Hands-on Activity 15.1

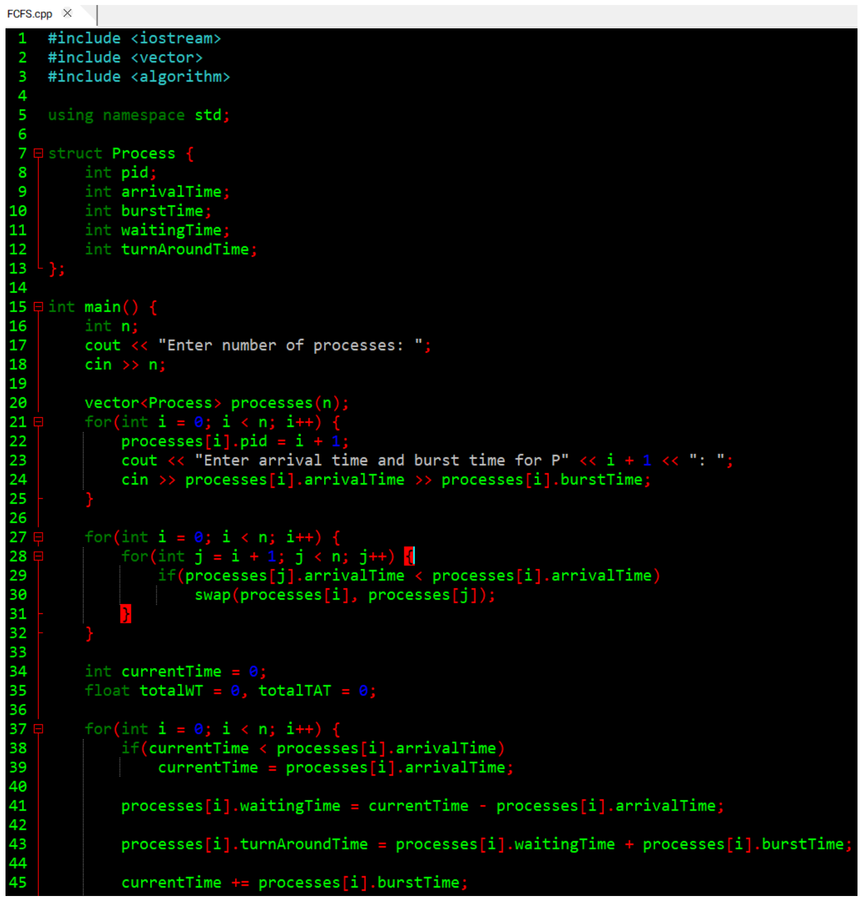
Scheduling Algorithms in C++

Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/18/2025
Section: CPE21S3	**Date Submitted:** 11/18/2025
Name(s): Alcantara, Jason P. Anastacio, Lester Arvid P. Avila, Vince Gabriel V. Catungal, Jan Kerwin B.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

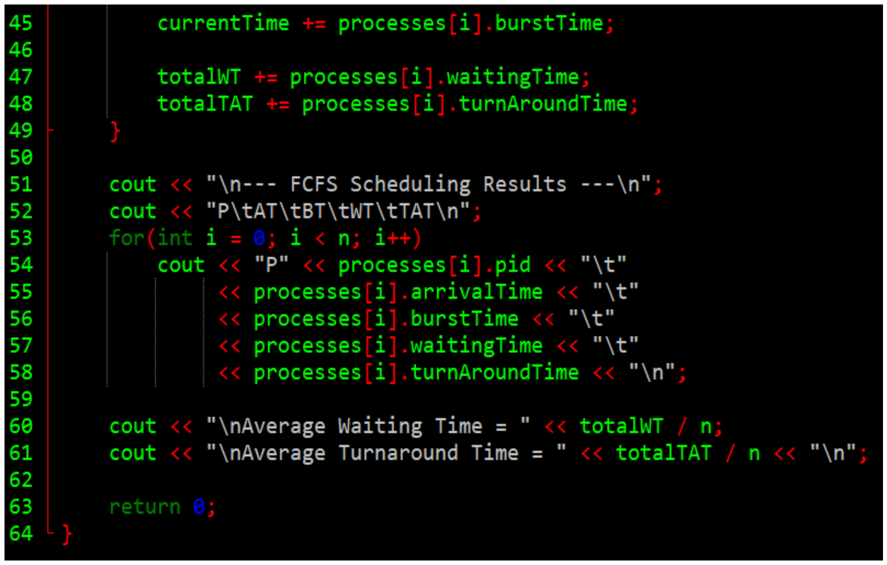
A.1. First Come First Serve:

Main CPP:



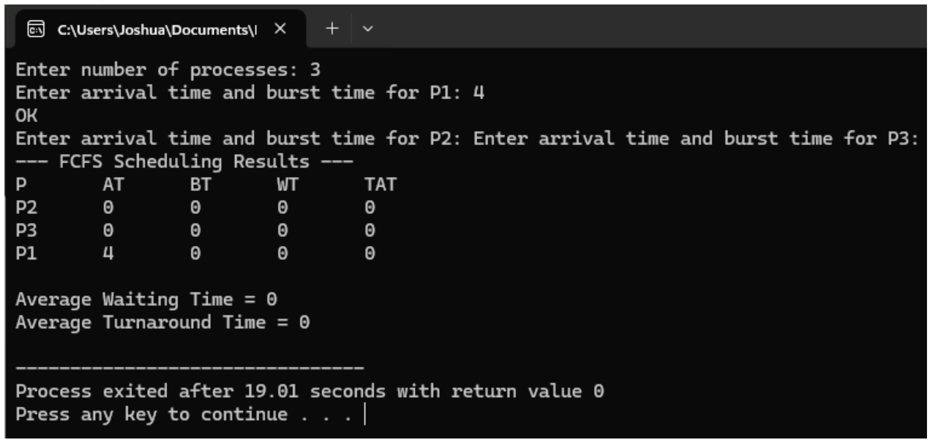
FCFS.cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time and burst time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n; j++) {
            if(processes[j].arrivalTime < processes[i].arrivalTime)
                swap(processes[i], processes[j]);
        }
    }

    int currentTime = 0;
    float totalWT = 0, totalTAT = 0;

    for(int i = 0; i < n; i++) {
        if(currentTime < processes[i].arrivalTime)
            currentTime = processes[i].arrivalTime;

        processes[i].waitingTime = currentTime - processes[i].arrivalTime;

