***FCFS problem:***

import java.util.Scanner;

class Process {

private static int count = 0; // Static variable to auto-increment process ID

private int pid;

private String processName;

private int burstTime;

private int waitingTime;

private int turnaroundTime;

public Process(String processName, int burstTime) {

this.pid = ++count;

this.processName = processName;

this.burstTime = burstTime;

this.waitingTime = 0;

this.turnaroundTime = 0;

}

public int getPid() {

return pid;

}

public String getProcessName() {

return processName;

}

public int getBurstTime() {

return burstTime;

}

public int getWaitingTime() {

return waitingTime;

}

public void setWaitingTime(int waitingTime) {

this.waitingTime = waitingTime;

}

public int getTurnaroundTime() {

return turnaroundTime;

}

public void setTurnaroundTime(int turnaroundTime) {

this.turnaroundTime = turnaroundTime;

}

}

class FCFS\_Scheduling {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

Process[] processes = new Process[n];

// Accept process name and burst time for each process

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

System.out.print("Enter process name for process " + (i + 1) + ": ");

String processName = scanner.next();

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

int burstTime = scanner.nextInt();

processes[i] = new Process(processName, burstTime);

}

scanner.close();

// Calculate waiting time and turnaround time for each process

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

for (int i = 1; i < n; i++) {

processes[i].setWaitingTime(processes[i - 1].getBurstTime() + processes[i - 1].getWaitingTime());

processes[i].setTurnaroundTime(processes[i].getWaitingTime() + processes[i].getBurstTime());

totalWaitingTime += processes[i].getWaitingTime();

totalTurnaroundTime += processes[i].getTurnaroundTime();

}

// Calculate average waiting time and average turnaround time

double averageWaitingTime = (double) totalWaitingTime / n;

double averageTurnaroundTime = (double) totalTurnaroundTime / n;

// Display process details, average waiting time, and average turnaround time

System.out.println("\nProcess\tProcess Name\tBurst Time\tWaiting Time\tTurnaround Time");

for (Process process : processes) {

System.out.println(process.getPid() + " \t" + process.getProcessName() + " \t\t\t" +

process.getBurstTime() + " \t\t\t" + process.getWaitingTime() + " \t\t\t" +

process.getTurnaroundTime());

}

System.out.println("\nAverage Waiting Time: " + averageWaitingTime);

System.out.println("Average Turnaround Time: " + averageTurnaroundTime);

}

}

**Week -4:  
1.**

import java.util.Arrays;

class FCFS\_Scheduling {

static class Process {

int id;

int burstTime;

public Process(int id, int burstTime) {

this.id = id;

this.burstTime = burstTime;

}

}

// Function to calculate waiting time and turnaround time for a given sequence of process IDs

static void calculateTimes(Process[] processes, int[] sequence, int[] waitingTime, int[] turnaroundTime) {

waitingTime[0] = 0;

turnaroundTime[0] = processes[sequence[0]].burstTime;

for (int i = 1; i < sequence.length; i++) {

waitingTime[i] = waitingTime[i - 1] + processes[sequence[i - 1]].burstTime;

turnaroundTime[i] = waitingTime[i] + processes[sequence[i]].burstTime;

}

}

// Function to calculate average waiting time

static double calculateAverageWaitingTime(int[] waitingTime) {

double sum = 0;

for (int time : waitingTime) {

sum += time;

}

return sum / waitingTime.length;

}

// Function to calculate average turnaround time

static double calculateAverageTurnaroundTime(int[] turnaroundTime) {

double sum = 0;

for (int time : turnaroundTime) {

sum += time;

}

return sum / turnaroundTime.length;

}

// Function to generate all permutations of process IDs iteratively

static void generatePermutations(Process[] processes, int[] sequence, double[][] avgTimes) {

int n = sequence.length;

int[] c = new int[n];

// Fill the c array with zeros

Arrays.fill(c, 0);

// Initialize the index variable

int i = 0;

int k = 0;

// Generate permutations iteratively

while (i < n) {

if (c[i] < i) {

if (i % 2 == 0) {

int temp = sequence[0];

sequence[0] = sequence[i];

sequence[i] = temp;

} else {

int temp = sequence[c[i]];

sequence[c[i]] = sequence[i];

sequence[i] = temp;

