**Name: Atharva Salitri Division: CSAI B**

**Roll No.: 37 PRN: 12310120**

**Subject: OS Lab Assignment 4**

**Title: Implementation of CPU Scheduling Algorithms**

**1) FCFS**

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process

{

int id;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

};

bool compareArrival(Process a, Process b)

{

return a.arrivalTime < b.arrivalTime;

}

void displayGanttChart(vector<Process> &processes)

{

cout << "\nGantt Chart:\n ";

for (size\_t i = 0; i < processes.size(); i++)

{

cout << "+-------";

}

cout << "+\n";

cout << "|";

for (auto &p : processes)

{

cout << " P" << p.id << setw(5) << "|";

}

cout << "\n ";

for (size\_t i = 0; i < processes.size(); i++)

{

cout << "+-------";

}

cout << "+\n";

cout << "0";

for (auto &p : processes)

{

cout << setw(8) << p.completionTime;

}

cout << "\n";

}

int main()

{

int n;

cout << "Enter the number of processes: ";

cin >> n;

float avgWT = 0;

float avgTAT = 0;

vector<Process> processes(n);

// Input process details

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

{

processes[i].id = i + 1;

cout << "Enter arrival time and burst time for process P" << processes[i].id

<< ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime;

}

sort(processes.begin(), processes.end(), compareArrival);

// Calculate Completion, Turnaround, and Waiting times

int currentTime = 0;

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

{

if (currentTime < processes[i].arrivalTime)

{

currentTime = processes[i].arrivalTime; // Idle until the process arrives

}

processes[i].completionTime = currentTime + processes[i].burstTime;

currentTime = processes[i].completionTime;

processes[i].turnaroundTime = processes[i].completionTime -

processes[i].arrivalTime;

processes[i].waitingTime = processes[i].turnaroundTime -

processes[i].burstTime;

}

for (auto &p : processes)

{

avgTAT += p.turnaroundTime;

avgWT += p.waitingTime;

}

avgTAT /= n;

avgWT /= n;

cout << "\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n";

for (auto &p : processes)

{

cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime << "\t"

<< p.completionTime << "\t\t" << p.turnaroundTime << "\t\t" << p.waitingTime << "\n";

}

cout << "The average Waiting time : " << avgWT << "\n";

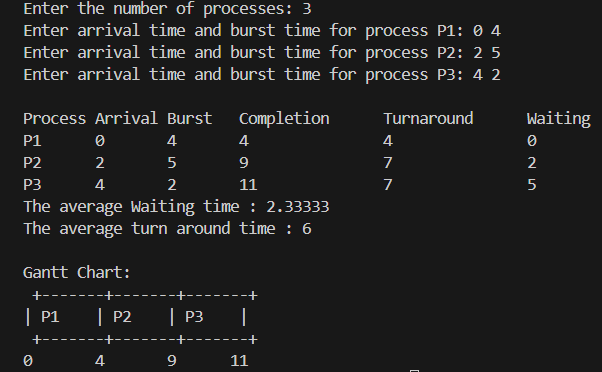
cout << "The average turn around time : " << avgTAT << "\n";

displayGanttChart(processes);

return 0;

}

**Output:**



**Q2) SJF**

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

#include <limits.h>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int completionTime;

int turnaroundTime;

int waitingTime;

bool isCompleted;

};

bool compareArrival(Process a, Process b) {

return a.arrivalTime < b.arrivalTime;

}

void displayGanttChart(vector<Process> &processes) {

cout << "\nGantt Chart:\n ";

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "|";

for (auto &p : processes) {

cout << " P" << p.id << setw(5) << "|";

}

cout << "\n ";

for (size\_t i = 0; i < processes.size(); i++) {

cout << "+-------";

}

cout << "+\n";

cout << "0";

for (auto &p : processes) {

cout << setw(8) << p.completionTime;

}

cout << "\n";

}

int main() {

int n;

cout << "Enter the number of processes: ";

cin >> n;

vector<Process> processes(n);

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

processes[i].id = i + 1;

cout << "Enter arrival time and burst time for process P" << processes[i].id

<< ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime;

processes[i].isCompleted = false;

