1. **FCFS Code:-**

**package** java\_programmes;

**import** java.util.Scanner;

**public** **class** FCFS {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

// Input

System.***out***.print("Enter number of processes: ");

**int** n = sc.nextInt();

**int**[] arrival = **new** **int**[n], burst = **new** **int**[n], waiting = **new** **int**[n], turnaround = **new** **int**[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival and burst time for process " + (i + 1) + ": ");

arrival[i] = sc.nextInt();

burst[i] = sc.nextInt();

}

**int** currentTime = 0;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

System.***out***.println("\nGantt Chart:");

**for** (**int** i = 0; i < n; i++) {

currentTime = Math.*max*(currentTime, arrival[i]) + burst[i];

turnaround[i] = currentTime - arrival[i];

waiting[i] = turnaround[i] - burst[i];

totalWaitingTime += waiting[i];

totalTurnaroundTime += turnaround[i];

System.***out***.print("| P" + (i + 1) + " ");

}

System.***out***.println("|");

// Output

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

sc.close();

}

}

1. **SJF Code:-**

**package** java\_spos;

**import** java.util.Scanner;

**import** java.util.Arrays;

**public** **class** Sjf {

**static** **class** Process {

**int** id, arrival, burst, waiting, turnaround;

**boolean** isCompleted = **false**;

Process(**int** id, **int** arrival, **int** burst) {

**this**.id = id;

**this**.arrival = arrival;

**this**.burst = burst;

}

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

// Input

System.***out***.print("Enter number of processes: ");

**int** n = sc.nextInt();

Process[] processes = **new** Process[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival and burst time for process " + (i + 1) + ": ");

processes[i] = **new** Process(i + 1, sc.nextInt(), sc.nextInt());

}

**int** currentTime = 0, completed = 0;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

System.***out***.println("\nGantt Chart SJF:");

**while** (completed < n) {

**int** minBurst = Integer.***MAX\_VALUE***;

Process shortestJob = **null**;

// Select the shortest job that has arrived

**for** (Process p : processes) {

**if** (!p.isCompleted && p.arrival <= currentTime && p.burst < minBurst) {

minBurst = p.burst;

shortestJob = p;

}

}

// If no job has arrived, move the current time forward

**if** (shortestJob == **null**) {

currentTime++;

**continue**;

}

currentTime += shortestJob.burst;

shortestJob.turnaround = currentTime - shortestJob.arrival;

shortestJob.waiting = shortestJob.turnaround - shortestJob.burst;

shortestJob.isCompleted = **true**;

completed++;

totalWaitingTime += shortestJob.waiting;

totalTurnaroundTime += shortestJob.turnaround;

System.***out***.print("| P" + shortestJob.id + " ");

}

System.***out***.println("|");

// Output

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

sc.close();

}

}

1. **SJF PREMP Code:-**

**package** java\_spos;

**import** java.util.Scanner;

**import** java.util.ArrayList;

**public** **class** SJFPreemptive {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter number of processes: ");

**int** n = sc.nextInt();

**int**[] arrival = **new** **int**[n];

**int**[] burst = **new** **int**[n];

**int**[] remaining = **new** **int**[n];

**int**[] waiting = **new** **int**[n];

**int**[] turnaround = **new** **int**[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

arrival[i] = sc.nextInt();

burst[i] = sc.nextInt();

remaining[i] = burst[i];

}

ArrayList<Integer> ganttChart = **new** ArrayList<>();

**int** currentTime = 0, completed = 0, minIndex = -1;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

**while** (completed < n) {

minIndex = -1;

**int** minBurst = Integer.***MAX\_VALUE***;

**for** (**int** i = 0; i < n; i++) {

**if** (arrival[i] <= currentTime && remaining[i] > 0 && remaining[i] < minBurst) {

minBurst = remaining[i];

minIndex = i;

}

}

**if** (minIndex == -1) {

currentTime++;

**continue**;

}

ganttChart.add(minIndex + 1);

remaining[minIndex]--;

currentTime++;

