1. AIM: To write a java program to implement the concept of first in first out job scheduling.

DESCRIPTION: This program implements the **First Come First Serve (FCFS)** CPU scheduling algorithm using Java. In FCFS scheduling, the process that arrives first is executed first. The program accepts the number of processes and their burst times as input. It then calculates:

- Waiting Time (WT)
- Turnaround Time (TAT)
- Average Waiting Time
- Average Turnaround Time

Finally, it displays the process details in tabular form and prints the **Gantt Chart** for execution order.

ALGORITHM:

Step 1: Start

Step 2: Input Data

- Step 2.1: Read the number of processes p.
- Step 2.2: Read the burst time of each process into an array bt[].

Step 3: Initialize Arrays and Variables

- Step 3.1: Initialize waiting time array wt[] and turnaround time array tat[].
- Step 3.2: Set wt[0] = 0 (first process does not wait).
- Step 3.3: Initialize total waiting time t1 = 0 and total turnaround time t2 = 0.

Step 4: Calculate Waiting Time (WT)

- Step 4.1: For each process i from 1 to p-1: wt[i] = wt[i-1] + bt[i-1]
- Step 4.2: Add wt[i] to total waiting time t1.

Step 5: Calculate Turnaround Time (TAT)

- Step 5.1: For each process i from 0 to p-1: tat[i] = bt[i] + wt[i]
- Step 5.2: Add tat[i] to total turnaround time t2.

Step 6: Calculate Average Times

- Step 6.1: Average waiting time = t1 / p
- Step 6.2: Average turnaround time = t2 / p

Step 7: Display Results

- Step 7.1: Display a table with process number, burst time, waiting time, and turnaround time.
- Step 7.2: Display average waiting time and average turnaround time.
- Step 7.3: Display **Gantt Chart** showing process execution order and completion times.

Step 8: End

CONCLUSION: This program successfully demonstrates FCFS scheduling by computing waiting times, turnaround times, averages, and visualizing execution using a Gantt Chart.

2.AIM: To write a java program to implement the concept of shortest job first scheduling.

Description: The program simulates the **SJF scheduling algorithm**, a **non-preemptive scheduling technique** in operating systems where the process with the **smallest burst time** is executed first.

The user inputs the number of processes and their burst times. The program then **sorts the processes based on burst time**, calculates **waiting time** for each process as the sum of the burst times of all previous processes, and computes the **turnaround time** as the sum of waiting time and burst time.

Finally, the program calculates the **average waiting time** and displays a table showing **process number, burst time, waiting time, and turnaround time**. This helps to evaluate the efficiency of the SJF scheduling in reducing waiting time compared to other scheduling methods like FCFS.

ALGORITHM

Step 1: Start

Step 2: Input Data

- Step 2.1: Read the number of processes n.
- Step 2.2: Read burst time (BT) for each process and store in array BT[].

Step 3: Initialize Arrays

- Step 3.1: Initialize waiting time array WT[] and turnaround time array TAT[] to 0.
- Step 3.2: Initialize total waiting time variable AWT = 0.

Step 4: Sort Processes by Burst Time

- Step 4.1: Compare each pair of burst times BT[j] and BT[j+1].
- Step 4.2: If BT[j] > BT[j+1], swap the burst times and also swap the corresponding waiting times.
- Step 4.3: Repeat until all processes are sorted in ascending order of burst time.

Step 5: Calculate Waiting Time (WT) and Turnaround Time (TAT)

- Step 5.1: For the first process, WT[0] = 0 and TAT[0] = BT[0].
- Step 5.2: For each subsequent process i (from 1 to n-1):
- WT[i] = TAT[i-1]
- TAT[i] = BT[i] + WT[i]

Step 6: Calculate Average Waiting Time (AWT)

- Step 6.1: Sum all waiting times: AWT = Σ WT[i] / n

Step 7: Display Results

- Step 7.1: Display process number, burst time, waiting time, and turnaround time for each process.
- Step 7.2: Display the average waiting time.