}

sort(processes.begin(), processes.end(), compareArrival);

int completed = 0, currentTime = 0;

double totalWaitingTime = 0, totalTurnaroundTime = 0;

vector<Process> ganttChart;

while (completed < n) {

int idx = -1;

int minBurstTime = INT\_MAX;

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

if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted) {

if (processes[i].burstTime < minBurstTime) {

minBurstTime = processes[i].burstTime;

idx = i;

}

if (processes[i].burstTime == minBurstTime) {

if (processes[i].arrivalTime < processes[idx].arrivalTime) {

idx = i;

}

}

}

}

if (idx == -1) {

currentTime++;

} else {

processes[idx].completionTime = currentTime + processes[idx].burstTime;

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

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

totalWaitingTime += processes[idx].waitingTime;

totalTurnaroundTime += processes[idx].turnaroundTime;

processes[idx].isCompleted = true;

currentTime = processes[idx].completionTime;

completed++;

ganttChart.push\_back(processes[idx]);

}

}

cout << "\nProcess\tArrival\tBurst\tCompletion\tTurnaround\tWaiting\n";

for (auto &p : processes) {

cout << "P" << p.id << "\t" << p.arrivalTime << "\t" << p.burstTime << "\t"

<< p.completionTime << "\t\t" << p.turnaroundTime << "\t\t"

<< p.waitingTime << "\n";

}

displayGanttChart(ganttChart);

cout << fixed << setprecision(2);

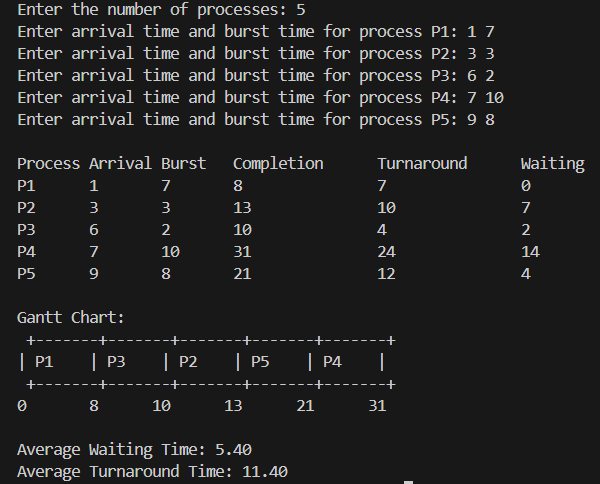
cout << "\nAverage Waiting Time: " << totalWaitingTime / n << endl;

cout << "Average Turnaround Time: " << totalTurnaroundTime / n << endl;

return 0;

}

**Output:**



**3) SRTF**

**Code:**

#include <iostream>

#include <vector>

#include <iomanip>

#include <limits>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int completionTime;

int waitingTime;

int turnaroundTime;

};

void printGanttChart(const vector<pair<int, int>> &ganttChart) {

cout << "\nGantt Chart:\n";

for (const auto &entry : ganttChart) {

cout << "| " << setw(8) << "P" << entry.first;

}

cout << "|";

cout << "\n";

cout << setw(1) << "0";

for (size\_t i = 0; i < ganttChart.size(); i++) {

cout << setw(8) << ganttChart[i].second;

}

cout << endl;

}

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

cout<<"\nProcess Table:\n";

cout<<setw(5)<< "ID"

<<setw(15)<< "Arrival Time"

<<setw(15)<< "Burst Time"

<<setw(20)<< "Completion Time"

<<setw(15)<< "Waiting Time"

<<setw(20)<< "Turnaround Time"

<<" \n";

for (const auto &p : processes){

cout<<setw(5)<<p.id

<<setw(15)<<p.arrivalTime

<<setw(15)<<p.burstTime

<<setw(20)<<p.completionTime

<<" "<<setw(15)<<p.waitingTime

<<" "<<setw(20)<<p.turnaroundTime

<<" \n";

}

cout<<"\nAverage Waiting Time: "<<avgWT

<<" \nAverage Turnaround Time: "<<avgTAT

<<" \n";