**if** (remaining[minIndex] == 0) {

completed++;

turnaround[minIndex] = currentTime - arrival[minIndex];

waiting[minIndex] = turnaround[minIndex] - burst[minIndex];

totalWaitingTime += waiting[minIndex];

totalTurnaroundTime += turnaround[minIndex];

}

}

System.***out***.print("Gantt Chart: ");

**for** (**int** pid : ganttChart) System.***out***.print("| P" + pid + " ");

System.***out***.println("|");

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

sc.close();

}

}

4)**SJF NON-PREMP Code:-**

**package** java\_spos;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Scanner;

**public** **class** SJFNonPreemptive {

**static** **class** Process {

**int** id, arrivalTime, burstTime, waitingTime, turnaroundTime, completionTime;

Process(**int** id, **int** arrivalTime, **int** burstTime) {

**this**.id = id;

**this**.arrivalTime = arrivalTime;

**this**.burstTime = burstTime;

}

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter the number of processes: ");

**int** n = sc.nextInt();

Process[] processes = **new** Process[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

**int** arrivalTime = sc.nextInt();

**int** burstTime = sc.nextInt();

processes[i] = **new** Process(i + 1, arrivalTime, burstTime);

}

// Sort by arrival time, and by burst time if arrival times are equal

Arrays.*sort*(processes, Comparator.*comparingInt*((Process p) -> p.arrivalTime).thenComparingInt(p -> p.burstTime));

**int** currentTime = 0;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

StringBuilder ganttChart = **new** StringBuilder();

**for** (**int** i = 0; i < n; i++) {

Process process = processes[i];

// Wait for the process to arrive if needed

**if** (currentTime < process.arrivalTime) {

currentTime = process.arrivalTime;

}

// Calculate completion, turnaround, and waiting times

process.completionTime = currentTime + process.burstTime;

process.turnaroundTime = process.completionTime - process.arrivalTime;

process.waitingTime = process.turnaroundTime - process.burstTime;

// Update total waiting and turnaround times

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

// Append to Gantt chart

ganttChart.append("P").append(process.id).append(" ");

// Move current time forward

currentTime = process.completionTime;

}

// Display results

System.***out***.println("\nProcess\tArrival\tBurst\tWaiting\tTurnaround");

**for** (Process process : processes) {

System.***out***.printf("P%d\t%d\t%d\t%d\t%d\n", process.id, process.arrivalTime, process.burstTime, process.waitingTime, process.turnaroundTime);

}

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

System.***out***.println("Gantt Chart: " + ganttChart);

sc.close();

}

}

1. **Priority (Premptive) Code:-**

**package** java\_spos;

**import** java.util.ArrayList;

**import** java.util.Scanner;

**public** **class** PriorityPreemptive {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter number of processes: ");

**int** n = sc.nextInt();

**int**[] arrival = **new** **int**[n];

**int**[] burst = **new** **int**[n];

**int**[] remaining = **new** **int**[n];

**int**[] priority = **new** **int**[n];

**int**[] waiting = **new** **int**[n];

**int**[] turnaround = **new** **int**[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival time, burst time, and priority for process " + (i + 1) + ": ");

arrival[i] = sc.nextInt();

burst[i] = sc.nextInt();

priority[i] = sc.nextInt();

remaining[i] = burst[i];

}

ArrayList<Integer> ganttChart = **new** ArrayList<>();

**int** currentTime = 0, completed = 0, minIndex = -1;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

**while** (completed < n) {

minIndex = -1;

**int** highestPriority = Integer.***MAX\_VALUE***;

**for** (**int** i = 0; i < n; i++) {

**if** (arrival[i] <= currentTime && remaining[i] > 0 && priority[i] < highestPriority) {

highestPriority = priority[i];

minIndex = i;

}

}

**if** (minIndex == -1) {

currentTime++;

**continue**;

}

ganttChart.add(minIndex + 1);

remaining[minIndex]--;

currentTime++;