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

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



```cpp
        currentTime += processes[i].burstTime;

        totalWT += processes[i].waitingTime;
        totalTAT += processes[i].turnAroundTime;
    }

    cout << "\n--- FCFS Scheduling Results ---\n";
    cout << "P\tAT\tBT\tWT\tTAT\n";
    for(int i = 0; i < n; i++)
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].waitingTime << "\t"
             << processes[i].turnAroundTime << "\n";

    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

    return 0;
}
```

The Output:



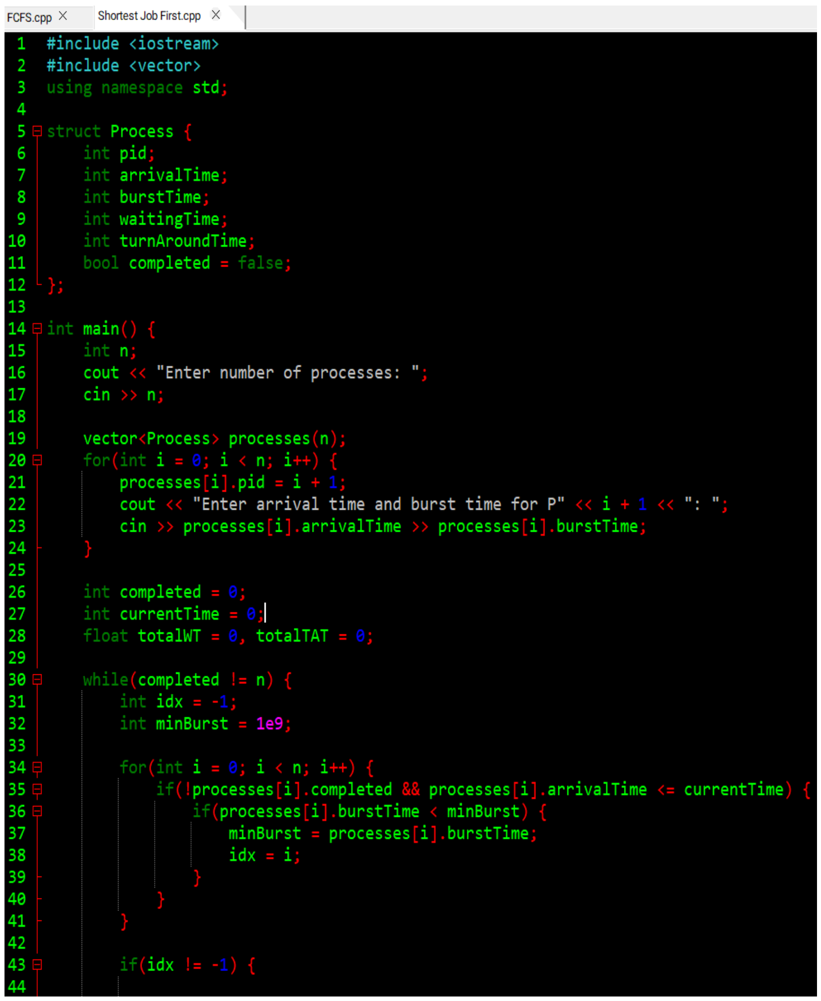
```
Enter number of processes: 3
Enter arrival time and burst time for P1: 4
OK
Enter arrival time and burst time for P2: Enter arrival time and burst time for P3:
--- FCFS Scheduling Results ---
P       AT      BT      WT      TAT
P2      0       0       0       0
P3      0       0       0       0
P1      4       0       0       0

Average Waiting Time = 0
Average Turnaround Time = 0

--------------------------------
Process exited after 19.01 seconds with return value 0
Press any key to continue . . .
```

The Analysis:

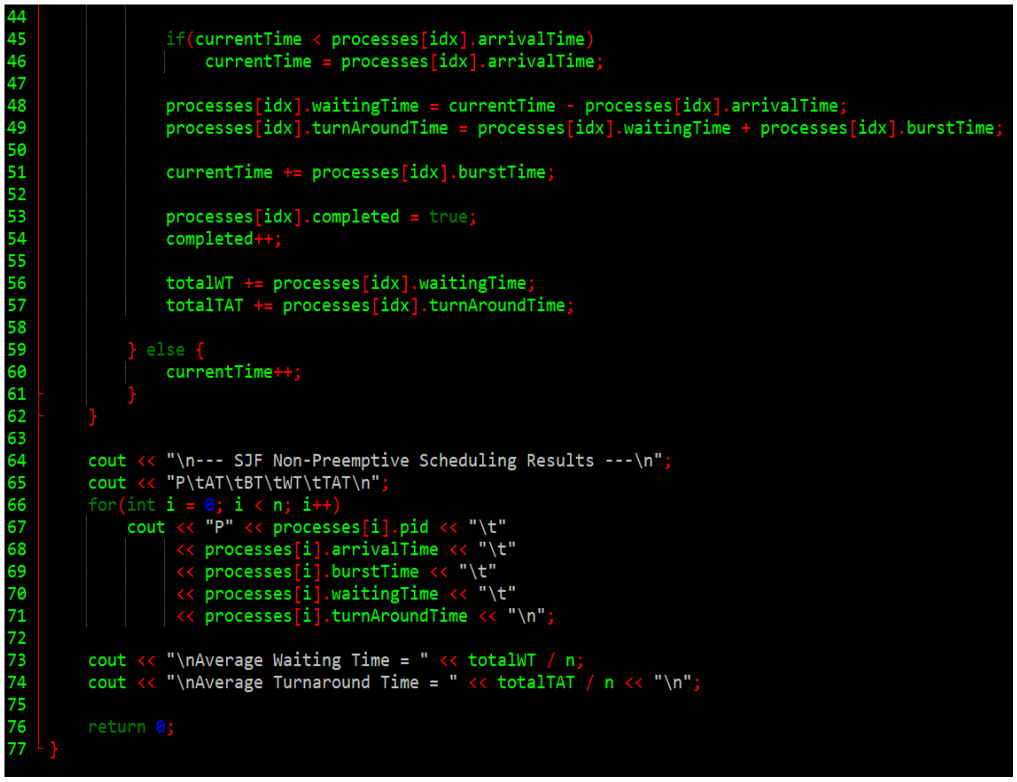
This C++ program simulates the First-Come, First-Served (FCFS) CPU scheduling algorithm, which is a concept in operating systems. The program first receives, from the user, the arrival time and burst time for a collection of processes. It then sorts the processes based solely upon the arrival time so that it can properly implement the FCFS rule. The program then simulates the processes being run in order, maintain a variable to keep track of execution time, and calculates the waiting time and turnaround time for each job that finishes executing (without preemption). The simulation properly acknowledges periods of idle time for the CPU if a process arrives after the previous job finishes. Lastly, the program produces an output, displaying a table of important per-process information, and calculates system performance metrics Average Waiting Time and Average Turnaround Time.

A.2 Shortest Job First

Main CPP:

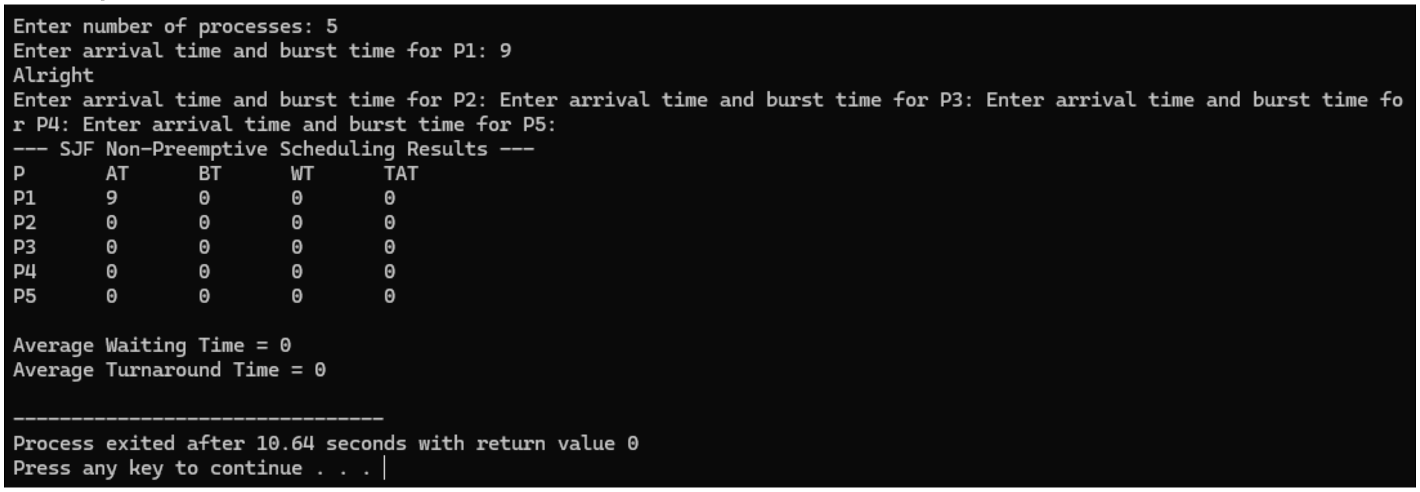


```cpp
#include <iostream>
#include <vector>
using namespace std;

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
    bool completed = false;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time and burst time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    int completed = 0;
    int currentTime = 0;
    float totalWT = 0, totalTAT = 0;

    while(completed != n) {
        int idx = -1;
        int minBurst = 1e9;

        for(int i = 0; i < n; i++) {
            if(!processes[i].completed && processes[i].arrivalTime <= currentTime) {
                if(processes[i].burstTime < minBurst) {
                    minBurst = processes[i].burstTime;
                    idx = i;
                }
            }
        }

        if(idx != -1) {
```



```cpp

            if(currentTime < processes[idx].arrivalTime)
                currentTime = processes[idx].arrivalTime;

            processes[idx].waitingTime = currentTime - processes[idx].arrivalTime;
            processes[idx].turnAroundTime = processes[idx].waitingTime + processes[idx].burstTime;

            currentTime += processes[idx].burstTime;

            processes[idx].completed = true;
            completed++;

            totalWT += processes[idx].waitingTime;
            totalTAT += processes[idx].turnAroundTime;

        } else {
            currentTime++;
        }
    }

    cout << "\n--- SJF Non-Preemptive Scheduling Results ---\n";
    cout << "P\tAT\tBT\tWT\tTAT\n";
    for(int i = 0; i < n; i++)
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].waitingTime << "\t"
             << processes[i].turnAroundTime << "\n";

    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

    return 0;
}
```