Step 8: End

CONCLUSION: The SJF scheduling program successfully calculates the waiting time, turnaround time, and average waiting time by executing processes in increasing order of burst time. This demonstrates how the Shortest Job First algorithm minimizes average waiting time compared to other scheduling techniques.

3. AIM: To write a java program to implement the concept of round robin job scheduling.

DESCRIPTION : This program implements the **Round Robin CPU Scheduling** algorithm using Java. Round Robin scheduling is a **preemptive** scheduling technique where each process is assigned a fixed time quantum. The CPU executes each process for a given time slice, and if the process is not completed within that time, it is moved to the back of the ready queue. The program takes the number of processes, their burst times, and the time quantum as input. It calculates the **waiting time**, **turnaround time**, and their averages, and displays them in tabular form.

ALGORITHM

Step 1: Start
Step 2: Input Data
- **Step 2.1:** Input the number of processes n.
- **Step 2.2:** Input the burst time (BT) for each process into the array bt[].
- **Step 2.3:** Input the time quantum q.

Step 3: Initialize Arrays
- **Step 3.1:** Copy burst times into a temporary array a[] to preserve original BT values.
- **Step 3.2:** Initialize waiting time array wt[] with 0.
- **Step 3.3:** Initialize turnaround time array tat[].

Step 4: Execute Round Robin Scheduling
- **Step 4.1:** Repeat the following until all processes are completed:
    - **Step 4.1.1:** For each process i from 0 to n-1:
        - **Step 4.1.1.1:** If bt[i] > q, reduce bt[i] by q and add q to the waiting time of all other unfinished processes.
        - **Step 4.1.1.2:** Else, add the remaining bt[i] to the waiting time of all other unfinished processes and set bt[i] = 0.
- **Step 4.2:** Calculate the sum of bt[] to check if all processes are completed. Repeat step 4.1 until the sum becomes 0.

Step 5: Calculate Turnaround Time (TAT)
- **Step 5.1:** For each process i, calculate tat[i] = wt[i] + a[i].

Step 6: Output Results
- **Step 6.1:** Display process number, burst time, waiting time, and turnaround time for each process.
- **Step 6.2:** Calculate and display the average waiting time and average turnaround time.

Step 7: End

CONCLUSION: The Round Robin scheduling program successfully demonstrates time-sharing CPU scheduling by allocating a fixed time quantum to each process. It ensures fairness among processes and accurately computes waiting time, turnaround time, and their averages.

4. AIM: To write a java program to implement the concept of priority scheduling.

DESCRIPTION: This program implements the Priority Scheduling algorithm used in an Operating System. Each process is assigned a priority, and the process with the highest priority (lower priority number) is executed first using a non-preemptive approach. The program takes the number of processes as input along with their burst times and priority values. It then sorts the processes based on priority, calculates the waiting time and turnaround time for each process, and finally displays the results in a tabular format.

ALGORITHM

Step 1: Start
Step 2: Input Data
- **Step 2.1:** Input the number of processes n.
- **Step 2.2:** For each process, input:
    - Burst Time (BT) into bt[].
    - Priority into pr[].
- **Step 2.3:** Assign process IDs to pid[] (e.g., 1 to n).

Step 3: Sort Processes by Priority
- **Step 3.1:** Compare the priorities of all processes.
- **Step 3.2:** Swap processes so that the process with the **highest priority (lowest number)** comes first.
- **Step 3.3:** Ensure that burst times and process IDs are swapped along with priorities.

Step 4: Calculate Waiting Time (WT)
- **Step 4.1:** Set wt[0] = 0 for the first process.
- **Step 4.2:** For each subsequent process i (from 1 to n-1):
    - wt[i] = wt[i-1] + bt[i-1].

Step 5: Calculate Turnaround Time (TAT)
- **Step 5.1:** For each process i, calculate tat[i] = wt[i] + bt[i].

Step 6: Output Results
- **Step 6.1:** Display the Process ID, Burst Time, Priority, Waiting Time, and Turnaround Time for each process.

