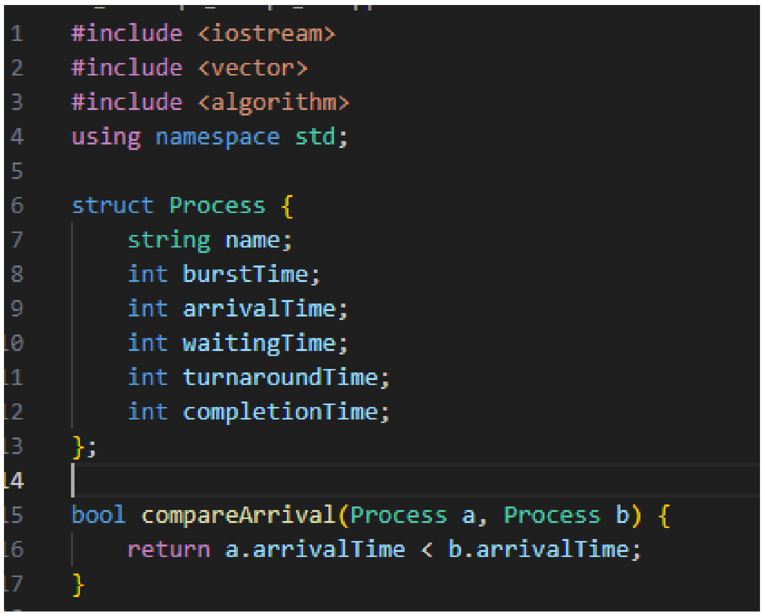
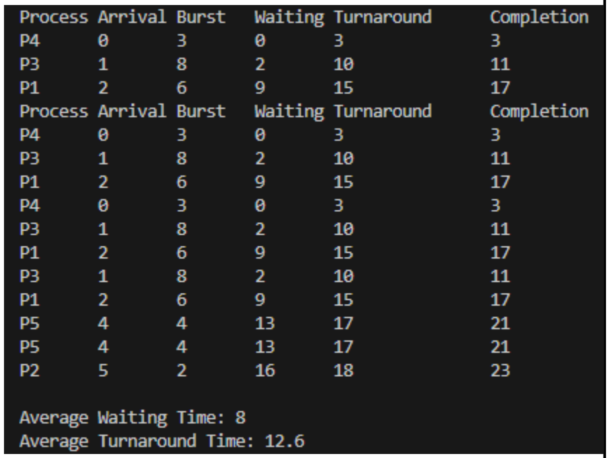
Hands-on Activity 15.1	
Hands-on Activity 15.1 Scheduling Algorithms in C++	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: Nov. 11, 2025**
Section: CPE21S4	**Date Submitted: Nov. 13, 2025**
Name(s): **Crishen Luper S. Pulgado** **Nikko Francis I. Andoy** **David Owen Santiago** **Heidee Punay**	**Instructor:** Engr. Jimlord Quejado

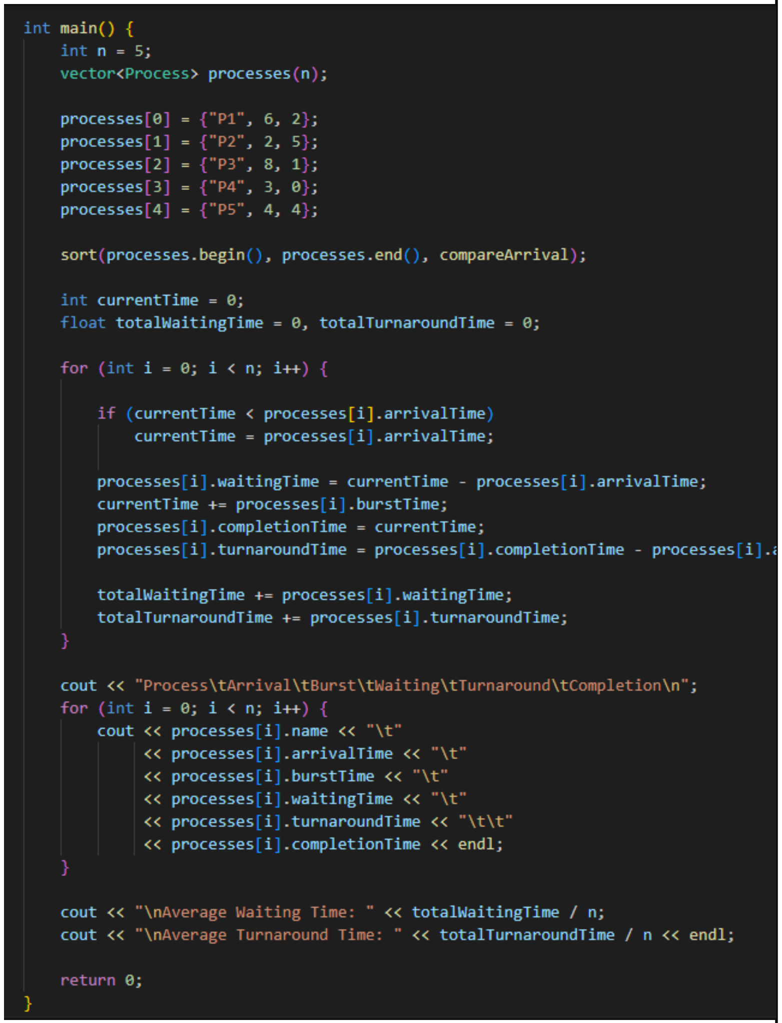
A. Output(s) and Observation(s)