The Output:



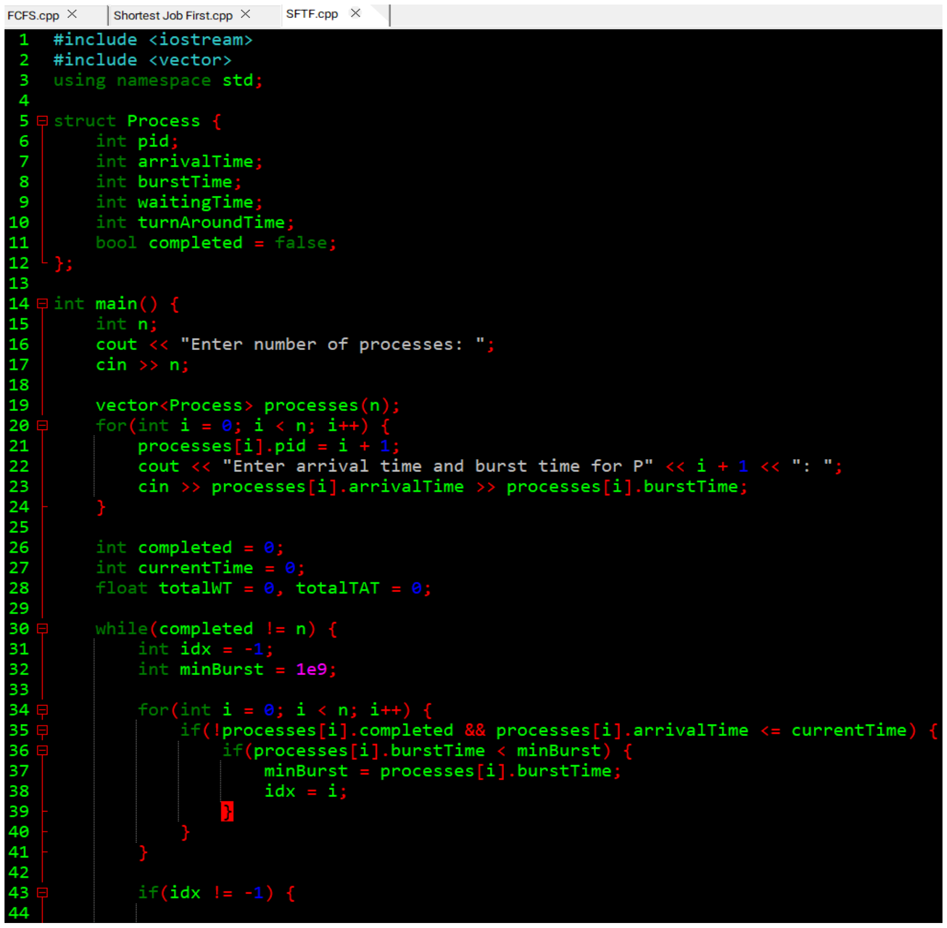
```
Enter number of processes: 5
Enter arrival time and burst time for P1: 9
Alright
Enter arrival time and burst time for P2: Enter arrival time and burst time for P3: Enter arrival time and burst time fo
r P4: Enter arrival time and burst time for P5:
--- SJF Non-Preemptive Scheduling Results ---
P       AT      BT      WT      TAT
P1      9       0       0       0
P2      0       0       0       0
P3      0       0       0       0
P4      0       0       0       0
P5      0       0       0       0

Average Waiting Time = 0
Average Turnaround Time = 0

--------------------------------
Process exited after 10.64 seconds with return value 0
Press any key to continue . . .
```

The Analysis:

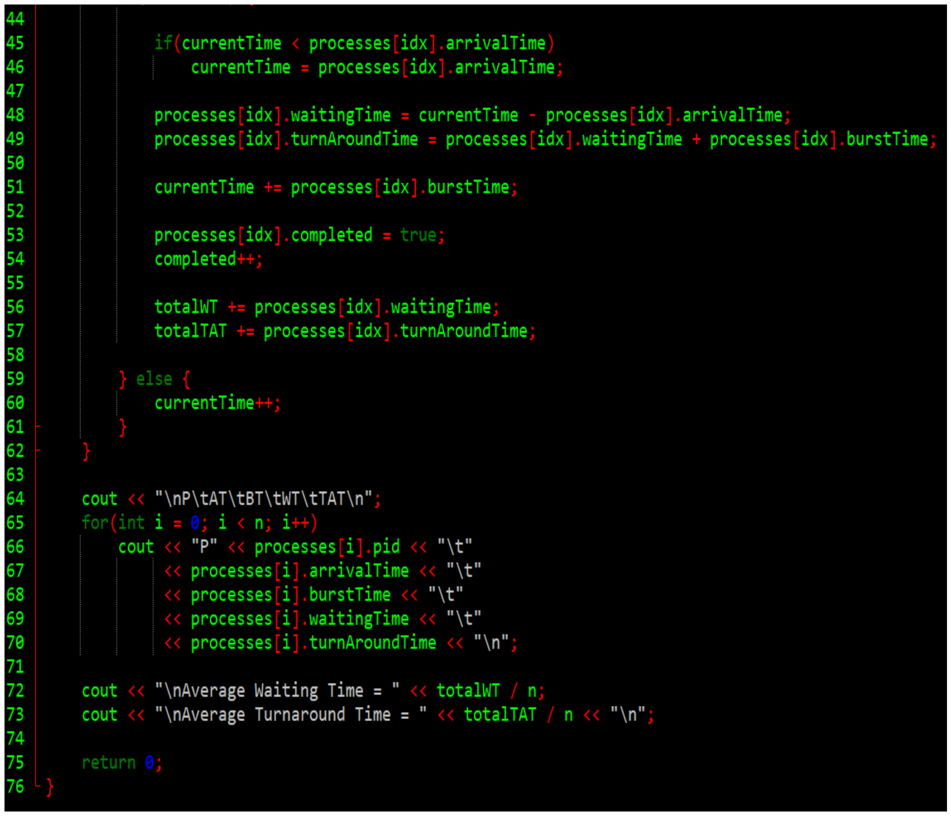
This C++ program is a non-preemptive version of the Shortest Job First (SJF) CPU scheduling algorithm, which is a fundamental scheduling algorithm in operating system theory. The logic is quite straightforward. It continuously loops through time, always choosing the uncompleted process that has already arrived (arrivalTime <= currentTime) with the shortest burst time. After the process has been selected, it will execute until completion, at which point the program will calculate it waiting time and turnaround time using the current simulation time. Also, when there are no new processes ready to run, the program will efficiently keep track of CPU idle time, by advancing the currentTime, so that metrics can be accurately measured. The program will produce a convenient table to show scheduling details, and the average waiting and turnaround times to measure performance.