Step 7: End

CONCLUSION: The program successfully demonstrates **Priority Scheduling** in an Operating System using Java. It efficiently orders processes based on priority and accurately computes waiting and turnaround times. This implementation helps in understanding how CPU scheduling decisions are made and how process execution order affects system performance.

5. **AIM:** To write a java program to implement the concept of banker's algorithm.

DESCRIPTION: This program implements the Banker's Algorithm to prevent deadlocks in an Operating System. It checks whether resources can be allocated safely to processes using available, claim, and allocation data. The program calculates the need of each process and determines whether all processes can complete safely. Finally, it displays whether the system is in a safe state or not.

ALGORITHM:

Step 1: Start
Step 2: Input Data
- **Step 2.1:** Input the number of processes p and the number of resources r.
- **Step 2.2:** Input the **available resources vector** av[].
- **Step 2.3:** Input the **claim matrix** claim[p][r].
- **Step 2.4:** Input the **allocation matrix** alloc[p][r].

Step 3: Calculate Initial Need
- **Step 3.1:** For each resource, calculate need[i][j] = claim[i][j] - alloc[i][j].
- **Step 3.2:** Calculate the remaining available resources after current allocations.

Step 4: Check Resource Requests
- **Step 4.1:** For each process i:
  - **Step 4.1.1:** If need[i] <= available resources, the process can complete.
  - **Step 4.1.2:** Add the resources allocated to this process back to the available resources after completion.
  - **Step 4.1.3:** Mark the process as completed.
- **Step 4.2:** Repeat until all processes are checked or no more processes can be completed in this iteration.

Step 5: Determine System State
- **Step 5.1:** If all processes are completed, print **"System is in Safe State"**.
- **Step 5.2:** If one or more processes cannot complete, print **"System is Not in Safe State"**.

Step 6: End

CONCLUSION: The program successfully demonstrates the working of **Banker's Algorithm** for deadlock avoidance. It ensures that resources are allocated safely by checking system conditions before execution. By identifying whether the system is in a safe or unsafe state, this program helps in understanding how operating systems prevent deadlocks and manage resources efficiently.

6. AIM: To implement the **Dining Philosophers Problem** using Java and demonstrate how synchronization helps avoid deadlock while multiple processes compete for shared resources.

DESCRIPTION: This program simulates the Dining Philosophers Problem, where five philosophers alternate between thinking and eating. To eat, a philosopher needs two shared forks. Each philosopher can be thinking, hungry, or eating. A hungry philosopher eats only if both neighbors are not eating; otherwise, they wait. After eating, they return to thinking and notify neighbors to check if they can eat.
If you want, I can make it even **one short paragraph** version suitable for documentation or comments. Do you want me to do that?

ALGORITHM

Step 1: Start the Program
- 1.1 Create an object of the Philo class.
- 1.2 Initialize an array state[5] to keep track of each philosopher's state (Thinking = 0, Eating = 1, Hungry = 2).

Step 2: Philosophers Become Hungry
- 2.1 Loop through all philosophers (i = 0 to 4).
- 2.2 Call pickup(i) for each philosopher:
  - 2.2.1 Set state[i] = Hungry (2)
  - 2.2.2 Print "Philosopher i is hungry"
  - 2.2.3 Call test(i) to check if the philosopher can eat immediately

Step 3: Check If a Philosopher Can Eat (test(i))
- 3.1 If state[i] == Hungry **AND** both neighbors are not eating:
  - 3.1.1 Left neighbour: (i+4)%5
  - 3.1.2 Right neighbour: (i+1)%5
- 3.2 If neighbors are not eating, set state[i] = Eating (1)
- 3.3 Print "Philosopher i is eating"

Step 4: Waiting for Forks
- 4.1 If a philosopher cannot eat after testing, print "Philosopher i is waiting"
- 4.2 Philosopher remains in Hungry state until neighbors finish eating