**if** (remaining[minIndex] == 0) {

completed++;

turnaround[minIndex] = currentTime - arrival[minIndex];

waiting[minIndex] = turnaround[minIndex] - burst[minIndex];

totalWaitingTime += waiting[minIndex];

totalTurnaroundTime += turnaround[minIndex];

}

}

System.***out***.print("Gantt Chart: ");

**for** (**int** pid : ganttChart) System.***out***.print("P" + pid + " ");

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

sc.close();

}

}

1. **Priority (Non- Premptive) Code:-**

**package** java\_spos;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Scanner;

**public** **class** PriorityNonPreemptive {

**static** **class** Process {

**int** id, arrivalTime, burstTime, priority, waitingTime, turnaroundTime, completionTime;

Process(**int** id, **int** arrivalTime, **int** burstTime, **int** priority) {

**this**.id = id;

**this**.arrivalTime = arrivalTime;

**this**.burstTime = burstTime;

**this**.priority = priority;

}

}

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter the number of processes: ");

**int** n = sc.nextInt();

Process[] processes = **new** Process[n];

**for** (**int** i = 0; i < n; i++) {

System.***out***.print("Enter arrival time, burst time, and priority for process " + (i + 1) + ": ");

**int** arrivalTime = sc.nextInt();

**int** burstTime = sc.nextInt();

**int** priority = sc.nextInt();

processes[i] = **new** Process(i + 1, arrivalTime, burstTime, priority);

}

// Sort by arrival time first, and by priority if arrival times are equal

Arrays.*sort*(processes, Comparator.*comparingInt*((Process p) -> p.arrivalTime)

.thenComparingInt(p -> p.priority));

**int** currentTime = 0;

**double** totalWaitingTime = 0, totalTurnaroundTime = 0;

StringBuilder ganttChart = **new** StringBuilder();

**for** (**int** i = 0; i < n; i++) {

Process process = processes[i];

// Wait for the process to arrive if needed

**if** (currentTime < process.arrivalTime) {

currentTime = process.arrivalTime;

}

// Calculate completion, turnaround, and waiting times

process.completionTime = currentTime + process.burstTime;

process.turnaroundTime = process.completionTime - process.arrivalTime;

process.waitingTime = process.turnaroundTime - process.burstTime;

// Update total waiting and turnaround times

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

// Append to Gantt chart

ganttChart.append("| P").append(process.id).append(" ");

System.***out***.println("|");

// Move current time forward

currentTime = process.completionTime;

}

// Display results

System.***out***.println("\nProcess\tArrival\tBurst\tPriority\tWaiting\tTurnaround");

**for** (Process process : processes) {

System.***out***.printf("P%d\t%d\t%d\t%d\t\t%d\t%d\n", process.id, process.arrivalTime, process.burstTime, process.priority, process.waitingTime, process.turnaroundTime);

}

System.***out***.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.***out***.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

System.***out***.println("Gantt Chart: " + ganttChart);

sc.close();

}

}

1. **Round Robin Code:-**

package java\_spos;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

public class RoundRobin {

static class Process {

int id, arrivalTime, burstTime, remainingTime, waitingTime, turnaroundTime, completionTime;

Process(int id, int arrivalTime, int burstTime) {

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.remainingTime = burstTime;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of processes: ");

int n = sc.nextInt();

System.out.print("Enter the time quantum: ");

int quantum = sc.nextInt();

Process[] processes = new Process[n];

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

System.out.print("Enter arrival time and burst time for process " + (i + 1) + ": ");

int arrivalTime = sc.nextInt();

int burstTime = sc.nextInt();

processes[i] = new Process(i + 1, arrivalTime, burstTime);

}

// Queue for round robin scheduling

Queue<Process> queue = new LinkedList<>();

int currentTime = 0, completedProcesses = 0;

double totalWaitingTime = 0, totalTurnaroundTime = 0;

StringBuilder ganttChart = new StringBuilder();

// Add processes to the queue when they arrive

while (completedProcesses < n) {

// Add any new arrivals to the queue

for (Process process : processes) {

if (process.arrivalTime <= currentTime && process.remainingTime > 0 && !queue.contains(process)) {

queue.add(process);