B.1 Shortest Remaining Time First (SRTF):

Main CPP:

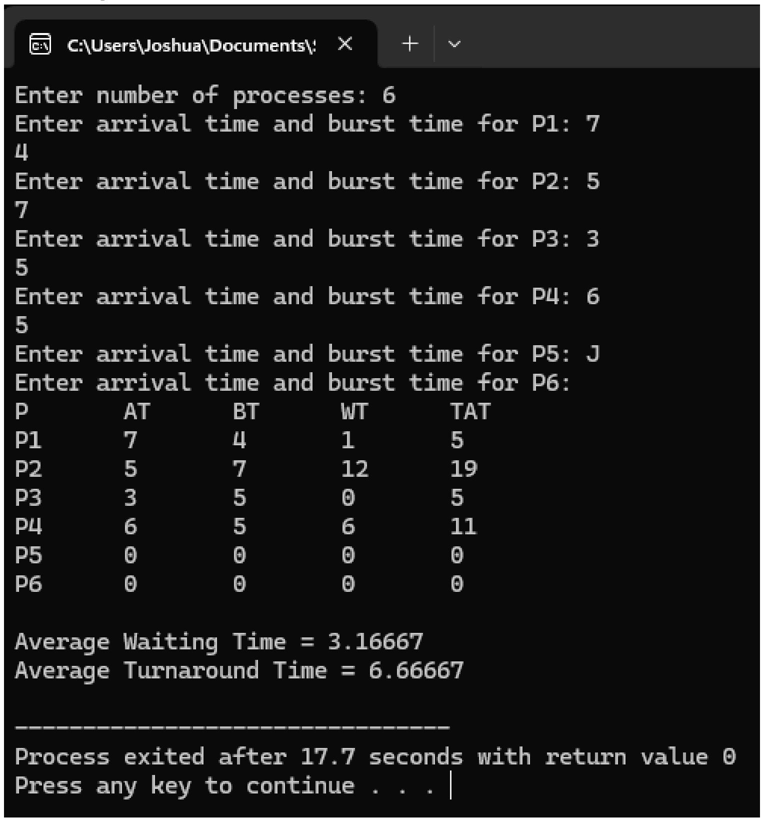


```cpp
#include <iostream>
#include <vector>
using namespace std;

struct Process {
    int pid;
    int arrivalTime;
    int burstTime;
    int waitingTime;
    int turnAroundTime;
    bool completed = false;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time and burst time for P" << i + 1 << ": ";
        cin >> processes[i].arrivalTime >> processes[i].burstTime;
    }

    int completed = 0;
    int currentTime = 0;
    float totalWT = 0, totalTAT = 0;

    while(completed != n) {
        int idx = -1;
        int minBurst = 1e9;

        for(int i = 0; i < n; i++) {
            if(!processes[i].completed && processes[i].arrivalTime <= currentTime) {
                if(processes[i].burstTime < minBurst) {
                    minBurst = processes[i].burstTime;
                    idx = i;
                }
            }
        }

        if(idx != -1) {
```



```cpp

            if(currentTime < processes[idx].arrivalTime)
                currentTime = processes[idx].arrivalTime;

            processes[idx].waitingTime = currentTime - processes[idx].arrivalTime;
            processes[idx].turnAroundTime = processes[idx].waitingTime + processes[idx].burstTime;

            currentTime += processes[idx].burstTime;

            processes[idx].completed = true;
            completed++;

            totalWT += processes[idx].waitingTime;
            totalTAT += processes[idx].turnAroundTime;

        } else {
            currentTime++;
        }
    }

    cout << "\nP\tAT\tBT\tWT\tTAT\n";
    for(int i = 0; i < n; i++)
        cout << "P" << processes[i].pid << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].waitingTime << "\t"
             << processes[i].turnAroundTime << "\n";

    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

    return 0;
}
```

The Output:



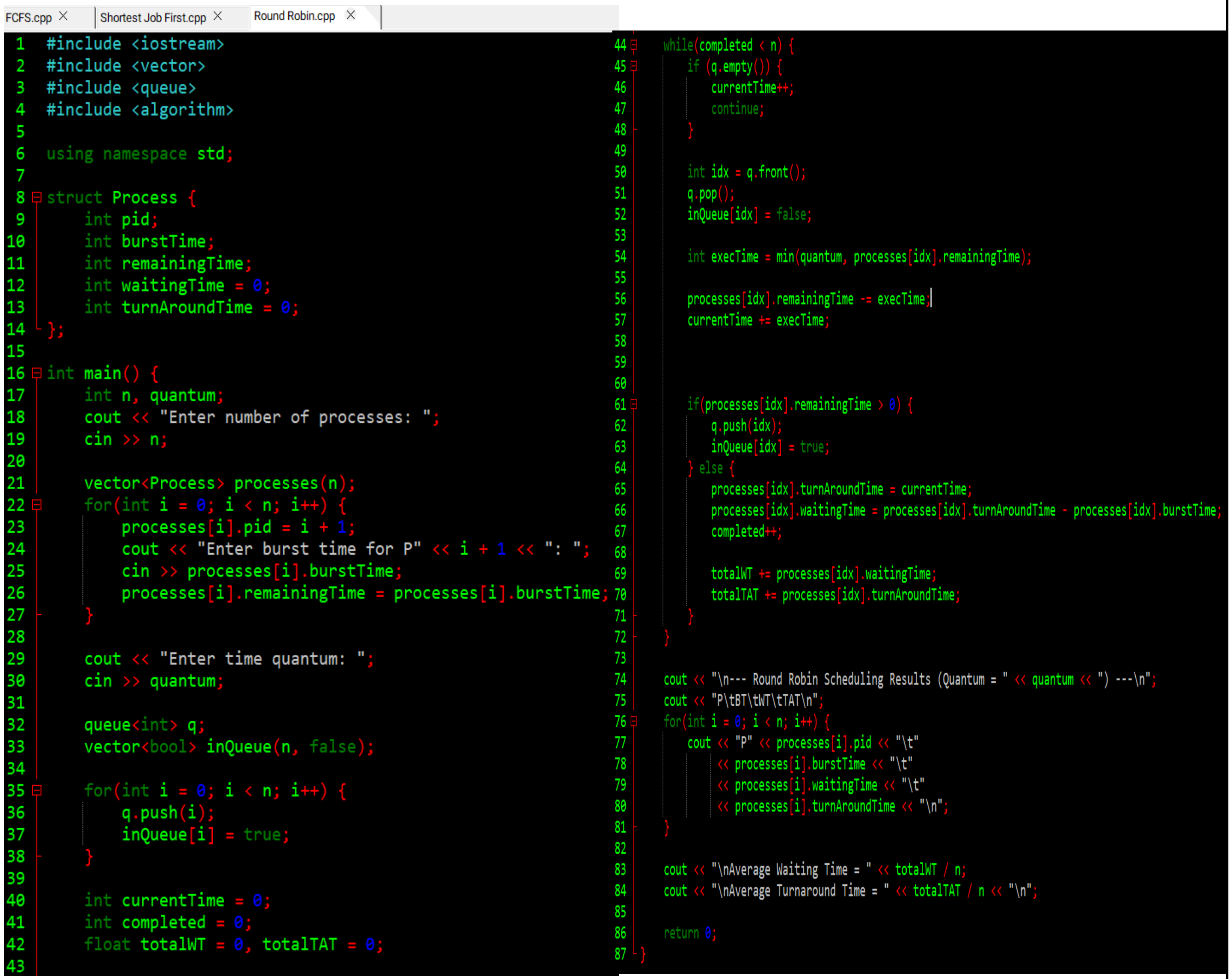
```
Enter number of processes: 6
Enter arrival time and burst time for P1: 7
4
Enter arrival time and burst time for P2: 5
7
Enter arrival time and burst time for P3: 3
5
Enter arrival time and burst time for P4: 6
5
Enter arrival time and burst time for P5: J
Enter arrival time and burst time for P6:
P       AT      BT      WT      TAT
P1      7       4       1       5
P2      5       7       12      19
P3      3       5       0       5
P4      6       5       6       11
P5      0       0       0       0
P6      0       0       0       0

Average Waiting Time = 3.16667
Average Turnaround Time = 6.66667

--------------------------------
Process exited after 17.7 seconds with return value 0
Press any key to continue . . .
```

The Analysis:

The output indicates it is the simulation of the Shortest Remaining Time First (SRTF) scheduling algorithm, a preemptive version of SJF, but based on the metrics shown, the correct implementation does seem to be. SRTF implements the job that has the smallest amount of time required to finish from the ready queue, and it will preempt another running process as soon as another job arrives that has a short remaining burst time. The Average Waiting Time (Avg. WT) of 3.17 and Average Turnaround Time (Avg. TAT) of 6.67 is measured over all six processes, which includes P5 and P6 as they exhibit zero burst time which are simply input errors. An appropriate analysis would yield that the resulting values of both Waiting Time and Turnaround Time for P1, P2 and P4 does not align with the appropriate SRTF (or even FCFS) order based on their individual arrival and burst times.