Step 5: Philosopher Finishes Eating (putdown(i))
- 5.1 Change state[i] = Thinking (0)
- 5.2 Print "Philosopher i is thinking"
- 5.3 Call test() for **both neighbors**:
  - 5.3.1 (i+1)%5 → Right neighbour
  - 5.3.2 (i+4)%5 → Left neighbour
- 5.4 This checks if any waiting philosophers can now start eating

Step 6: Repeat for All Philosophers
- 6.1 Loop through all philosophers again (i = 0 to 4) and call putdown(i)
- 6.2 This ensures every philosopher eventually eats safely

Step 7: End Program

CONCLUSION: The program successfully demonstrates the Dining Philosophers Problem using Java. It shows how proper synchronization can prevent deadlock when multiple processes share limited resources. This simulation helps in understanding resource allocation, process coordination, and deadlock avoidance in Operating Systems.

7. AIM: To implement a circular buffer based Producer–Consumer simulation in Java that allows inserting and removing items from a fixed-size buffer.

DESCRIPTION: This Java program simulates a simple **Producer–Consumer** system using a circular buffer of size 10. The user can choose to **produce** (insert) items into the buffer or **consume** (remove) items from it. The buffer uses in and out pointers that wrap around using modulo arithmetic.

- When producing, if the buffer is full, insertion is prevented.
- When consuming, if the buffer is empty, removal is prevented.
- The program continues until the user chooses to exit.

ALGORITHM

Step 1: Start
Step 2: Initialize an integer array buffer of size 10
Step 3: Set bufsize = 10, in = 0, out = 0
Step 4: Repeat until user enters 3 (Exit)
Step 5: Display menu:
1. Produce
2. Consume
3. Exit
Step 6: Read user choice

Step 7: If choice == 1 (Produce)
Step 7.1: Check if (in + 1) % bufsize == out
Step 7.2: If TRUE → Buffer is Full (display message)
Step 7.3: Else → Prompt for value
Step 7.4: Insert the value at buffer[in]
Step 7.5: Update in = (in + 1) % bufsize

Step 8: If choice == 2 (Consume)
Step 8.1: Check if in == out
Step 8.2: If TRUE → Buffer is Empty (display message)
Step 8.3: Else → Read value from buffer[out]
Step 8.4: Display consumed value
Step 8.5: Update out = (out + 1) % bufsize

Step 9: If choice == 3 → Exit loop
Step 10: End

CONCLUSION : The program successfully implements a circular queue (buffer) for producer and consumer operations. It handles buffer overflow and underflow conditions and repeatedly accepts user input until the exit option is selected. The correct use of modular arithmetic ensures efficient reuse of buffer space.

8. **AIM** : To implement the **Best Fit Memory Allocation** algorithm in Java for assigning files to memory blocks such that the smallest sufficient block is selected for each file.

DESCRIPTION: This Java program simulates the **Best Fit** strategy of memory management. The user enters the number of memory blocks and files along with their sizes. For each file, the program searches through the list of available blocks and assigns the file to the block that leaves the smallest leftover space (i.e., the best fitting block). Once a block is allocated, it is marked as unavailable for further allocation. Finally, the program displays each file's assigned block and the resulting fragmentation.

ALGORITHM

Step 1: Start
Step 2: Initialize arrays for blocks (b), files (f), block flag (bf), file-to-block map (ff), and fragmentation (frag)
Step 3: Read number of blocks (nb) and number of files (nf)
Step 4: Input size of each block into array b[1..nb]
Step 5: Input size of each file into array f[1..nf]

Step 6: For each file i from 1 to nf
  Step 6.1: Set lowest = a large value (e.g., 9999)
  Step 6.2: For each block j from 1 to nb
    Step 6.2.1: If block j is not allocated (bf[j] != 1)
      Step 6.2.1.1: Calculate temp = b[j] – f[i]
      Step 6.2.1.2: If temp >= 0 and temp < lowest
        Step 6.2.1.2.1: Set lowest = temp
        Step 6.2.1.2.2: Assign ff[i] = j (block index for file i)
  Step 6.3: After scanning all blocks
    Step 6.3.1: Store fragmentation frag[i] = lowest
    Step 6.3.2: Mark chosen block as allocated: bf[ff[i]] = 1
Step 7: Display table of File No, File Size, Block No, Block Size, and Fragmentation
Step 8: End