}

// Calculate average times for the current sequence

int[] waitingTime = new int[n];

int[] turnaroundTime = new int[n];

calculateTimes(processes, sequence, waitingTime, turnaroundTime);

avgTimes[k++] = new double[]{calculateAverageWaitingTime(waitingTime),

calculateAverageTurnaroundTime(turnaroundTime)};

// Print the sequence of processes for the current permutation

System.out.print("Sequence: ");

for (int id : sequence) {

System.out.print("P" + (id + 1) + " ");

}

System.out.println("\nAverage Waiting Time: " + avgTimes[k - 1][0]);

System.out.println("Average Turnaround Time: " + avgTimes[k - 1][1]);

System.out.println();

c[i]++;

i = 0;

} else {

c[i] = 0;

i++;

}

}

}

public static void main(String[] args) {

Process[] processes = {new Process(0, 8), new Process(1, 3), new Process(2, 5), new Process(3, 3)};

int n = processes.length;

// Create an array to store process IDs in sequence

int[] sequence = new int[n];

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

sequence[i] = i;

}

// Calculate the number of permutations

int numPermutations = factorial(n);

// Create 2D array to store average waiting time and average turnaround time for each permutation

double[][] avgTimes = new double[numPermutations][2];

// Generate all permutations and calculate average times for each

generatePermutations(processes, sequence, avgTimes);

}

// Function to calculate factorial of a number

static int factorial(int n) {

if (n == 0)

return 1;

return n \* factorial(n - 1);

}

}

**2.**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

/ class FCFS\_Scheduling {

static class Process {

String name;

int arrivalTime;

int burstTime;

public Process(String name, int arrivalTime, int burstTime) {

this.name = name;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

}

}

public static void main(String[] args) {

List<Process> processes = new ArrayList<>();

processes.add(new Process("P1", 0, 4));

processes.add(new Process("P2", 1, 3));

processes.add(new Process("P3", 2, 1));

processes.add(new Process("P4", 3, 2));

processes.add(new Process("P5", 4, 5));

// Sort processes based on arrival time

Collections.sort(processes, (p1, p2) -> p1.arrivalTime - p2.arrivalTime);

int currentTime = 0;

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

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

for (Process process : processes) {

// Calculate waiting time

int waitingTime = Math.max(0, currentTime - process.arrivalTime);

totalWaitingTime += waitingTime;

// Calculate turnaround time

int turnaroundTime = waitingTime + process.burstTime;

totalTurnaroundTime += turnaroundTime;

// Update current time

currentTime += process.burstTime;

// Print process details

System.out.println(process.name + "\t\t" + process.arrivalTime + "\t\t\t" + process.burstTime + "\t\t\t" +

waitingTime + "\t\t\t\t" + turnaroundTime);

}

// Calculate and print average waiting time and average turnaround time

double avgWaitingTime = totalWaitingTime / processes.size();

double avgTurnaroundTime = totalTurnaroundTime / processes.size();

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

}

**3.**

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class SJF\_Scheduling {

static class Process {

String name;

int burstTime;

public Process(String name, int burstTime) {

this.name = name;

this.burstTime = burstTime;

}

}

public static void main(String[] args) {

List<Process> processes = new ArrayList<>();

processes.add(new Process("P1", 4));

processes.add(new Process("P2", 3));

processes.add(new Process("P3", 1));

processes.add(new Process("P4", 2));

processes.add(new Process("P5", 5));

// Sort processes based on burst time

Collections.sort(processes, (p1, p2) -> p1.burstTime - p2.burstTime);

int currentTime = 0;

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

System.out.println("Process\tBurst Time\tWaiting Time\tTurnaround Time");

for (Process process : processes) {

// Calculate waiting time

int waitingTime = Math.max(0, currentTime);

totalWaitingTime += waitingTime;

// Calculate turnaround time

int turnaroundTime = waitingTime + process.burstTime;

totalTurnaroundTime += turnaroundTime;

// Update current time

currentTime += process.burstTime;

// Print process details

System.out.println(process.name + "\t\t" + process.burstTime + "\t\t\t" +

waitingTime + "\t\t\t\t" + turnaroundTime);

}

// Calculate and print average waiting time and average turnaround time

double avgWaitingTime = totalWaitingTime / processes.size();

double avgTurnaroundTime = totalTurnaroundTime / processes.size();

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

}

4.Round Robbin:

import java.util.LinkedList;

import java.util.Queue;

import java.util.Scanner;

class RoundRobinScheduling {

static class Process {

int id;

int arrivalTime;

int burstTime;

int remainingBurstTime;