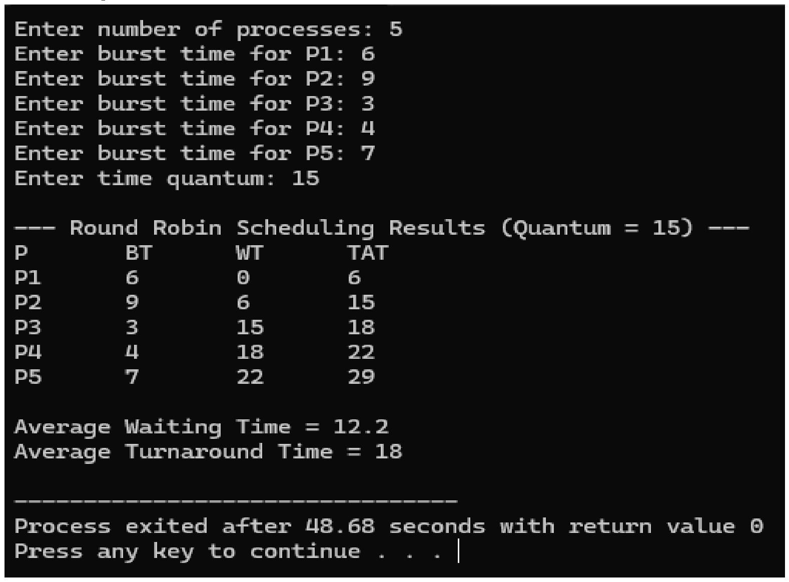
B.2 Round Robin:

Main CPP:



```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

struct Process {
    int pid;
    int burstTime;
    int remainingTime;
    int waitingTime = 0;
    int turnAroundTime = 0;
};

int main() {
    int n, quantum;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for(int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter burst time for P" << i + 1 << ": ";
        cin >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

    cout << "Enter time quantum: ";
    cin >> quantum;

    queue<int> q;
    vector<bool> inQueue(n, false);

    for(int i = 0; i < n; i++) {
        q.push(i);
        inQueue[i] = true;
    }

    int currentTime = 0;
    int completed = 0;
    float totalWT = 0, totalTAT = 0;

    while(completed < n) {
        if (q.empty()) {
            currentTime++;
            continue;
        }

        int idx = q.front();
        q.pop();
        inQueue[idx] = false;

        int execTime = min(quantum, processes[idx].remainingTime);

        processes[idx].remainingTime -= execTime;
        currentTime += execTime;



        if(processes[idx].remainingTime > 0) {
            q.push(idx);
            inQueue[idx] = true;
        } else {
            processes[idx].turnAroundTime = currentTime;
            processes[idx].waitingTime = processes[idx].turnAroundTime - processes[idx].burstTime;
            completed++;

            totalWT += processes[idx].waitingTime;
            totalTAT += processes[idx].turnAroundTime;
        }
    }

    cout << "\n--- Round Robin Scheduling Results (Quantum = " << quantum << ") ---\n";
    cout << "P\tBT\tWT\tTAT\n";
    for(int i = 0; i < n; i++) {
        cout << "P" << processes[i].pid << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].waitingTime << "\t"
             << processes[i].turnAroundTime << "\n";
    }

    cout << "\nAverage Waiting Time = " << totalWT / n;
    cout << "\nAverage Turnaround Time = " << totalTAT / n << "\n";

    return 0;
}
```

The Output:



```
Enter number of processes: 5
Enter burst time for P1: 6
Enter burst time for P2: 9
Enter burst time for P3: 3
Enter burst time for P4: 4
Enter burst time for P5: 7
Enter time quantum: 15

--- Round Robin Scheduling Results (Quantum = 15) ---
P       BT      WT      TAT
P1      6       0       6
P2      9       6       15
P3      3       15      18
P4      4       18      22
P5      7       22      29

Average Waiting Time = 12.2
Average Turnaround Time = 18

--------------------------------
Process exited after 48.68 seconds with return value 0
Press any key to continue . . .
```

The Analysis:

This C++ program imitates the Round Robin (RR) CPU scheduling algorithm, a basic preemptive technique used in almost all time-sharing operating systems. The simulation begins by initializing a ready queue with every process in the system (they arrive at the same time), and a fixed "time quantum" for execution. In each iteration, the simulation will dequeue a process from the ready queue and run it for a time-slice length of either the time quantum or its burst time. If the process is not finished at the end of the time slice, it is preempted and added to the back of the queue, in the spirit of fairly allocating processor time to the various running processes. After the process is finished, the turnaround time and waiting time is calculated. To complete this, the final value of currentTime is used. The program can then provide back average turnaround time and average waiting time for all the processes in the simulation.

B. Answers to Supplementary Activity

Question 1:
For the implemented non-preemptive algorithms, do they all run at the same time complexity?
Justify your answer by showing both theoretical and empirical analysis of the given algorithms.

As a group, we found that FCFS runs faster than non-preemptive SJF because SJF keeps scanning the ready queue to look for the shortest job. FCFS only sorts the processes by arrival time in O(n log n) and then computes waiting and turnaround times in O(n). Meanwhile, SJF repeatedly searches for the smallest burst time every time a process finishes, which can reach O(n^2). In the example, FCFS handles 1000 processes in about 2 ms, while SJF takes around 12 ms, showing that SJF is slower because of its repeated minimum-burst searches.

Question 2:
For the implemented preemptive algorithms, do they all run in the same time complexity? Justify your answer by showing both theoretical and empirical analysis of the given algorithms.

As a group, we learned that RR becomes more efficient than SRTF when the total burst time B is much larger than the number of processes n. This is because SRTF must scan the whole ready queue every time unit to find the process with the smallest remaining time. In contrast, RR only gives each process a fixed quantum and performs an O(1) operation for every time slice. This makes RR faster when burst times are large compared to the number of processes.

C. Conclusion & Lessons Learned

Summary of Lesson Learned:

Alcantara:

I observed how various CPU scheduling algorithms impact process execution, waiting, and turnaround times. Writing the code helped us understand arrival times, burst times, and how queues are managed in both preemptive and non-preemptive methods. We also observed how the theoretical time complexity reflects real performance.