CONCLUSION: The program correctly implements the **Best Fit** memory allocation technique by assigning each file to the most appropriately sized memory block available. The result shows efficient utilization of memory with minimum fragmentation for each file. If no suitable block remains for a file, it is not allocated. This helps in understanding dynamic memory allocation strategies used in operating systems.

9. AIM: To write a java program to implement the concept of memory fragmentation.

DESCRIPTION: This Java program simulates memory allocation using **Fixed Partitioning**. The user enters total memory size, block size, and number of processes with their memory requirements. The total memory is divided into fixed-sized partitions (blocks). Each process is allocated a block if its required memory is less than or equal to the block size. The program displays whether each process is allocated, computes the internal fragmentation for each allocated block, and finally shows the total internal and external fragmentation.

ALGORITHM

Step 1: Start
Step 2: Input total memory size (ms) and block size (bs)
Step 3: Calculate number of blocks: nob = ms / bs
Step 4: Calculate external fragmentation: ef = ms - (nob * bs)
Step 5: Input number of processes (n)
Step 6: For i = 0 to n-1
    Input memory required for process i into array mp[i]
Step 7: Display number of blocks available
Step 8: Initialize total internal fragmentation (tif) = 0 and block pointer p = 0
Step 9: For i = 0 to n-1 and p < nob
    9.1. Print process number and memory requirement
    9.2 . If mp[i] > bs
        9.2.1 Mark as “Not Allocated”
    9.3 Else
        9.3.1 Allocate block → Print “YES” and (bs - mp[i])
        9.3.2 Add (bs - mp[i]) to total internal fragmentation (tif)
        9.3.3 p = p + 1
Step 10: If there are unallocated processes due to no free blocks
    10.1 Display that remaining processes cannot be accommodated
Step 11: Print total internal fragmentation (tif) and total external fragmentation (ef)
Step 12: End

CONCLUSION: The program successfully demonstrates **Fixed Partition Memory Allocation**. It partitions memory into fixed blocks and allocates them to processes if possible. Internal fragmentation is calculated for each allocated process, and total internal and external fragmentation values are displayed. This helps in understanding how fixed partitions can lead to wasted memory and the limitations of this allocation technique.

10. AIM: To implement the **First-In First-Out (FIFO) Page Replacement** algorithm in Java and compute the number of page *hits*, *faults*, and the *hit ratio* for a given page reference string and number of frames.

DESCRIPTION: This program simulates the **FIFO (First-In First-Out) page replacement** algorithm, a basic memory management technique used in operating systems, where pages are loaded into a fixed number of frames and if a referenced page is already present it is counted as a **hit**, otherwise it results in a **page fault**; when all frames are full and a new page must be loaded, the **oldest page in memory (the one that arrived first)** is replaced with the new one, and the program tracks and displays the frame contents over time and finally prints the total number of hits, the hit ratio, and the total faults.

ALGORITHM

Step 1: Start
Step 2: Read number of frames (frames)
Step 3: Read length of reference string (ref_len)
Step 4: Read reference string into array reference[0..ref_len-1]
Step 5: Initialize buffer[0..frames-1] = -1 (empty)
Step 6: Initialize pointer = 0, hit = 0, fault = 0
Step 7: For i from 0 to ref_len-1:
    7.1 . Set search = -1
    7.2 . For j from 0 to frames-1:
        7.2.1 If buffer[j] == reference[i]:
        7.2.2 search = j
        7.2.3 hit++ // Page already in memory
        7.2.4 Break
    7.3 . If search == -1 (page fault):
        7.3.1 buffer[pointer] = reference[i] // Replace at pointer
        7.3.2. fault++
        7.3.3 .pointer = pointer + 1
        7.3.4 If pointer == frames:
        7.3.5. pointer = 0 // Wrap around FIFO
    7.4 . Store current buffer state in mem_layout[i][j]
Step 8: Display frame contents over the reference sequence
Step 9: Print total hits, hit ratio (hit/ref_len), and total faults
Step 10: End

