

**Program (FCFS):**

#include <stdio.h>

struct fcfs {

int pNo;

int AT;

int BT;

int CT;

int TAT;

int WT;

};

void sort(struct fcfs arr[], int n) {

struct fcfs temp;

for (int i = 0; i < n - 1; i++) {

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

if (arr[i].AT > arr[j].AT) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

int main() {

int n, atat = 0, awt = 0, temp = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct fcfs arr[n];

// Input process details

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

arr[i].pNo = i + 1;

printf("Enter arrival time for process %d: ", i + 1);

scanf("%d", &arr[i].AT);

printf("Enter burst time for process %d: ", i + 1);

scanf("%d", &arr[i].BT);

}

sort(arr, n); // Sort by arrival time

// Calculate Completion Time, Turnaround Time, and Waiting Time

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

if (temp < arr[i].AT) { // CPU is idle

printf("\nIdle time: %d -> %d\n", temp, arr[i].AT);

temp = arr[i].AT; // Update current time

}

arr[i].CT = temp + arr[i].BT; // Compute Completion Time

arr[i].TAT = arr[i].CT - arr[i].AT; // Turnaround Time

arr[i].WT = arr[i].TAT - arr[i].BT; // Waiting Time

atat += arr[i].TAT; // Accumulate Turnaround Time

awt += arr[i].WT; // Accumulate Waiting Time

temp = arr[i].CT; // Update temp to latest Completion Time

}

// Display process details

printf("\nProcess No\tArrival Time\tBurst Time\tCompletion Time\tTAT\tWaiting Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t%d\n",

arr[i].pNo, arr[i].AT, arr[i].BT, arr[i].CT, arr[i].TAT, arr[i].WT);

}

// Display average times

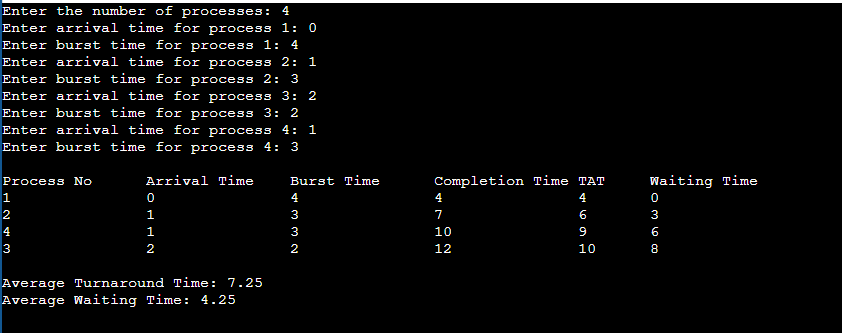
printf("\nAverage Turnaround Time: %.2f\n", (float)atat / n);

printf("Average Waiting Time: %.2f\n", (float)awt / n);

return 0;

}

**Output:**



**Program (SJF):**

#include <stdio.h>

struct Process {

int pNo, AT, BT, CT, TAT, WT;

};

void sortProcesses(struct Process p[], int n) {

struct Process temp;

for (int i = 0; i < n - 1; i++) {

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

// Sort based on Burst Time (if same, sort by Arrival Time)

if ((p[i].BT > p[j].BT) || (p[i].BT == p[j].BT && p[i].AT > p[j].AT)) {

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

void sjfNonPreemptive(struct Process p[], int n) {

int temp = 0, totalTAT = 0, totalWT = 0;

sortProcesses(p, n); // Sort by burst time

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

if (temp < p[i].AT) {

printf("Idle Time: %d -> %d", temp , p[i].AT);

temp = p[i].AT; // CPU remains idle

}

p[i].CT = temp + p[i].BT; // Completion Time

p[i].TAT = p[i].CT - p[i].AT; // Turnaround Time

p[i].WT = p[i].TAT - p[i].BT; // Waiting Time

totalTAT += p[i].TAT;

totalWT += p[i].WT;

temp = p[i].CT;

}

printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pNo, p[i].AT, p[i].BT, p[i].CT, p[i].TAT, p[i].WT);

}

printf("\nAverage Turnaround Time: %.2f", (float)totalTAT / n);

printf("\nAverage Waiting Time: %.2f\n", (float)totalWT / n);

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

p[i].pNo = i + 1;

printf("Enter Arrival Time and Burst Time for Process %d: ", i + 1);

scanf("%d %d", &p[i].AT, &p[i].BT);

