# LABORATORY EXERCISE 6

# SHORTEST JOB FIRST SCHEDULING

**Learning Objectives**

* To understand the Shortest Job First (SJF) CPU scheduling algorithm.
* To implement the SJF algorithm in C++.
* To analyze and compare the efficiency of SJF in terms of average waiting time and turnaround time.
* To explore the impact of SJF scheduling on system performance and response time.

**Prerequisite student experiences and knowledge**

A basic understanding of CPU scheduling algorithms and process management in operating systems is essential for optimizing how processes are executed and managed. Familiarity with key concepts such as burst time, waiting time, and turnaround time, along with their calculation in scheduling algorithms, helps analyze and improve system performance. Additionally, experience in C or C++ programming, particularly with loops, arrays, and basic input/output handling, is crucial for implementing these scheduling algorithms effectively and understanding their impact on real-time process execution.

**Background**

In operating systems, CPU scheduling determines which processes are executed by the CPU and in what order. The Shortest Job First (SJF) scheduling algorithm is a non-preemptive scheduling technique where the process with the shortest burst time is selected for execution next. This scheduling method minimizes the average waiting time and is optimal when all jobs are available at the same time. However, SJF can suffer from the "starvation" problem for longer processes if shorter jobs continuously arrive.

This exercise involves implementing the SJF algorithm to simulate how it works and calculating key performance metrics like waiting and turnaround times.

**Materials/Resources**

* PC/Internet
* Pen
* C++ (TurboC)
* Web Browser (Internet Explorer, Mozilla, Google Chrome, Etc.)
* Word-processing program

**Laboratory Activity**

**Instructions:** *Perform the following steps.*

Step 1: Start the IDE

* + Open your C++ IDE (e.g., DevC++ or TurboC++).
  + Create a new file called SJF\_scheduling.cpp.

Step 2: Define the Process Structure

* + Define a structure called a process that contains:
    - Process ID.
    - Arrival Time.
    - Burst Time.
    - Waiting Time.
    - Turnaround Time.

Step 3: Implement the SJF Scheduling Algorithm

* + Sort the processes by burst time using an appropriate sorting algorithm (e.g., std::sort in C++).
  + Calculate the waiting time for each process:
  + The first process has zero waiting time.
  + For each subsequent process, the waiting time is the sum of the burst times of all previous processes.
  + Calculate the turnaround time for each process:
  + Turnaround time is the sum of each process's waiting time and burst time.

Step 4: Display the Results

* + After calculating waiting and turnaround times, display the process ID, burst time, waiting time, and turnaround time for each process in tabular format.
  + Also, compute and display the average waiting time and average turnaround time.

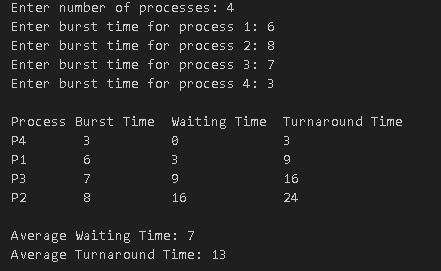
Step 5: Testing the Program

* + Run the program and test it with at least three sets of input data:
  + One set with randomly ordered burst times.
  + One set with burst times in ascending order.
  + One set with burst times in descending order.

Step 6: Output/Result

* + Capture the output for each test case and display it in tabular format.
  + Provide a final summary of the test results, including each test case's average waiting time and turnaround time.

**Sample:**

****

**Test Table**

**Table 1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process Number** | **Arrival Time** | **Burst Time** | **Completion Time** | **Waiting Time** | **Turnaround Time** |
| **A** | **0** | **5** | **5** | **0** | **5** |
| **B** | **1** | **3** | **17** | **13** | **16** |
| **C** | **2** | **4** | **24** | **18** | **22** |
| **D** | **3** | **2** | **8** | **3** | **5** |
| **E** | **4** | **1** | **6** | **1** | **2** |
| **F** | **5** | **2** | **10** | **3** | **5** |
| **G** | **6** | **4** | **28** | **18** | **22** |
| **H** | **7** | **3** | **20** | **10** | **13** |
| **I** | **8** | **2** | **12** | **2** | **4** |
| **J** | **9** | **2** | **14** | **3** | **5** |

