Lab No: 5 **Date: 2082/**

Title: Write a program to calculate the average turnaround time and waiting time for user input process parameters using SJF process scheduling algorithm.

Shortest Job First (SJF) is a CPU scheduling algorithm where the process with the smallest burst time is executed first. It can be preemptive or non-preemptive. SJF gives the minimum average waiting time among all scheduling algorithms but requires knowledge of process burst times in advance.

Algorithm:

Step 1: Start with the list of processes and note their arrival time and burst time.

Step 2: At every scheduling point, select the process with the shortest burst time among the processes that have arrived.

Step 3: Allocate CPU to that process.

Step 4: After completion, update the ready queue and select the next shortest process.

Step 5: Calculate waiting time and turnaround time for each process.

Step 6: Compute the average waiting time and average turnaround time.

Step 7: Stop.

Language: C++

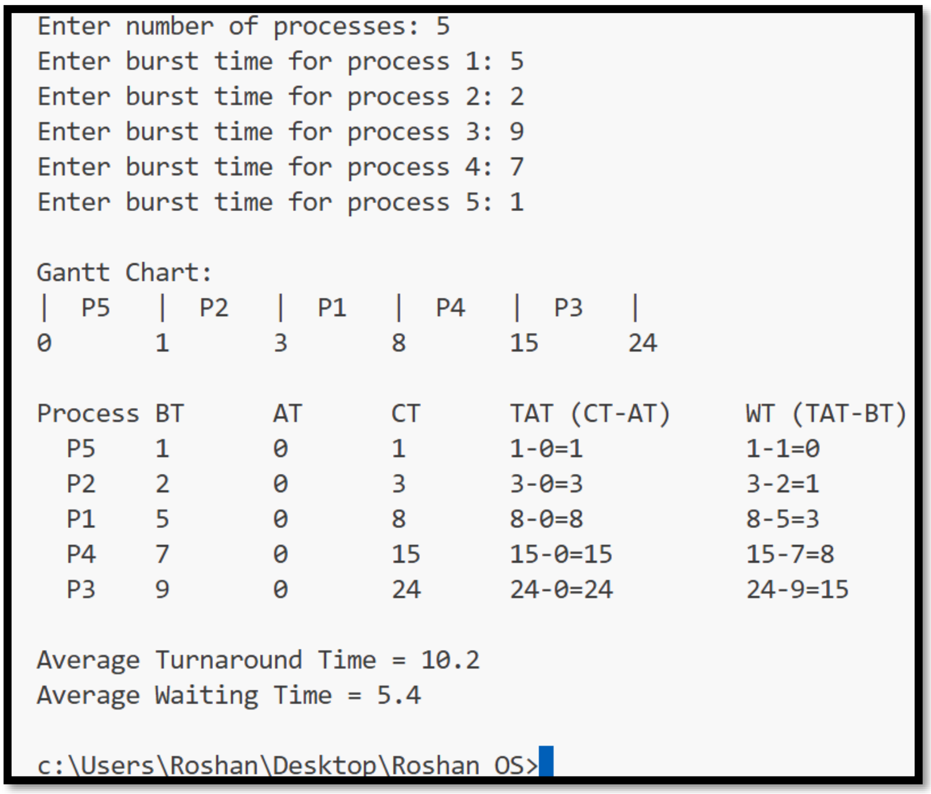
IDE: VS Code

Code:

```
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;


struct Job {

    int pid;   // process id

    int bt;    // burst time

    int at;    // arrival time

    int ct;    // completion time

    int tat;   // turnaround time

    int wt;    // waiting time

};


int main() {

    int count;

    cout << "Enter number of processes: ";

    cin >> count;

    vector<Job> procs(count);

    // All arrival times = 0

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

        procs[i].pid = i + 1;

        procs[i].at = 0;

        cout << "Enter burst time for process " << i + 1 << ": ";
        cin >> procs[i].bt;
    }
    // ---- SJF: sort by burst time ----
    sort(procs.begin(), procs.end(), [](Job &x, Job &y) {
        return x.bt < y.bt;
    });
    // Calculate CT, TAT, WT
    int clk = 0;
    for (int i = 0; i < count; i++) {
        if (clk < procs[i].at)
            clk = procs[i].at;
        clk += procs[i].bt;
        procs[i].ct = clk;
        procs[i].tat = procs[i].ct - procs[i].at;
        procs[i].wt = procs[i].tat - procs[i].bt;
    }
    // Gantt Chart
    cout << "\nGantt Chart:\n|";
    clk = 0;
    for (int i = 0; i < count; i++) {
        cout << "  P" << procs[i].pid << "\t|";
    }
    cout << "\n0";
    for (int i = 0; i < count; i++) {
        clk += procs[i].bt;
```

```
        cout << "     \t" << clk;
    }
    cout << "\n";
    // Result table
    cout << "\nProcess\tBT\tAT\tCT\tTAT (CT-AT)\tWT (TAT-BT)\n";
    double sumTAT = 0, sumWT = 0;
    for (int i = 0; i < count; i++) {
            cout << "  P" << procs[i].pid << "\t";
            cout<< procs[i].bt << "\t";
            cout<< procs[i].at << "\t";
            cout<< procs[i].ct << "\t";
            cout<< procs[i].ct << "-" << procs[i].at;
            cout<< "=" << procs[i].tat << "\t\t";
            cout<< procs[i].tat << "-" << procs[i].bt;
            cout<< "=" << procs[i].wt << "\n";
        sumTAT += procs[i].tat;
        sumWT += procs[i].wt;
    }
    cout << "\nAverage Turnaround Time = " << sumTAT / count;
    cout << "\nAverage Waiting Time = " << sumWT / count << "\n";
    return 0;
}
```

Output:


```
Enter number of processes: 5
Enter burst time for process 1: 5
Enter burst time for process 2: 2
Enter burst time for process 3: 9
Enter burst time for process 4: 7
Enter burst time for process 5: 1

Gantt Chart:
|  P5    |  P2    |  P1    |  P4    |  P3    |
0       1       3       8       15      24

Process BT      AT      CT      TAT (CT-AT)     WT (TAT-BT)
  P5    1       0       1       1-0=1           1-1=0
  P2    2       0       3       3-0=3           3-2=1
  P1    5       0       8       8-0=8           8-5=3
  P4    7       0       15      15-0=15         15-7=8
  P3    9       0       24      24-0=24         24-9=15

Average Turnaround Time = 10.2
Average Waiting Time = 5.4

c:\Users\Roshan\Desktop\Roshan OS>
```

Conclusion:

SJF is efficient because it reduces average waiting time and turnaround time compared to FCFS. However, it suffers from the problem of starvation (long processes may never get CPU if shorter ones keep arriving). To overcome this, aging can be used.