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

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.remainingBurstTime = burstTime;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

Process[] processes = new Process[n];

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

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

int arrivalTime = scanner.nextInt();

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

int burstTime = scanner.nextInt();

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

}

System.out.print("Enter time quantum for Round Robin: ");

int timeQuantum = scanner.nextInt();

scanner.close();

calculateTimes(processes, timeQuantum);

}

static void calculateTimes(Process[] processes, int timeQuantum) {

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

int currentTime = 0;

int[] waitingTime = new int[processes.length];

int[] turnaroundTime = new int[processes.length];

while (true) {

boolean done = true;

for (Process process : processes) {

if (process.arrivalTime <= currentTime && process.remainingBurstTime > 0) {

done = false;

if (process.remainingBurstTime > timeQuantum) {

currentTime += timeQuantum;

process.remainingBurstTime -= timeQuantum;

} else {

currentTime += process.remainingBurstTime;

waitingTime[process.id - 1] = currentTime - process.arrivalTime - process.burstTime;

process.remainingBurstTime = 0;

turnaroundTime[process.id - 1] = currentTime - process.arrivalTime;

}

}

}

if (done) {

break;

}

}

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

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

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

System.out.println("P" + processes[i].id + "\t\t" + waitingTime[i] + "\t\t\t" + turnaroundTime[i]);

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

}

double avgWaitingTime = totalWaitingTime / processes.length;

double avgTurnaroundTime = totalTurnaroundTime / processes.length;

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

}

***All programs:***

import java.util.\*;

class CPUScheduling {

static class Process {

int id;

int arrivalTime;

int burstTime;

int remainingBurstTime;

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

this.id = id;

this.arrivalTime = arrivalTime;

this.burstTime = burstTime;

this.remainingBurstTime = burstTime;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

Process[] processes = new Process[n];

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

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

int arrivalTime = scanner.nextInt();

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

int burstTime = scanner.nextInt();

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

}

while (true) {

System.out.println("\nSelect scheduling algorithm:");

System.out.println("1. First Come First Serve (FCFS)");

System.out.println("2. Shortest Job First (SJF)");

System.out.println("3. Preemptive Shortest Job First (Preemptive SJF)");

System.out.println("4. Round Robin (RR)");

System.out.println("5. Performance (Execute all algorithms)");

System.out.println("6. Exit");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.println("Executing FCFS scheduling algorithm...");

executeFCFS(processes);

break;

case 2:

System.out.println("Executing SJF scheduling algorithm...");

executeSJF(processes);

break;

case 3:

System.out.println("Executing Preemptive SJF scheduling algorithm...");

executePreemptiveSJF(processes);

break;

case 4:

System.out.print("Enter time quantum for Round Robin: ");

int timeQuantum = scanner.nextInt();

System.out.println("Executing Round Robin scheduling algorithm...");

executeRoundRobin(processes, timeQuantum);

break;

case 5:

System.out.println("Executing all scheduling algorithms...");

executeAllAlgorithms(processes);

break;

case 6:

System.out.println("Exiting...");

scanner.close();

System.exit(0);

default:

System.out.println("Invalid choice. Please choose again.");

}

}

}

static void executeFCFS(Process[] processes) {

// Sort processes based on arrival time

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

int currentTime = 0;

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

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

for (Process process : processes) {

int waitingTime = Math.max(0, currentTime - process.arrivalTime);

totalWaitingTime += waitingTime;

int turnaroundTime = waitingTime + process.burstTime; // Corrected calculation

totalTurnaroundTime += turnaroundTime;

currentTime = Math.max(currentTime, process.arrivalTime) + process.burstTime; // Corrected calculation

System.out.println("P" + process.id + "\t\t" + process.arrivalTime + "\t\t\t" + process.burstTime +

"\t\t\t" + waitingTime + "\t\t\t\t" + turnaroundTime);

}

double avgWaitingTime = totalWaitingTime / processes.length;

double avgTurnaroundTime = totalTurnaroundTime / processes.length;

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

static void executeSJF(Process[] processes) {

// Sort processes based on burst time

Arrays.sort(processes, Comparator.comparingInt(p -> p.burstTime));

int currentTime = 0;