**Average Waiting Time: 0+13+18+3+1+3+18+10+2+3/10 =**

**7.10**

**Average Turnaround Time:5+16+22+5+2+5+22+13+2+4/10 =**

**9.90**

**Gantt Chart**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | **E** | **D** | **F** | **I** | **J** | **B** | **H** | **C** | **G** |

**0 5 6 8 10 12 14 17 20 24 28**

**Table 1.2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Process Number** | **Arrival Time** | **Burst Time** | **Completion Time** | **Waiting Time** | **Turnaround Time** |
| **A** | **1** | **3** | **10** | **6** | **9** |
| **B** | **1** | **1** | **2** | **0** | **1** |
| **C** | **2** | **5** | **25** | **18** | **23** |
| **D** | **3** | **3** | **13** | **7** | **10** |
| **E** | **2** | **1** | **3** | **0** | **1** |
| **F** | **3** | **5** | **30** | **22** | **27** |
| **G** | **6** | **4** | **20** | **10** | **14** |
| **H** | **4** | **3** | **16** | **9** | **12** |
| **I** | **2** | **2** | **5** | **1** | **3** |
| **J** | **5** | **2** | **7** | **0** | **2** |

**Average Waiting Time: 6+0+18+7+0+22+10+9+1+0/10 =**

**7.30**

**Average Turnaround Time: 9+1+23+10+1+27+14+12+3+2/10 =**

**10.20**

**Gantt Chart**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **B** | **E** | **I** | **J** | **A** | **D** | **H** | **G** | **C** | **F** |

**0 2 3 5 7 10 13 16 20 25 30**

**QUESTIONS (10 pts each)**

1. How does the SJF scheduling algorithm minimize the average waiting time compared to other scheduling algorithms like FCFS?

SJF gives priority to shorter tasks first which helps clear the queue quicker and longer task do not wait for so much time. While FCFS queues every process based on arrival time, SJF reduces total time in queue by focusing on the shortest process first.

1. Discuss the limitations of the SJF algorithm. In what scenarios might SJF not be the optimal choice?

The Shortest Job First (SJF) scheduling algorithm carries with it a number of disadvantages. First of all, the scheduling can be done only if the burst time of each task is known in advance, which is most difficult to obtain realistically. Secondly, SJF may cause some intolerance in the system, where long jobs keep waiting for a long time due to the ongoing arrival of shorter tasks. Lastly, it is unfit for any real-time systems as it is very inefficient in handling deadlines, which is the opposite of what is required for such time-constrained activities.

1. Explain the concept of "starvation" in CPU scheduling. How can it occur in the SJF algorithm?

Longer operations may be delayed repeatedly by shorter ones leading to starvation. Since SJF schedules all the jobs on hand starting with the shortest, longer jobs do not get executed at all if shorter ones keep coming.

1. Provide a Flowchart of the given task.

Start

Input Process Data

Sort by Burst Time

Initialize Waiting Time

Calculate Waiting Time

Calculate Turnaround Time

Display Result

Calculate Average

End

**Source Code**

#include <iostream>

#include <iomanip>

#include <vector>

#include <algorithm>

struct Process {

int processID;

int arrivalTime;

int burstTime;

int waitingTime;

int turnaroundTime;

};

void calculateTimes(std::vector<Process>& processes) {

processes[0].waitingTime = 0;

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

for (size\_t i = 1; i < processes.size(); i++) {

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

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

}

}

void displayResults(const std::vector<Process>& processes) {

int totalWaitingTime = 0, totalTurnaroundTime = 0;

std::cout << std::setw(10) << "Process ID"

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

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

<< std::setw(20) << "Turnaround Time\n";

for (const auto& process : processes) {

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

std::cout << std::setw(10) << process.processID

<< std::setw(15) << process.burstTime

<< std::setw(15) << process.waitingTime

<< std::setw(20) << process.turnaroundTime << "\n";