ILO A.1

Code	Output
	

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

struct Process {
    string name;
    int burstTime;
    int arrivalTime;
    int waitingTime;
    int turnaroundTime;
    int completionTime;
};

bool compareArrival(Process a, Process b) {
    return a.arrivalTime < b.arrivalTime;
}
```

```
Process Arrival Burst   Waiting Turnaround      Completion
P4      0       3       0       3               3
P3      1       8       2       10              11
P1      2       6       9       15              17
Process Arrival Burst   Waiting Turnaround      Completion
P4      0       3       0       3               3
P3      1       8       2       10              11
P1      2       6       9       15              17
P4      0       3       0       3               3
P3      1       8       2       10              11
P1      2       6       9       15              17
P3      1       8       2       10              11
P1      2       6       9       15              17
P5      4       4       13      17              21
P5      4       4       13      17              21
P2      5       2       16      18              23

Average Waiting Time: 8
Average Turnaround Time: 12.6
```



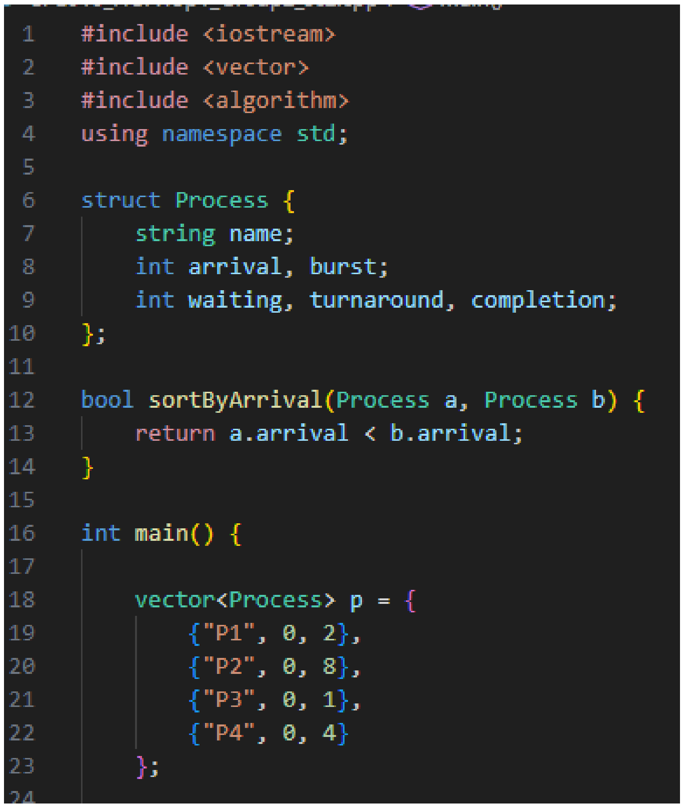
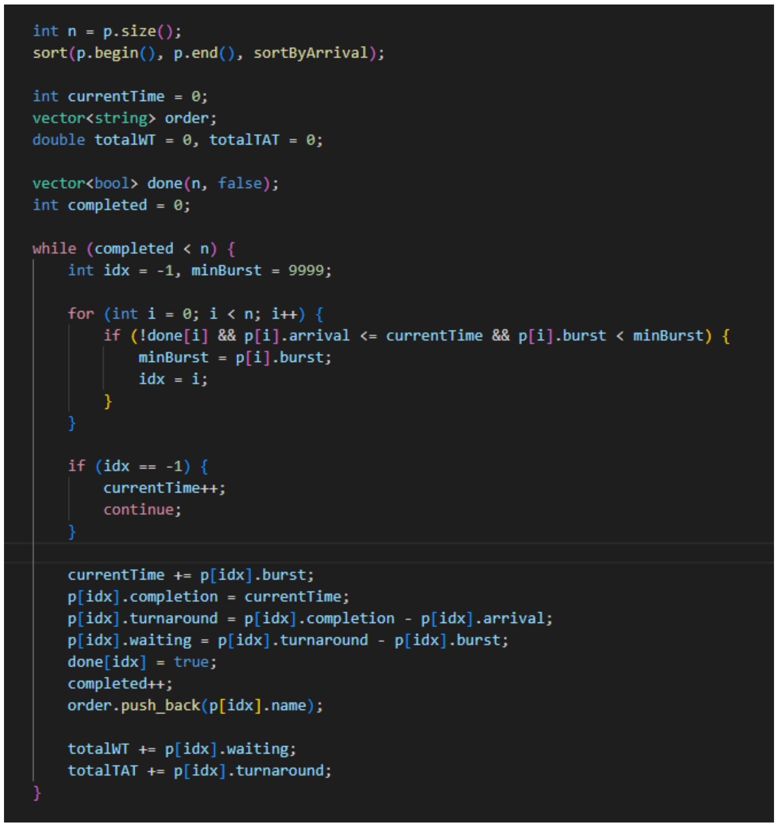
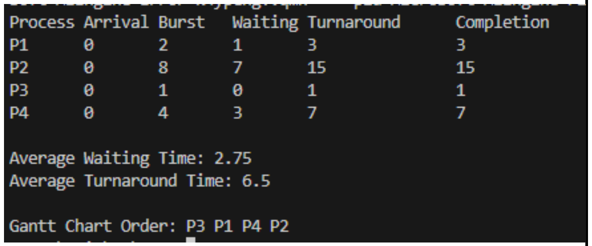
```
int main() {
    int n = 5;
    vector<Process> processes(n);