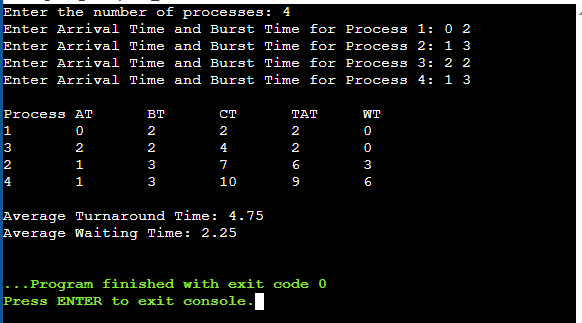
}

sjfNonPreemptive(p, n);

return 0;

}

**Output:**



**Program (Round Robin):**

#include <stdio.h>

#include <stdbool.h>

#define Q\_SIZE 1000

int Q[Q\_SIZE], front = -1, rear = -1;

struct RR {

int pNo;

int AT;

int BT;

int original\_BT;

int CT;

int TAT;

int WT;

};

// Function to sort processes based on arrival time

void sort(struct RR arr[], int n) {

struct RR temp;

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

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

if (arr[i].AT > arr[j].AT) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

// Function to add a process to the queue

void Enqueue(int value) {

if (rear == Q\_SIZE - 1) {

printf("\nQueue is full.");

return;

}

if (front == -1) {

front = 0;

}

rear++;

Q[rear] = value;

}

// Function to remove a process from the queue

int Dequeue() {

if (front == -1 || front > rear) {

return -1; // Queue is empty

}

int x = Q[front];

front++;

return x;

}

// Function to check if the queue is empty

bool isEmpty() {

return front == -1 || front > rear;

}

// Function to check if a process is already in the queue

bool contains(int value) {

for (int i = front; i <= rear; i++) {

if (Q[i] == value) {

return true;

}

}

return false;

}

int main() {

int n, TQ, atat = 0, awt = 0, temp = 0;

// Take Time Quantum

printf("\nEnter the Time Quantum: ");

scanf("%d", &TQ);

printf("Enter the number of processes: ");

scanf("%d", &n);

struct RR arr[n];

// Input process details

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

arr[i].pNo = i + 1;

printf("Enter arrival time for process %d: ", i + 1);

scanf("%d", &arr[i].AT);

printf("Enter burst time for process %d: ", i + 1);

scanf("%d", &arr[i].BT);

arr[i].original\_BT = arr[i].BT; // Store original burst time

}

// Sort processes as per the arrival time

sort(arr, n);

// Initially add the first process to the queue

Enqueue(0); // 0 index contains the first process

// Process the queue

while (!isEmpty()) {

int i = Dequeue();

// Handle idle time if no process is ready

if (temp < arr[i].AT) {

printf("\nIdle time: %d -> %d\n", temp, arr[i].AT);

temp = arr[i].AT;

}

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

if (arr[i].BT <= TQ) {

temp += arr[i].BT;

arr[i].BT = 0; // Process finishes

arr[i].CT = temp;

} else {

temp += TQ;

arr[i].BT -= TQ;

}

// Add newly arrived processes to the queue

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

if (arr[j].AT <= temp && arr[j].BT > 0 && !contains(j)) {

Enqueue(j);

}

}

// If the current process is not finished, add it back to the queue

if (arr[i].BT > 0) {

Enqueue(i);

}

}

// Calculate TAT, WT, ATAT, and AWT

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

arr[i].TAT = arr[i].CT - arr[i].AT;

arr[i].WT = arr[i].TAT - arr[i].original\_BT;

atat += arr[i].TAT;

awt += arr[i].WT;

}

// Print results

printf("\nPNo\tAT\tBT\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\t%d\n",

arr[i].pNo,

arr[i].AT,

arr[i].original\_BT,

arr[i].CT,

arr[i].TAT,

arr[i].WT);

}

// Print averages

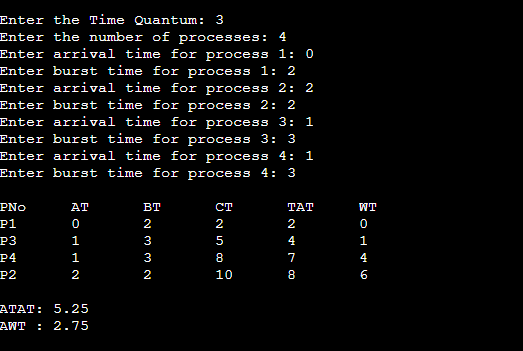
printf("\nATAT: %.2f", (float)atat / n);

printf("\nAWT : %.2f\n", (float)awt / n);

return 0;

}

**Output:**



**Program (Priority Non Pre-emptive):**

#include <stdio.h>

// Lesser no higher priority

struct priority{

int pNo;

int AT;

int BT;

int priority;

int CT;

int TAT;

int WT;

};

void sort(struct priority arr[], int n){

struct priority temp;