}

double avgWaitingTime = static\_cast<double>(totalWaitingTime) / processes.size();

double avgTurnaroundTime = static\_cast<double>(totalTurnaroundTime) / processes.size();

std::cout << "\nAverage Waiting Time: " << avgWaitingTime

<< "\nAverage Turnaround Time: " << avgTurnaroundTime << "\n";

}

int main() {

int n;

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

std::cin >> n;

std::vector<Process> processes(n);

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

processes[i].processID = i + 1;

std::cout << "Enter Arrival Time and Burst Time for Process " << i + 1 << ": ";

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

}

std::sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {

return a.burstTime < b.burstTime;

});

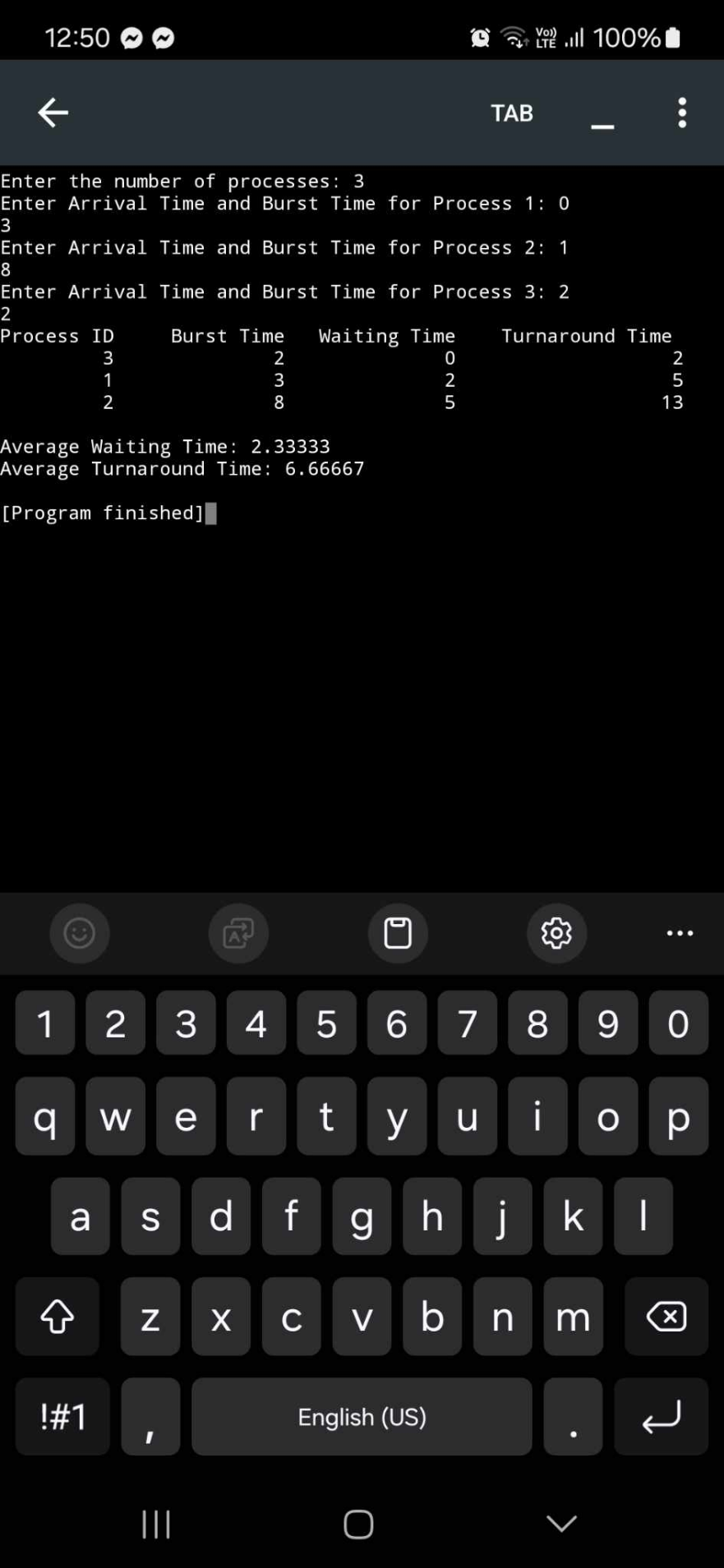
calculateTimes(processes);

displayResults(processes);