}

void srtf(vector<Process> &processes){

int n=processes.size();

vector<pair<int,int>> ganttChart; // Store pairs of (process ID, end time)

int completed=0,current\_time=0,totalWT=0,totalTAT=0;

while(completed<n){

int idx=-1,min\_time=numeric\_limits<int>::max();

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

if(processes[i].arrivalTime<=current\_time &&

processes[i].remainingTime>0 &&

processes[i].remainingTime<min\_time){

min\_time=processes[i].remainingTime;

idx=i;

}

}

if(idx!=-1){

if(ganttChart.empty() || ganttChart.back().first!=processes[idx].id){

ganttChart.push\_back({processes[idx].id,current\_time});

}

processes[idx].remainingTime--;

current\_time++;

if(processes[idx].remainingTime==0){

processes[idx].completionTime=current\_time;

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

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

totalWT+=processes[idx].waitingTime;

totalTAT+=processes[idx].turnaroundTime;

completed++;

ganttChart.back().second=current\_time; // Update last added process's end time.

}

} else{

current\_time++; // Idle time.

}

}

double avgWT=totalWT/n;

double avgTAT=totalTAT/n;

printGanttChart(ganttChart);

printProcessTable(processes,avgWT,avgTAT);

}

int main(){

int n;

cout<<"Enter the number of processes: ";

cin>>n;

vector<Process> processes(n);

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

processes[i].id=i+1;

cout<<"Enter arrival time and burst time for process P"<<i+1<<": ";

cin>>processes[i].arrivalTime>>processes[i].burstTime;

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

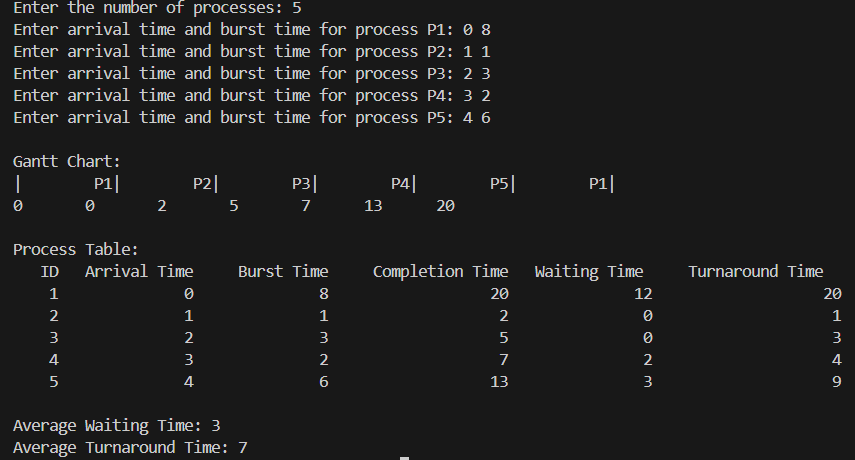
}

srtf(processes);

return 0;

}

**Output:**

****

**4) Priority-Non Preemptive:**

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int priority;

int completionTime;

int waitingTime;

int turnaroundTime;

};

// Function to print the process table and calculate average times

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

cout << "\nProcess Table:\n";

cout << setw(5) << "ID"

<< setw(15) << "Arrival Time"

<< setw(15) << "Burst Time"

<< setw(10) << "Priority"

<< setw(20) << "Completion Time"

<< setw(15) << "Waiting Time"

<< setw(20) << "Turnaround Time"

<< endl;

for (const auto &p : processes) {

cout << setw(5) << p.id

<< setw(15) << p.arrivalTime

<< setw(15) << p.burstTime

<< setw(10) << p.priority

<< setw(20) << p.completionTime

<< setw(15) << p.waitingTime

<< setw(20) << p.turnaroundTime

<< endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

void printGanttChart(const vector<Process> &ganttChart) {

cout << "\nGantt Chart:\n";

for (const auto &process : ganttChart) {

cout << "| " << setw(8) << "P" << process.id;