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

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

// Sort as per the priority and AT

if(arr[i].priority > arr[j].priority || (arr[i].priority == arr[j].priority && arr[i].AT >arr[j].AT)){

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

int main(){

int n, atat = 0, awt = 0 , temp = 0;

printf("Enter the no of process:");

scanf("%d", &n);

struct priority arr[n];

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

arr[i].pNo = i+1;

printf("Enter the AT, BT & Priority for the process %d: ", i+1);

scanf("%d %d %d", &arr[i].AT, &arr[i].BT, &arr[i].priority);

}

sort(arr, n);

// Calculate CT, TAT, WT, ATAT & AWt

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

// Handle idle time

if(temp < arr[i].AT){

printf("Idle time: %d -> %d\n", temp, arr[i].AT);

temp = arr[i].AT;

}

arr[i].CT = temp + arr[i].BT;

arr[i].TAT = arr[i].CT - arr[i].AT;

arr[i].WT = arr[i].TAT - arr[i].BT;

temp = arr[i].CT;

atat += arr[i].TAT;

awt += arr[i].WT;

}

// Display results in short form

printf("\nPNo\tAT\tBT\tPri\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

arr[i].pNo,

arr[i].AT,

arr[i].BT,

arr[i].priority,

arr[i].CT,

arr[i].TAT,

arr[i].WT);

}

// Display average times

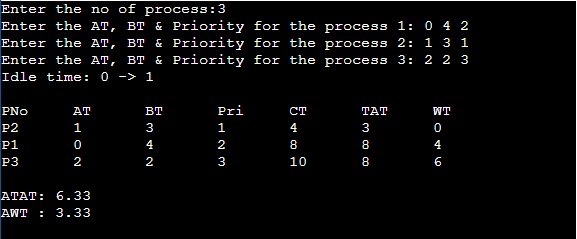
printf("\nATAT: %.2f", (float)atat / n);

printf("\nAWT : %.2f\n", (float)awt / n);

return 0;

}

**Output:**



**Program (Priority Pre-emptive):**

#include <stdio.h>

#include <stdbool.h>

// Lesser number means higher priority

struct priority {

int pNo;

int AT;

int BT;

int remain\_BT;

int priority;

int CT;

int TAT;

int WT;

bool completed;

};

void sort(struct priority arr[], int n) {

struct priority temp;

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

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

// Sort by arrival time

if (arr[i].AT > arr[j].AT) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

int findHighestPriority(struct priority arr[], int n, int currentTime) {

int highestPriorityIndex = -1;

int highestPriority = 9999; // Assuming lower number means higher priority

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

if (!arr[i].completed && arr[i].AT <= currentTime && arr[i].priority < highestPriority) {

highestPriority = arr[i].priority;

highestPriorityIndex = i;

}

}

return highestPriorityIndex;

}

int main() {

int n, atat = 0, awt = 0, currentTime = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct priority arr[n];

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

arr[i].pNo = i + 1;

printf("Enter the AT, BT & Priority for process %d: ", i + 1);

scanf("%d %d %d", &arr[i].AT, &arr[i].BT, &arr[i].priority);

arr[i].remain\_BT = arr[i].BT;

arr[i].completed = false;

}

sort(arr, n);

int completedProcesses = 0;

while (completedProcesses < n) {

int highestPriorityIndex = findHighestPriority(arr, n, currentTime);

if (highestPriorityIndex == -1) {

// No process available, CPU is idle

printf("Idle time: %d -> %d\n", currentTime, currentTime + 1);

currentTime++;

} else {

// Execute the process for 1 unit of time

arr[highestPriorityIndex].remain\_BT--;

currentTime++;

if (arr[highestPriorityIndex].remain\_BT == 0) {

// Process completed

arr[highestPriorityIndex].completed = true;

arr[highestPriorityIndex].CT = currentTime;

arr[highestPriorityIndex].TAT = arr[highestPriorityIndex].CT - arr[highestPriorityIndex].AT;

arr[highestPriorityIndex].WT = arr[highestPriorityIndex].TAT - arr[highestPriorityIndex].BT;

atat += arr[highestPriorityIndex].TAT;

awt += arr[highestPriorityIndex].WT;

completedProcesses++;

}

}

}

// Display results in short form

printf("\nPNo\tAT\tBT\tPri\tCT\tTAT\tWT\n");

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

printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

arr[i].pNo,

arr[i].AT,

arr[i].BT,

arr[i].priority,

arr[i].CT,

arr[i].TAT,

arr[i].WT);

}

// Display average times

printf("\nATAT: %.2f", (float)atat / n);

printf("\nAWT : %.2f\n", (float)awt / n);

return 0;

}

**Output:**