CONCLUSION: The program implements the **FIFO page replacement** algorithm, showing how pages are loaded into fixed memory frames and counting **hits** when a page is already in memory and **faults** when it must be loaded. It calculates a **hit ratio** to measure performance. FIFO is simple and easy to implement with low overhead, but because it always replaces the oldest page without considering usage patterns, it can perform poorly and may even exhibit **Belady's anomaly**, where increasing the number of frames increases page faults.

11. AIM: To implement the **Least Recently Used (LRU)** page replacement algorithm in Java for managing page frames and counting the number of page faults based on recent page usage.

DESCRIPTION: This Java program simulates the **LRU page replacement** strategy, a common memory management technique used in operating systems to decide which page to replace when a new page must be loaded but all frames are full. LRU works on the principle that the page that has not been used for the longest time is the one least likely to be used soon, so that page is replaced first when needed. The program accepts the number of frames and the sequence of pages, loads pages into frames, and replaces pages using the LRU logic. It displays the frame contents after each insertion and finally prints the total number of page faults.

ALGORITHM

Step 1: Start
Step 2: Read number of frames (size) and number of pages (pages)
Step 3: Initialize arrays: frame[size], present[size], page[pages]
Step 4: Read the page reference string into page[]
Step 5: Set all frame entries to -1 (empty)
Step 6: Initialize page fault count pf = 0
Step 7: For each page index i from 0 to pages-1
  7.1. If frame contains page[i] (check returns >= 0) → no fault
  7.2 Else → page fault
    7.2.1. If frames not filled yet (i<size), load page directly
    7.2.2 . Else find the least recently used page using replace(i)
    7.2.3 Replace that frame with page[i]
    7.2.4 Increment pf
  7.3 Display current frame contents
Step 8: After all references, print total page faults (pf)
Step 9: End

CONCLUSION : The program successfully implements the **LRU page replacement** algorithm by loading pages into fixed frames and replacing the least recently used page when necessary. It accurately counts and displays page faults, helping to understand how LRU prioritizes recent usage to reduce page faults compared with simpler strategies like FIFO. LRU often performs better in practice because it uses recent access history to approximate future use, though it can require more tracking overhead.

12. AIM: To write a java program to implement the concept of optimal page replacement.

DESCRIPTION: This Java program simulates the **Optimal Page Replacement Algorithm** for efficient memory management. It takes the number of memory frames, the length of the reference string, and the reference string as input, and processes each page sequentially. If a page is already in memory, it is counted as a hit; if not, a page fault occurs. When memory is full, the program replaces the page that will not be used for the longest time in the future, following the Optimal strategy. The program displays the memory layout at each step along with the total hits, hit ratio, and page faults, demonstrating how the algorithm minimizes page faults and manages memory effectively.

ALGORITHM:

Step 1: Start

Step 2: Initialize variables and arrays

- Step 2.1: Declare variables: frames, pointer = 0, hit = 0, fault = 0, ref_len, isFull = false.
- Step 2.2: Declare arrays:
  - 2.2.1 buffer[frames] → stores pages currently in memory. Initialize all elements to -1.
  - 2.2.2 reference[ref_len] → stores the reference string entered by the user.
  - 2.2.3 mem_layout[ref_len][frames] → stores memory layout at each step.

Step 3: Input

- Step 3.1: Input the number of frames (frames).
- Step 3.2: Input the length of the reference string (ref_len).
- Step 3.3: Input the reference string (pages requested) into reference[].