}

cout << "|"; // Close Gantt chart line

cout << "\n";

cout << setw(1) << "0";

int currentTime = 0;

for (const auto &process : ganttChart) {

currentTime += process.burstTime;

cout << setw(8) << currentTime; // Print end time of each process

}

cout << endl; // New line after printing all times

}

void priorityNonPreemptive(vector<Process> &processes) {

int n = processes.size();

vector<int> isCompleted(n, 0);

vector<Process> ganttChart; // To store the order of execution

double totalWT = 0, totalTAT = 0;

int currentTime = 0, completed = 0;

while (completed < n) {

int idx = -1;

int lowestPriority = numeric\_limits<int>::max(); // Initialize to max value

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

if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {

if (processes[i].priority < lowestPriority) { // Change the comparison here

lowestPriority = processes[i].priority;

idx = i;

}

}

}

if (idx != -1) {

ganttChart.push\_back(processes[idx]); // Add to Gantt chart

currentTime += processes[idx].burstTime;

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

isCompleted[idx] = 1;

completed++;

} else {

currentTime++;

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printGanttChart(ganttChart);

printProcessTable(processes, avgWT, avgTAT);

}

int main() {

int n;

cout << "Enter the number of processes: ";

cin >> n;

vector<Process> processes(n);

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

processes[i].id = i + 1;

cout << "Enter arrival time, burst time, and priority for process P" << i + 1 << ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime >> processes[i].priority;

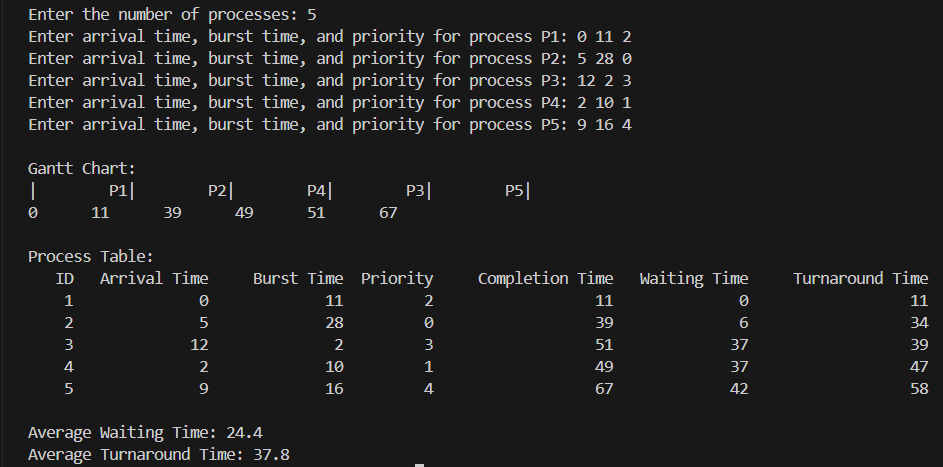
}

priorityNonPreemptive(processes);

return 0;

}

**Output:**



**Q5) Priority-preemptive**

**Code:**

#include <iostream>

#include <vector>

#include <iomanip>

#include <algorithm>

#include <limits>

using namespace std;

struct Process {

int id;

int arrivalTime;

int burstTime;

int remainingTime;

int priority;

int completionTime;

int waitingTime;

int turnaroundTime;

};

// Function to print the process table and calculate average times

void printProcessTable(const vector<Process> &processes, double avgWT, double avgTAT) {

cout << "\nProcess Table:\n";

cout << setw(5) << "ID"

<< setw(15) << "Arrival Time"

<< setw(15) << "Burst Time"

<< setw(10) << "Priority"

<< setw(20) << "Completion Time"

<< setw(15) << "Waiting Time"

<< setw(20) << "Turnaround Time"

<< endl;

for (const auto &p : processes) {

cout << setw(5) << p.id

<< setw(15) << p.arrivalTime

<< setw(15) << p.burstTime

<< setw(10) << p.priority

<< setw(20) << p.completionTime

<< setw(15) << p.waitingTime

<< setw(20) << p.turnaroundTime

<< endl;