}

}

if (!queue.isEmpty()) {

Process process = queue.poll();

// Append to Gantt chart

ganttChart.append("| P").append(process.id).append(" ");

System.out.println("|");

// Execute the process for a time quantum or its remaining time

int executionTime = Math.min(process.remainingTime, quantum);

currentTime += executionTime;

process.remainingTime -= executionTime;

// If process is completed

if (process.remainingTime == 0) {

completedProcesses++;

process.completionTime = currentTime;

process.turnaroundTime = process.completionTime - process.arrivalTime;

process.waitingTime = process.turnaroundTime - process.burstTime;

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

}

// Re-add process to the queue if it's not finished

if (process.remainingTime > 0) {

queue.add(process);

}

} else {

// If no process is in the queue, increment the time

currentTime++;

}

}

// Display results

System.out.println("\nProcess\tArrival\tBurst\tWaiting\tTurnaround");

for (Process process : processes) {

System.out.printf("P%d\t%d\t%d\t%d\t%d\n", process.id, process.arrivalTime, process.burstTime, process.waitingTime, process.turnaroundTime);

}

System.out.printf("\nAverage Waiting Time: %.2f\n", totalWaitingTime / n);

System.out.printf("Average Turnaround Time: %.2f\n", totalTurnaroundTime / n);

System.out.println("Gantt Chart: " + ganttChart);

sc.close();

}

}

1. **FIFO Code :-**

**package** java\_spos;

**import** java.util.\*;

**public** **class** FIFOPageReplacement {

// Function to implement FIFO Page Replacement

**public** **static** **void** fifoPageReplacement(**int**[] referenceString, **int** frameCount) {

**int**[] frames = **new** **int**[frameCount];

Arrays.*fill*(frames, -1); // Initialize frames to -1 (empty slots)

**int** pageFaults = 0;

**int** pointer = 0; // Pointer to keep track of the oldest page in the frames

// Loop through the reference string

**for** (**int** i = 0; i < referenceString.length; i++) {

**int** currentPage = referenceString[i];

**boolean** pageFound = **false**;

// Check if the page is already in one of the frames

**for** (**int** j = 0; j < frameCount; j++) {

**if** (frames[j] == currentPage) {

pageFound = **true**;

**break**;

}

}

// If page is not found, it's a page fault

**if** (!pageFound) {

pageFaults++;

// If there's an empty frame, insert the page in it

**if** (frames[pointer] == -1) {

frames[pointer] = currentPage;

} **else** {

// If frames are full, replace the page at the pointer (FIFO)

frames[pointer] = currentPage;

}

// Move the pointer to the next frame

pointer = (pointer + 1) % frameCount;

}

// Print the current state of frames

System.***out***.print("Reference: " + currentPage + " | ");

**for** (**int** j = 0; j < frameCount; j++) {

System.***out***.print(frames[j] + " ");

}

System.***out***.println();

}

// Print the total number of page faults

System.***out***.println("Total Page Faults: " + pageFaults);

}

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

// Taking input for the reference string

System.***out***.print("Enter the number of pages in the reference string: ");

**int** n = scanner.nextInt();

**int**[] referenceString = **new** **int**[n];

System.***out***.println("Enter the reference string (space-separated): ");

**for** (**int** i = 0; i < n; i++) {

referenceString[i] = scanner.nextInt();

}

// Taking input for the number of frames

System.***out***.print("Enter the number of frames: ");

**int** frameCount = scanner.nextInt();

// Call the FIFO page replacement function

*fifoPageReplacement*(referenceString, frameCount);

}

}

1. **Optimal Page Replacement Code:-**

**package** java\_spos;

**import** java.util.\*;

**public** **class** OptimalPageReplacement {

// Function to implement Optimal Page Replacement

**public** **static** **void** optimalPageReplacement(**int**[] referenceString, **int** frameCount) {

**int**[] frames = **new** **int**[frameCount];

Arrays.*fill*(frames, -1); // Initializing frames with -1, representing empty slots