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

System.out.println("Process\tBurst Time\tWaiting Time\tTurnaround Time");

for (Process process : processes) {

int waitingTime = Math.max(0, currentTime);

totalWaitingTime += waitingTime;

int turnaroundTime = waitingTime + process.burstTime;

totalTurnaroundTime += turnaroundTime;

currentTime += process.burstTime;

System.out.println("P" + process.id + "\t\t" + process.burstTime + "\t\t\t" +

waitingTime + "\t\t\t\t" + turnaroundTime);

}

double avgWaitingTime = totalWaitingTime / processes.length;

double avgTurnaroundTime = totalTurnaroundTime / processes.length;

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

static void executePreemptiveSJF(Process[] processes) {

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

int n = processes.length;

int currentTime = 0;

int completed = 0;

int[] waitingTime = new int[n];

int[] turnaroundTime = new int[n];

int[] remainingTime = new int[n]; // Track remaining burst time

boolean[] completedProcesses = new boolean[n]; // Track completed processes

// Initialize remaining time and completed processes arrays

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

remainingTime[i] = processes[i].burstTime;

completedProcesses[i] = false;

}

while (completed != n) {

int shortestBurst = Integer.MAX\_VALUE;

int selectedProcess = -1;

// Find the process with the shortest remaining burst time among the arrived processes

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

if (!completedProcesses[i] && processes[i].arrivalTime <= currentTime && remainingTime[i] < shortestBurst) {

selectedProcess = i;

shortestBurst = remainingTime[i];

}

}

// If no process is selected, move to the next time unit

if (selectedProcess == -1) {

currentTime++;

} else {

// Execute the selected process for 1 time unit

remainingTime[selectedProcess]--;

currentTime++;

// If the process finishes its burst time, update turnaround and waiting time

if (remainingTime[selectedProcess] == 0) {

completed++;

completedProcesses[selectedProcess] = true;

int completionTime = currentTime;

turnaroundTime[selectedProcess] = completionTime - processes[selectedProcess].arrivalTime;

waitingTime[selectedProcess] = turnaroundTime[selectedProcess] - processes[selectedProcess].burstTime;

}

}

}

// Print results

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

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

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

System.out.println("P" + processes[i].id + "\t\t" + waitingTime[i] + "\t\t\t" + turnaroundTime[i]);

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

}

double avgWaitingTime = totalWaitingTime / n;

double avgTurnaroundTime = totalTurnaroundTime / n;

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

static void executeRoundRobin(Process[] processes, int timeQuantum) {

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

int currentTime = 0;

int[] waitingTime = new int[processes.length];

int[] turnaroundTime = new int[processes.length];

for (Process process : processes) {

queue.add(process);

}

while (!queue.isEmpty()) {

Process currentProcess = queue.poll();

int executionTime = Math.min(currentProcess.remainingBurstTime, timeQuantum);

currentProcess.remainingBurstTime -= executionTime;

currentTime += executionTime;

// Update waiting time for processes in the queue

for (Process process : queue) {

waitingTime[process.id - 1] += executionTime; // Increment waiting time

}

if (currentProcess.remainingBurstTime > 0) {

queue.add(currentProcess);

} else {

// Process has completed its execution

int completionTime = currentTime;

turnaroundTime[currentProcess.id - 1] = completionTime - currentProcess.arrivalTime;

waitingTime[currentProcess.id - 1] += currentTime - currentProcess.arrivalTime - currentProcess.burstTime;

}

}

// Print results

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

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

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

System.out.println("P" + processes[i].id + "\t\t" + waitingTime[i] + "\t\t\t" + turnaroundTime[i]);

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

}

double avgWaitingTime = totalWaitingTime / processes.length;

double avgTurnaroundTime = totalTurnaroundTime / processes.length;

System.out.println("\nAverage Waiting Time: " + avgWaitingTime);

System.out.println("Average Turnaround Time: " + avgTurnaroundTime);

}

static void executeAllAlgorithms(Process[] processes) {

System.out.println("Executing FCFS scheduling algorithm...");

executeFCFS(processes);

System.out.println("\nExecuting SJF scheduling algorithm...");

executeSJF(processes);

System.out.println("\nExecuting Preemptive SJF scheduling algorithm...");

executePreemptiveSJF(processes);

Scanner timeScanner = new Scanner(System.in);

System.out.print("\nEnter time quantum for Round Robin: ");

int timeQuantum = timeScanner.nextInt();

System.out.println("Executing Round Robin scheduling algorithm...");

executeRoundRobin(processes, timeQuantum);

}

}