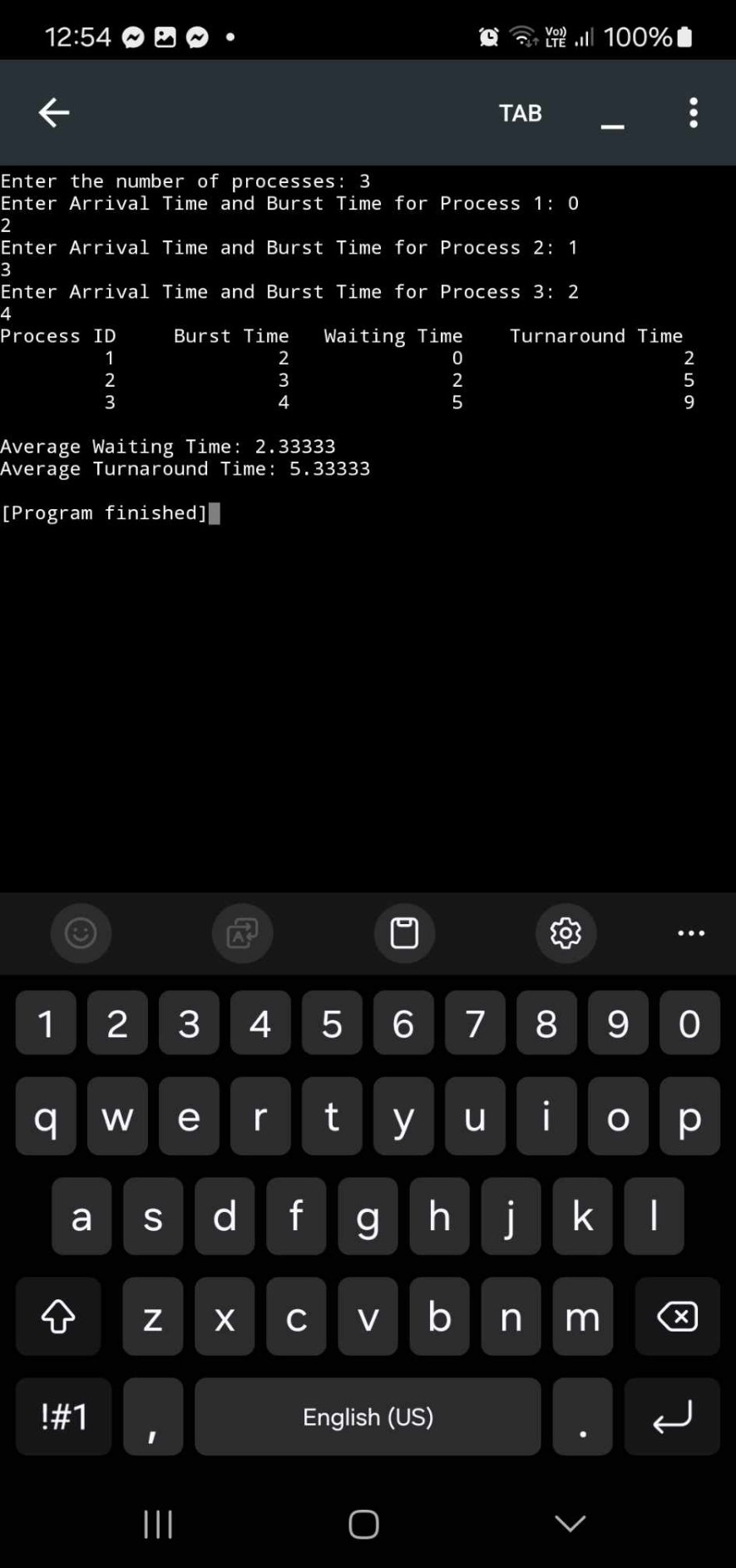
return 0;