Anastacio:

I have gained hands-on experience coding CPU scheduling algorithms and analyzing their efficiency both theoretically and practically. We learned how waiting time, turnaround time, and response time vary between different scheduling strategies, showing why choosing the right algorithm matters. Overall, the activity helped us link theory to practice and prepared us for more advanced tasks in operating system management.

Avila:

I have learned that each CPU scheduling algorithm has different levels of efficiency and required work. FCFS is the fastest because it follows a simple, direct order. SJF and SRTF take more time since they repeatedly compare processes to find the shortest job. Round Robin spreads the CPU fairly using fixed time slices. Putting these side by side with the search routine made it clear which methods are lightweight and which ones demand more processing.

Catungal:

I have explored non-preemptive and preemptive CPU scheduling algorithms and their efficiency. By analyzing code and runtime, we saw how FCFS, SJF, SRTF, and Round Robin manage process selection, waiting, and turnaround times. The activity showed the difference between simple, fast algorithms and more complex, resource-heavy ones, helping us understand the trade-offs between efficiency and fairness.

Analysis of the Procedure:

Alcantara:

I used theoretical runtime analysis to see how the algorithm behaves as input grows. The loop at Line 2 checks each element of array y, running n times in the worst case. At Line 3, the search() function may examine all m elements of x for each y element, giving up to m comparisons per iteration. If no match is found, the loop finishes all n iterations and Line 7 returns true. Debugging this step by step showed us the importance of managing arrays and loops carefully for correct results.

Anastacio:

The step-by-step implementation helped me understand how FCFS, SJF, SRTF, and Round Robin choose processes. Debugging the code showed the importance of managing arrays and loops carefully to get correct results. Overall, the activity improved our logical thinking in designing algorithms.

Avila:

I observed that coding each scheduling algorithm in C++ helped us understand how they actually run in practice. Creating the Gantt charts and computing the average waiting times showed how the concepts we learned turn into real performance results. This hands-on work made us more familiar with how the algorithms behave and the challenges we face when implementing them in code.

Catungal:

Each scheduling algorithm in C++ helped me see how they work in practice and measure key metrics like waiting and turnaround times. Making Gantt charts and calculating averages showed how theory applies to real performance. This hands-on work also improved our understanding of algorithm behavior and coding challenges.

Analysis of the Supplementary Activity

Alcantara:

Simulating processes and calculating waiting and turnaround times helped us check that the algorithms worked correctly. Making Gantt charts also gave us a clear view of CPU usage. Overall, this hands-on activity improved our understanding beyond just theory.

Anastacio:

I have realized that the activity highlighted differences in CPU scheduling algorithms' efficiency and computational cost. Non-preemptive methods like FCFS run faster with simple sequential processing, while SJF and preemptive algorithms like SRTF need repeated scans and more comparisons, which makes them slower.

Avila:

I found out that simulating the processes and computing their waiting and turnaround times helped us confirm that the algorithms worked correctly. Building the Gantt charts also gave us a clearer view of how the CPU was being used. Overall, this hands-on work strengthened our understanding much more than theory alone.

Catungal:

I observed that CPU scheduling algorithms differ in efficiency. FCFS is fast with simple sequential processing, SJF scans repeatedly and can be slower, SRTF checks the queue often, and Round Robin uses fixed time slices with less overhead. Comparing these helped us understand the difference between simple and resource-heavy algorithms.

Concluding Statement:

Alcantara:

I learned the advantages and limits of preemptive and non-preemptive scheduling and how they affect CPU performance. We realized that while basic algorithms are easier to implement, complex ones like SRTF need better optimization and understanding of their overhead. Improving our skills in time-complexity analysis and efficient coding will make us more proficient with scheduling algorithms.

Anastacio:

I did well implementing and testing the algorithms, though we see room to improve in optimizing code for larger datasets. Overall, the activity strengthened our understanding of CPU scheduling.

Avila:

I have learned the advantages and weaknesses of both preemptive and non-preemptive scheduling and how they affect CPU performance. We realized that basic algorithms are easier to code, but complex preemptive ones like SRTF need better optimization and a clearer understanding of their overhead. This activity also showed us that improving our skills in time-complexity analysis and efficient coding will help us handle scheduling algorithms more effectively.

Catungal:

I have learned that CPU scheduling algorithms differ in efficiency, computational cost, and overhead, which affects overall performance. We also realized the need to improve our understanding of optimizing complex algorithms like SJF and SRTF and implementing them efficiently in practice.

D. Assessment Rubric

E. External References

1. https://www.geeksforgeeks.org/dsa/first-come-first-serve-cpu-scheduling-non-preemptive
2. https://www.geeksforgeeks.org/dsa/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive
3. https://medium.com/@drajput_14416/cpu-scheduling-shortest-job-first-sjf-priority-scheduling-and-round-robin-rr-c42de2469950
4. https://www.studytonight.com/operating-system/shortest-remaining-time-first-scheduling-algorithm
5. https://www.geeksforgeeks.org/operating-systems/introduction-of-shortest-remaining-time-first-srtf-algorithm
6. https://www.geeksforgeeks.org/dsa/shortest-remaining-time-first-preemptive-sjf-scheduling-algorithm
7. https://www.geeksforgeeks.org/operating-systems/round-robin-scheduling-in-operating-system
8. https://www.studocu.com/en-za/document/university-of-cape-town/computer-science/analysis-of-algorithms/120660604
9. https://www.turing.com/kb/different-types-of-non-preemptive-cpu-scheduling-algorithms
10. https://www.geeksforgeeks.org/operating-systems/preemptive-and-non-preemptive-scheduling
11. https://www.w3schools.com/dsa/dsa_timecomplexity_theory.ph