}

cout << "\nAverage Waiting Time: " << avgWT << endl;

cout << "Average Turnaround Time: " << avgTAT << endl;

}

// Function to print the Gantt chart

void printGanttChart(const vector<pair<int, int>> &ganttChart) {

cout << "\nGantt Chart:\n";

// Print process IDs with grouping

for (size\_t i = 0; i < ganttChart.size(); ++i) {

if (i == 0 || ganttChart[i].first != ganttChart[i - 1].first) {

// Print the previous entry's duration if it's not the first entry

if (i > 0 && ganttChart[i - 1].first != -1) {

cout << "| ";

cout << std::setw(ganttChart[i - 1].second \* 4 + 1) // Adjust width based on duration

<< "P" << ganttChart[i - 1].first; // Print process ID

cout << " ";

} else if (i > 0 && ganttChart[i - 1].first == -1) {

cout << "| ";

cout << std::setw(ganttChart[i - 1].second \* 4 + 1)

<< "Idle"; // Print idle time

cout << " ";

}

}

// Handle last entry separately

if (i == ganttChart.size() - 1) {

if (ganttChart[i].first != -1)

cout << "| "

<< std::setw(ganttChart[i].second \* 4 + 1)

<< "P" << ganttChart[i].first; // Print last process ID

else

cout << "| "

<< std::setw(ganttChart[i].second \* 4 + 1)

<< "Idle"; // Print last idle time

}

}

cout << "|" << endl;

cout << "0 " ;

// Print time line, only at end of blocks

int time = 0;

for (size\_t i = 0; i < ganttChart.size(); ++i) {

time += ganttChart[i].second; // Increment time by burst time

// Only print the time at the end of each block

if (i == 0 || ganttChart[i].first != ganttChart[i - 1].first || i == ganttChart.size() - 1) {

cout << time; // Print end time for processes or idle periods

cout << " | "; // Separator for each time unit

}

}

cout << endl;

}

// Priority Preemptive Scheduling Function

void priorityPreemptive(vector<Process> &processes) {

int n = processes.size();

vector<int> isCompleted(n, 0);

vector<pair<int, int>> ganttChart; // To store (process ID, burst time)

double totalWT = 0, totalTAT = 0;

int currentTime = 0, completed = 0;

while (completed < n) {

int idx = -1;

int lowestPriority = numeric\_limits<int>::max(); // Initialize to max value

// Find the process with the lowest priority that has arrived and is not completed

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

if (processes[i].arrivalTime <= currentTime && !isCompleted[i]) {

if (processes[i].priority < lowestPriority) { // Change the comparison here

lowestPriority = processes[i].priority;

idx = i;

}

}

}

if (idx != -1) {

// Execute the process for 1 time unit

processes[idx].remainingTime--;

ganttChart.push\_back({processes[idx].id, 1}); // Record this process for Gantt chart

currentTime++;

// If the process is completed

if (processes[idx].remainingTime == 0) {

processes[idx].completionTime = currentTime;

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

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

totalWT += processes[idx].waitingTime;

totalTAT += processes[idx].turnaroundTime;

isCompleted[idx] = 1;

completed++;

}

} else {

// If no process is ready, increment time

ganttChart.push\_back({-1, 1}); // Indicate idle time

currentTime++;

}

}

double avgWT = totalWT / n;

double avgTAT = totalTAT / n;

printGanttChart(ganttChart);

printProcessTable(processes, avgWT, avgTAT);

}

int main() {

int n;

cout << "Enter the number of processes: ";

cin >> n;

vector<Process> processes(n);

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

processes[i].id = i + 1;

cout << "Enter arrival time, burst time, and priority for process P" << i + 1 << ": ";

cin >> processes[i].arrivalTime >> processes[i].burstTime >> processes[i].priority;