**int** pageFaults = 0;

// Loop through the reference string

**for** (**int** i = 0; i < referenceString.length; i++) {

**int** currentPage = referenceString[i];

**boolean** pageFound = **false**;

// Check if the page is already in the frame

**for** (**int** j = 0; j < frameCount; j++) {

**if** (frames[j] == currentPage) {

pageFound = **true**;

**break**;

}

}

// If the page is not found in the frame, it's a page fault

**if** (!pageFound) {

pageFaults++;

// If there's an empty frame, put the page in it

**boolean** pageInserted = **false**;

**for** (**int** j = 0; j < frameCount; j++) {

**if** (frames[j] == -1) {

frames[j] = currentPage;

pageInserted = **true**;

**break**;

}

}

// If all frames are full, we need to replace a page

**if** (!pageInserted) {

**int** farthestIndex = -1;

**int** pageToReplace = -1;

// Find the page that is used farthest in the future

**for** (**int** j = 0; j < frameCount; j++) {

**int** nextUse = -1;

**for** (**int** k = i + 1; k < referenceString.length; k++) {

**if** (frames[j] == referenceString[k]) {

nextUse = k;

**break**;

}

}

// If a page is not used again in the future, replace it

**if** (nextUse == -1) {

pageToReplace = frames[j];

farthestIndex = j;

**break**;

}

// Find the page with the farthest next use

**if** (nextUse > farthestIndex) {

farthestIndex = nextUse;

pageToReplace = frames[j];

}

}

// Replace the page with the one that has the farthest next use

**for** (**int** j = 0; j < frameCount; j++) {

**if** (frames[j] == pageToReplace) {

frames[j] = currentPage;

**break**;

}

}

}

}

// Print the current state of frames

System.***out***.print("Reference: " + currentPage + " | ");

**for** (**int** j = 0; j < frameCount; j++) {

System.***out***.print(frames[j] + " ");

}

System.***out***.println();

}

// Print the total number of page faults

System.***out***.println("Total Page Faults: " + pageFaults);

}

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

// Taking input for the reference string

System.***out***.print("Enter the number of pages in the reference string: ");

**int** n = scanner.nextInt();

**int**[] referenceString = **new** **int**[n];

System.***out***.println("Enter the reference string (space-separated): ");

**for** (**int** i = 0; i < n; i++) {

referenceString[i] = scanner.nextInt();

}

// Taking input for the number of frames

System.***out***.print("Enter the number of frames: ");

**int** frameCount = scanner.nextInt();

// Call the optimal page replacement function

*optimalPageReplacement*(referenceString, frameCount);

}

}

1. **LRU Code:-**

**package** java\_spos;

**import** java.util.ArrayList;

**import** java.util.Scanner;

**public** **class** LRUPageReplacement {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter the number of frames: ");

**int** frames = sc.nextInt();

System.***out***.print("Enter the number of page references: ");

**int** n = sc.nextInt();

**int**[] pages = **new** **int**[n];

System.***out***.print("Enter the page references: ");

**for** (**int** i = 0; i < n; i++) {

pages[i] = sc.nextInt();

}

ArrayList<Integer> memory = **new** ArrayList<>(frames);

**int** pageFaults = 0;

**for** (**int** i = 0; i < n; i++) {

**int** page = pages[i];

// Check if page is already in memory

**if** (memory.contains(page)) {

// If the page is already present, move it to the end to mark it as most recently used

memory.remove((Integer) page);

memory.add(page);

} **else** {

// Page fault occurs

pageFaults++;

// If memory is full, remove the least recently used page

**if** (memory.size() == frames) {

memory.remove(0);

}

// Add the current page to memory

memory.add(page);

}

// Display current memory content

System.***out***.print("Memory: ");

**for** (**int** m : memory) {

System.***out***.print(m + " ");

}

System.***out***.println();

}

System.***out***.println("\nTotal Page Faults: " + pageFaults);

sc.close();

}

}