Step 4: Process each page in the reference string

- Step 4.1: For each page reference[i] in the reference string:
  - Step 4.1.1: Set search = -1.
  - Step 4.1.2: Check if the page is already in memory (buffer[]):
    - Step 4.1.2.1: If yes, set search = j (index of page in memory) and increment hit.
  - Step 4.1.3: If search == -1 (page fault):
    - Step 4.1.3.1: If memory is not full (isFull = false):
      - Place the page in buffer[pointer].
      - Increment pointer.
      - If pointer == frames, set isFull = true and reset pointer = 0.
    - Step 4.1.3.2: If memory is full (isFull = true), use **Optimal Replacement**:
      - Step 4.1.3.2.1: For each page in buffer[], find its **next occurrence** in reference[i+1 … ref_len].
      - Step 4.1.3.2.2: If a page is not found in the future, assign a large value (e.g., 200) to indicate it won’t be used.

Step 4.1.3.2.3: Select the page in memory with the **farthest future use** and replace it with the current page.

Step 4.1.3.3: Increment fault.

Step 4.1.4: Update mem_layout[i][j] with the current contents of buffer[].

Step 5: Display memory layout

Step 5.1: Print mem_layout[][] row-wise to show how pages were loaded and replaced in memory.

Step 6: Calculate and display results

Step 6.1: Display total number of hits (hit).

Step 6.2: Display hit ratio (hit ÷ ref_len).

Step 6.3: Display total number of page faults (fault).

Step 7: End

CONCLUSION: The program demonstrates the Optimal Page Replacement Algorithm for efficient memory management. It shows how pages are loaded and replaced in memory to minimize page faults. From the output, we can see the number of hits and faults, and the memory is managed optimally to reduce unnecessary replacements.

13. **AIM:** To implement the **SCAN Disk Scheduling Algorithm** in Java and calculate the total head movement for a given set of disk I/O requests.

DESCRIPTION: The **SCAN Disk Scheduling Algorithm**, also known as the **Elevator Algorithm**, is used in operating systems to schedule disk I/O requests efficiently. In this algorithm, the disk arm starts from its current position and moves in a particular direction (towards one end of the disk). It services all requests in that direction until it reaches the end of the disk, then reverses direction and continues servicing requests.

This Java program takes the number of disk I/O requests, the initial head position, and the list of requests as input. It then sorts the requests in ascending order and calculates the total head movement based on the SCAN algorithm. The program simulates the movement of the disk head, ensuring that all requests are serviced while minimizing unnecessary head movement. Finally, it outputs the total head movement.

ALGORITHM

Step 1: Start
Step 2: Input Data
- Step 2.1: Input the number of processes/requests (a).
- Step 2.2: Input the initial position of the disk head (h).
- Step 2.3: Input the disk I/O requests into an array ar[].
- Step 2.4: Set ar[0] = 0 to include the start of the disk.

Step 3: Sort Requests
- Step 3.1: Sort all elements in ar[] in **ascending order**.

Step 4: Calculate Total Head Movement
- Step 4.1: Initialize hm = 0 (total head movement).
- Step 4.2: Traverse through the sorted request array:
    - Step 4.2.1: For each pair of consecutive requests, calculate the movement: x = ar[c+1] - ar[c].
    - Step 4.2.2: Add x to hm.
    - Step 4.2.3: If the current request equals the initial head position (ar[c] == h):
        4.2.3.1 Subtract the last added movement (hm = hm - x).
        4.2.3.2 Add the movement from head to next request (hm = hm + ar[c+1]).
- Step 4.3: Repeat until all requests are serviced in SCAN order.

Step 5: Output Results
- Step 5.1: Display the total head movement (hm).

Step 6: End

CONCLUSION: The program successfully implements the **SCAN Disk Scheduling Algorithm**. It demonstrates how the disk head moves in one direction, services all requests, and then reverses direction to continue processing, similar to an elevator. The total head movement is calculated efficiently, showing the advantage of SCAN over simpler algorithms like FCFS in reducing unnecessary disk arm movement. This program helps in understanding how operating systems manage disk I/O requests and optimize disk performance.