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

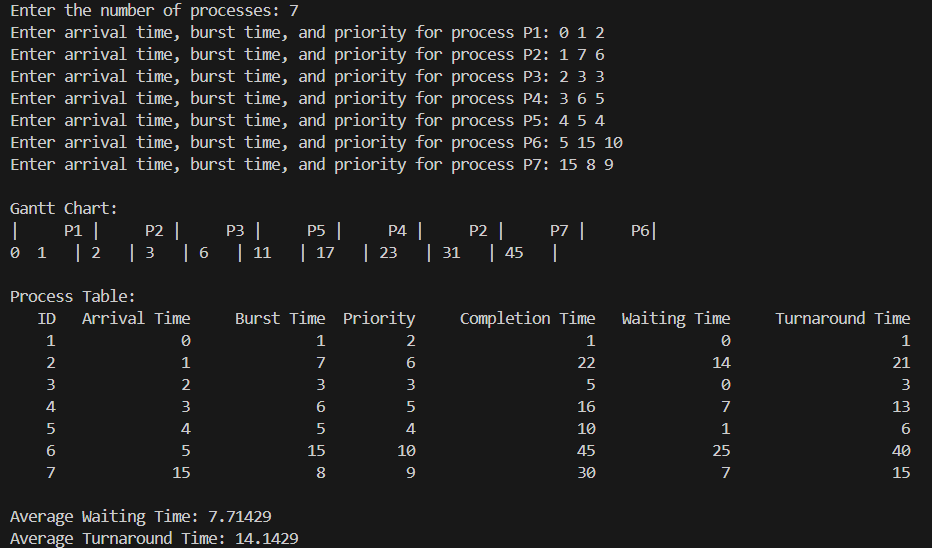
}

priorityPreemptive(processes);

return 0;

}

**Output:**

****

**6) Round Robin**

**Code:**

#include <iostream>

#include <vector>

#include <iomanip>

#include <algorithm>

using namespace std;

int main() {

int i, x = -1, m = 0, n, t, s = 0;

vector<int> bur(10), bur1(10), p(10), k(10), arrival(10);

vector<int> wat(10), tur(10);

vector<int> a(50);

int temp;

float awat, atur;

int ttur = 0, twat = 0, j = 0;

cout << "Enter no. of processes : ";

cin >> n;

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

cout << "Burst time for process P" << (i + 1) << " : ";

cin >> bur[i];

bur1[i] = bur[i];

cout << "Arrival time for process P" << (i + 1) << " : ";

cin >> arrival[i]; // Input for arrival time

}

cout << "Enter the time slice (in ms) : ";

cin >> t;

vector<int> b(n);

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

b[i] = bur[i] / t;

if ((bur[i] % t) != 0)

b[i] += 1;

m += b[i];

}

cout << "\nGANTT Chart\n";

for (i = 0; i < m; i++)

cout << "--------";

cout << "\n";

a[0] = 0;

vector<bool> done(n, false); // Track if a process is done

int currentTime = 0;

while (j < m) {

x = (x + 1) % n; // Cycle through processes

// Check if the process can run (it must be ready)

while (arrival[x] > currentTime) {

currentTime++; // Increment time until a process arrives

}

if (bur[x] > 0) { // Process can run

if (bur[x] >= t) {

bur[x] -= t;

a[j + 1] = currentTime + t;

currentTime += t;

} else {

currentTime += bur[x];

a[j + 1] = currentTime;

bur[x] = 0;

}

if (bur[x] == 0) {

p[s] = x;

k[s] = a[j + 1];

s++;

}

j++;

cout << " P" << (x + 1) << " |";

}

}

cout << "\n";

for (i = 0; i < m; i++)

cout << "--------";

cout << "\n";

for (j = 0; j <= m; j++)

cout << a[j] << "\t";

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

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

if (p[i] > p[j]) {

swap(p[i], p[j]);

swap(k[i], k[j]);

}

}

}

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

wat[i] = k[i] - arrival[p[i]] - bur1[p[i]];

tur[i] = k[i] - arrival[p[i]];

}

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

ttur += tur[i];

twat += wat[i];

}

awat = static\_cast<float>(twat) / n;

atur = static\_cast<float>(ttur) / n;

cout << "\n\nAverage waiting time : " << fixed << setprecision(2) << awat << " ms";

cout << "\nAverage turn around time : " << fixed << setprecision(2) << atur << " ms\n";

return 0;

}

**Output:**