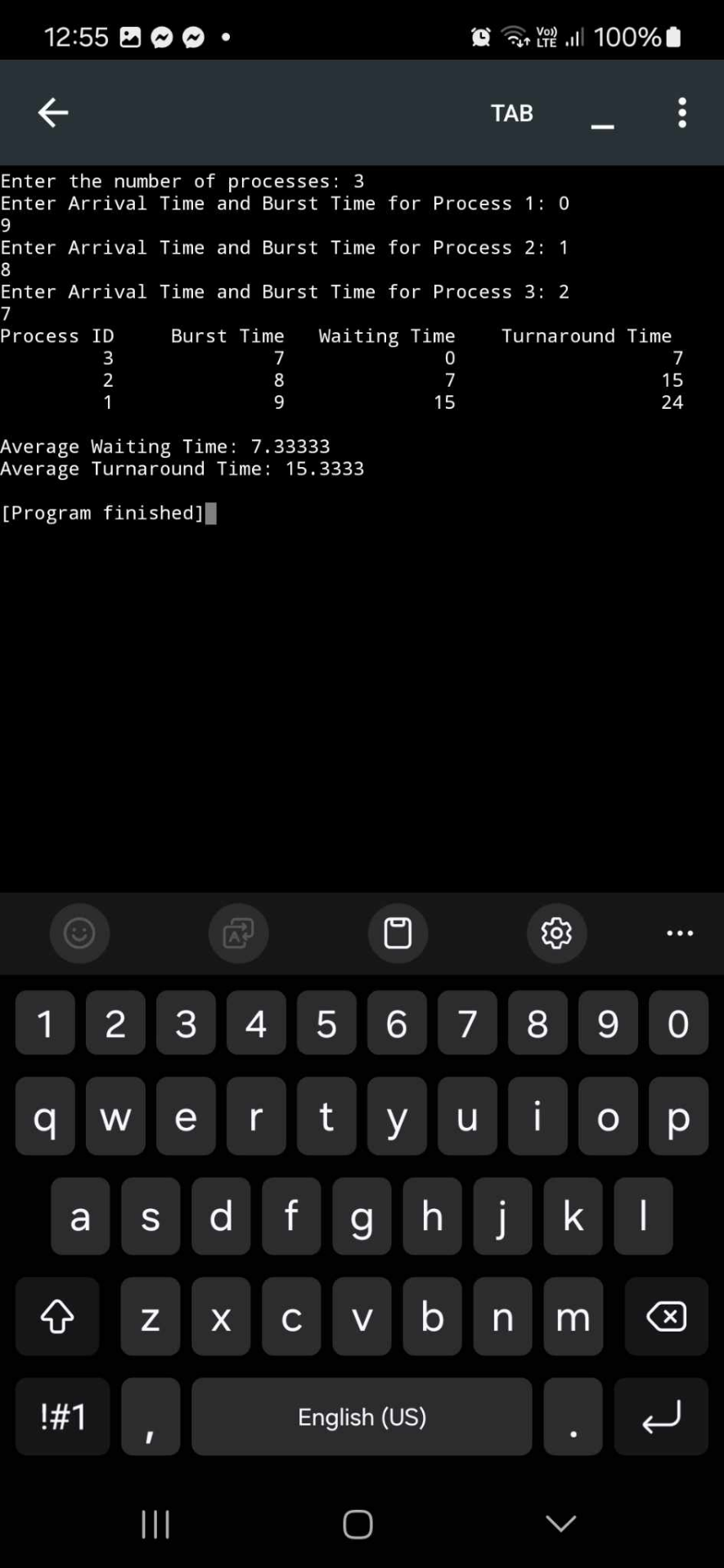
}

**Output / Results**

**Randomly ordered Burst times**

**Burst times in Ascending Order**

****

**Burst times in Descending Order**

**Conclusion**

The Shortest Job First (SJF) scheduling algorithm focuses on minimizing average waiting times by prioritizing shorter jobs, which helps clear the process queue more efficiently than the First Come First Served method. However, its effectiveness relies on accurately knowing task completion times. Additionally, if shorter jobs continuously arrive, longer tasks can be left waiting for extended periods. This makes implementing SJF in real-time systems with strict deadlines quite challenging. While SJF can significantly reduce waiting times, its limitations raise concerns about task management, especially in time-sensitive situations.