14. AIM: To write a java program to implement the concept of CLOOK disk scheduling.

DESCRIPTION : This program implements the **C-LOOK disk scheduling algorithm**, which efficiently manages disk head movement to service requests. It takes the number of requests, initial head position, and request locations as input, sorts the requests, and moves the head in one direction toward the largest request. After reaching the farthest request, the head jumps to the smallest request and continues in the same direction, servicing all remaining requests. The program calculates and displays the **total head movement**, showing how C-LOOK reduces unnecessary movement and improves disk efficiency.

ALGORITHM

Step 1: Start
Step 2: Input Data
- **Step 2.1:** Input the number of requests/processes a.
- **Step 2.2:** Input the initial head position h.
- **Step 2.3:** Input the disk I/O requests into an array ar[].
- **Step 2.4:** (Optional) Include the initial head position in ar[] for sorting purposes.

Step 3: Sort Requests
- **Step 3.1:** Sort all elements in ar[] in ascending order.
- **Step 3.2:** Identify the position/index of the initial head h in the sorted array.

Step 4: Calculate Total Head Movement
- **Step 4.1:** Initialize hm = 0 (total head movement).
- **Step 4.2:** Move head from the initial position toward the higher end of the request array:
    - **Step 4.2.1:** For each consecutive request ar[c] to ar[c+1], calculate the movement: x = |ar[c+1] - ar[c]|.
    - **Step 4.2.2:** Add x to hm.
- **Step 4.3:** When the last request in this direction is reached, move the head **circularly** to the first request in the array:
    - **Step 4.3.1:** Calculate movement from last request to first request: x = |ar[last] - ar[0]|.
    - **Step 4.3.2:** Add x to hm.
- **Step 4.4:** Continue moving the head to service the remaining requests in order, adding each movement to hm until all requests are serviced.

Step 5: Output Results
- **Step 5.1:** Display the total head movement hm.

Step 6: End.

CONCLUSION: The C-LOOK disk scheduling algorithm efficiently minimizes total head movement by servicing requests in one direction and then jumping circularly to the first request. This implementation accurately calculates the total head movements, demonstrating improved disk I/O performance and reduced seek time.

15. AIM: To calculate the **total head movement** for servicing disk I/O requests using the **First-Come, First-Served (FCFS) disk scheduling algorithm** in Java.

DESCRIPTION: The FCFS disk scheduling algorithm services disk I/O requests in the **order they arrive**. It is the simplest scheduling algorithm but may not always be the most efficient in terms of total head movement.
In this program:

- The user inputs the number of disk cylinders and their request sequence.
- The initial head position is provided.
- The program calculates the distance the head moves to service each request sequentially.
- The total head movement (sum of all individual movements) is then displayed.

This helps understand how the head travels across the disk and the efficiency of FCFS scheduling.

ALGORITHM

Step 1: Start
Step 2: Input Data

- **Step 2.1:** Input the number of cylinders `n`.
- **Step 2.2:** Input the cylinder numbers into an array `c[]`.
- **Step 2.3:** Input the initial head position `hs`.

Step 3: Calculate Head Movement

- **Step 3.1:** Initialize `total = 0` (total head movement).
- **Step 3.2:** Initialize `flag = hs` (current head position).
- **Step 3.3:** For each cylinder `c[i]` in the array:
    - **Step 3.3.1:** Calculate head movement `hm[i] = |c[i] - flag|`.
    - **Step 3.3.2:** Add `hm[i]` to `total`.
    - **Step 3.3.3:** Update `flag = c[i]` (move head to current request).

Step 4: Output Results

- **Step 4.1:** Display the total head movement `total`.

Step 5: End

CONCLUSION: The FCFS disk scheduling algorithm is simple and easy to implement, servicing requests in the order they arrive. This program accurately calculates the total head movement for the given request sequence. Although FCFS is straightforward, it may lead to higher seek times compared to other algorithms like SSTF, SCAN, or C-LOOK, especially when requests are widely scattered.