT"

        if len(memory) < frames:
            memory.append(page)
            recent.append(page)
        else:
            lru = recent.pop(0)
            index = memory.index(lru)
            memory[index] = page
            recent.append(page)

    print(f"{page}\t{memory}\t{status}")

print("\nTotal Page Faults =", faults)
print("Hit Ratio =", hits / len(ref))
```

Output:

```
Enter reference string: 1 2 3 1 3 4 2 5 3 6 7 5 8 6 9 7 0 7 8 0
Enter number of frames: 3

Ref     Frames          Status
--------------------------------
1       [1]     FAULT
2       [1, 2]  FAULT
3       [1, 2, 3]       FAULT
1       [1, 2, 3]       HIT
3       [1, 2, 3]       HIT
4       [1, 4, 3]       FAULT
2       [2, 4, 3]       FAULT
5       [2, 4, 5]       FAULT
3       [2, 3, 5]       FAULT
6       [6, 3, 5]       FAULT
7       [6, 3, 7]       FAULT
5       [6, 5, 7]       FAULT
8       [8, 5, 7]       FAULT
6       [8, 5, 6]       FAULT
9       [8, 9, 6]       FAULT
7       [7, 9, 6]       FAULT
0       [7, 9, 0]       FAULT
7       [7, 9, 0]       HIT
8       [7, 8, 0]       FAULT
0       [7, 8, 0]       HIT

Total Page Faults = 16
Hit Ratio = 0.2
```

Assignment Task 4: Disk Scheduling Algorithms

Aim

To simulate different disk scheduling algorithms and calculate total head movement and execution order.

Algorithms Implemented

1. FCFS (First Come First Serve)
2. SSTF (Shortest Seek Time First)
3. SCAN and C-SCAN
4. LOOK and C-LOOK

Input

- Disk request queue (list of track numbers)
- Initial head position
- Disk size (required for SCAN and C-SCAN)

Performance Metrics

- **Execution Order**: Order in which disk requests are serviced.
- **Total Head Movement**: Total Head Movement = Sum of absolute differences between consecutive track positions

Observation

- FCFS is simple but results in large head movement.
- SSTF reduces head movement but may cause starvation.
- SCAN and LOOK improve fairness.
- C-SCAN and C-LOOK provide uniform waiting time.

a) FCFS (First come First Serve)

Source Code:

fcfs.py U

oslab > question4 > fcfs.py > ...

```python
# FCFS Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))

total_movement = 0
current = head
order = []

for track in disk_queue:
    total_movement += abs(track - current)
    current = track
    order.append(track)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)

```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 120

Execution Order: [123, 165, 224, 265, 198, 298, 300, 45]
Total Head Movement: 569
```

b) SSTF (Shortest Seek Time First)

Source Code:

sstf.py U

oslab > question4 > sstf.py > ...

```python
# SSTF Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))

total_movement = 0
current = head
order = []

queue = disk_queue.copy()

while queue:
    nearest = min(queue, key=lambda x: abs(x - current))
    total_movement += abs(nearest - current)
    current = nearest
    order.append(nearest)
    queue.remove(nearest)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)
```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 140

Execution Order: [123, 165, 198, 224, 265, 298, 300, 45]
Total Head Movement: 449
```

c) SCAN

Source Code:

```python
# SCAN Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))
disk_size = int(input("Enter disk size: "))
direction = input("Direction (left/right): ")

left = [i for i in disk_queue if i < head]
right = [i for i in disk_queue if i >= head]

left.sort(reverse=True)
right.sort()

order = []
total_movement = 0
current = head

if direction == "left":
    for i in left:
        total_movement += abs(current - i)
        current = i
        order.append(i)
    total_movement += current
    current = 0
    for i in right:
        total_movement += abs(current - i)
        current = i
        order.append(i)
else:
    for i in right:
        total_movement += abs(current - i)
        current = i
        order.append(i)
    total_movement += (disk_size - 1 - current)
    current = disk_size - 1
    for i in left:
        total_movement += abs(current - i)
        current = i
        order.append(i)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)
```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 12
Enter disk size: 400
Direction (left/right): right

Execution Order: [45, 123, 165, 198, 224, 265, 298, 300]
Total Head Movement: 387
```

d) CSCAN

Source Code:

```
cscan.py U ×
oslab > question4 > cscan.py > ...
# C-SCAN Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))
disk_size = int(input("Enter disk size: "))

left = [i for i in disk_queue if i < head]
right = [i for i in disk_queue if i >= head]

left.sort()
right.sort()

order = []
total_movement = 0
current = head

for i in right:
    total_movement += abs(current - i)
    current = i
    order.append(i)

total_movement += (disk_size - 1 - current)  # Move to disk_size - 1
current = disk_size - 1
total_movement += (disk_size - 1)  # Jump to 0
current = 0

for i in left:
    total_movement += abs(current - i)
    current = i
    order.append(i)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)
```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 12
Enter disk size: 400

Execution Order: [45, 123, 165, 198, 224, 265, 298, 300]
Total Head Movement: 786
```

e) LOOK

Source Code:

```
look.py U ×
oslab > question4 > look.py > ...
# LOOK Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))
direction = input("Direction (left/right): ")

left = [i for i in disk_queue if i < head]
right = [i for i in disk_queue if i >= head]

left.sort(reverse=True)
right.sort()

order = []
total_movement = 0
current = head

if direction == "left":
    for i in left:
        total_movement += abs(current - i)
        current = i
        order.append(i)
    for i in right:
        total_movement += abs(current - i)
        current = i
        order.append(i)
else:
    for i in right:
        total_movement += abs(current - i)
        current = i
        order.append(i)
    for i in left:
        total_movement += abs(current - i)
        current = i
        order.append(i)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)
```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 21
Direction (left/right): left

Execution Order: [45, 123, 165, 198, 224, 265, 298, 300]
Total Head Movement: 279
```

f) C-LOOK

Source Code:

clook.py U

oslab > question4 > clook.py > ...

```python
# C-LOOK Disk Scheduling

disk_queue = list(map(int, input("Enter disk queue: ").split()))
head = int(input("Enter initial head position: "))

left = [i for i in disk_queue if i < head]
right = [i for i in disk_queue if i >= head]

left.sort()
right.sort()

order = []
total_movement = 0
current = head

for i in right:
    total_movement += abs(current - i)
    current = i
    order.append(i)

if left:  # Only process left if it's not empty
    total_movement += abs(order[-1] - left[0])
    current = left[0]

    for i in left:
        total_movement += abs(current - i)
        current = i
        order.append(i)

print("\nExecution Order:", order)
print("Total Head Movement:", total_movement)
```

Output:

```
Enter disk queue: 123 165 224 265 198 298 300 45
Enter initial head position: 3

Execution Order: [45, 123, 165, 198, 224, 265, 298, 300]
Total Head Movement: 297
```

Conclusion:

Among all algorithms:

- **SSTF** minimized head movement
- **LOOK / C-LOOK** provided a good balance between performance and fairness
- **FCFS** was simplest but least efficient