    processes[0] = {"P1", 6, 2};
    processes[1] = {"P2", 2, 5};
    processes[2] = {"P3", 8, 1};
    processes[3] = {"P4", 3, 0};
    processes[4] = {"P5", 4, 4};

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

    int currentTime = 0;
    float totalWaitingTime = 0, totalTurnaroundTime = 0;

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

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

        processes[i].waitingTime = currentTime - processes[i].arrivalTime;
        currentTime += processes[i].burstTime;
        processes[i].completionTime = currentTime;
        processes[i].turnaroundTime = processes[i].completionTime - processes[i].a

        totalWaitingTime += processes[i].waitingTime;
        totalTurnaroundTime += processes[i].turnaroundTime;
    }

    cout << "Process\tArrival\tBurst\tWaiting\tTurnaround\tCompletion\n";
    for (int i = 0; i < n; i++) {
        cout << processes[i].name << "\t"
             << processes[i].arrivalTime << "\t"
             << processes[i].burstTime << "\t"
             << processes[i].waitingTime << "\t"
             << processes[i].turnaroundTime << "\t\t"
             << processes[i].completionTime << endl;
    }

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

    return 0;
}
```

Analysis

Pulgado: For the code, we included a library such as algorithm, which is used for the sorting process of the arrival, vector, which is used for storing processes, arrival time, burst time, and waiting time, and lastly the iomanip for format console. In the process, we manually enter 5 processes with burst time, which is how long it takes to execute, and the arrival time, when each arrives at the CPU queue. After that, we use the sort algorithm so that the earliest arrivals are executed first so that it follows FCFS. Inside the loop, the currentTime keeps track of the current CPU clock. If the CPU is idle, it jumps to the next process's arrival time. The waitingTime is the time spent in the queue before running. The currentTime += burst time adds how long the process runs. The completionTime is when the process finishes and the turnaroundTime is the total time from arrival to completion. So after the sorting by arrival time the order becomes, P4 then P3 then P1 then P5 then lastly is P2. For the computation, the total of waiting time is 2+9+13+16 = 40; for the turnaround time is 3 + 10 + 15 + 17 + 18 = 63; Then using the AWT formula we get 8 which is why the result is 8. The turnaround time is 63/5 which is 12.6.

Punay:
In this part, we implemented the FCFS scheduling algorithm, which is the simplest way to manage processes. The CPU runs each task in the order it arrives, like a queue at a store. In the code, we used a vector to store process details and sorted them by arrival time. Then we calculated waiting time, turnaround time, and completion time for each process. This helped us understand how the CPU handles tasks one by one without switching. The average waiting time was 8, which shows that FCFS is fair but not always the fastest, especially if a long task arrives first.

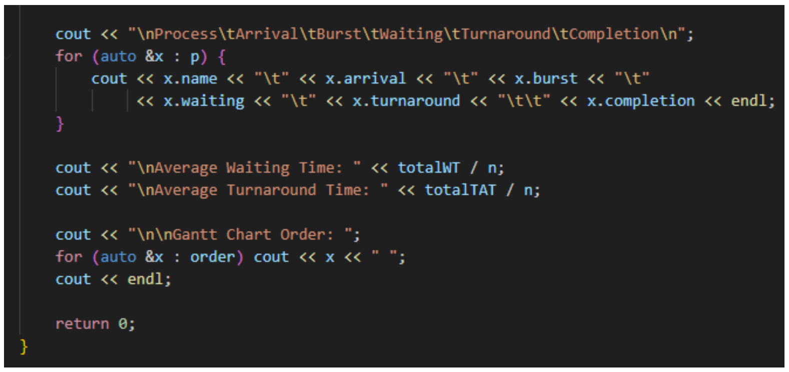
ILO A.2

Code	Output
 	

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

struct Process {
    string name;
    int arrival, burst;
    int waiting, turnaround, completion;
};

bool sortByArrival(Process a, Process b) {
    return a.arrival < b.arrival;
}

int main() {

    vector<Process> p = {
        {"P1", 0, 2},
        {"P2", 0, 8},
        {"P3", 0, 1},
        {"P4", 0, 4}
    };
```

```cpp
int n = p.size();
sort(p.begin(), p.end(), sortByArrival);

int currentTime = 0;
vector<string> order;
double totalWT = 0, totalTAT = 0;

vector<bool> done(n, false);
int completed = 0;

while (completed < n) {
    int idx = -1, minBurst = 9999;

    for (int i = 0; i < n; i++) {
        if (!done[i] && p[i].arrival <= currentTime && p[i].burst < minBurst) {
            minBurst = p[i].burst;
            idx = i;
        }
    }

    if (idx == -1) {
        currentTime++;
        continue;
    }

    currentTime += p[idx].burst;
    p[idx].completion = currentTime;
    p[idx].turnaround = p[idx].completion - p[idx].arrival;
    p[idx].waiting = p[idx].turnaround - p[idx].burst;
    done[idx] = true;
    completed++;
    order.push_back(p[idx].name);

    totalWT += p[idx].waiting;
    totalTAT += p[idx].turnaround;
}
```

```
Process Arrival Burst  Waiting Turnaround      Completion
P1      0       2      1       3               3
P2      0       8      7       15              15
P3      0       1      0       1               1
P4      0       4      3       7               7

Average Waiting Time: 2.75
Average Turnaround Time: 6.5

Gantt Chart Order: P3 P1 P4 P2
```



```
    cout << "\nProcess\tArrival\tBurst\tWaiting\tTurnaround\tCompletion\n";
    for (auto &x : p) {
        cout << x.name << "\t" << x.arrival << "\t" << x.burst << "\t"
             << x.waiting << "\t" << x.turnaround << "\t\t" << x.completion << endl;
    }

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

    cout << "\n\nGantt Chart Order: ";
    for (auto &x : order) cout << x << " ";
    cout << endl;

    return 0;
}
```

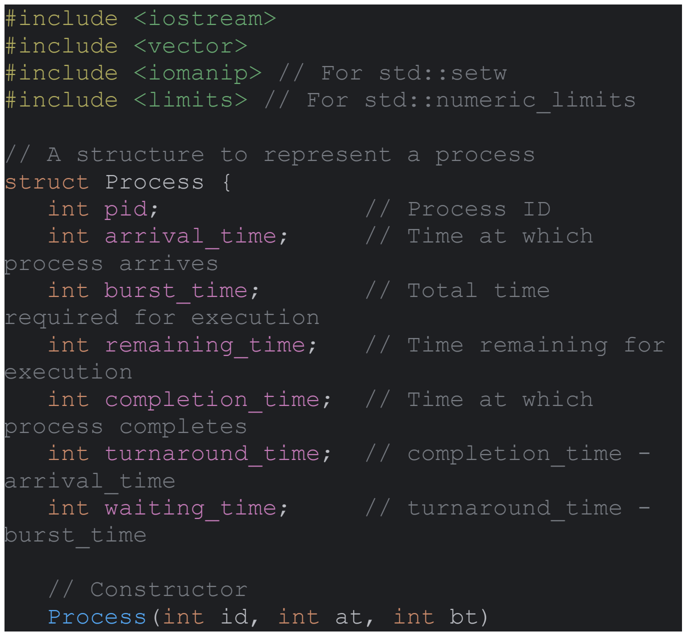
Analysis

Pulgado: Here in this code, we use a vector for storing and an algorithm for the sorting process. So the Process stores the name, arrival, burst, waiting, turnaround, and completion. The sortByArrival function helps the sort process by arrival time using the sort(). The SJF starts checking from the earliest-arriving process. What happens in the sort process is that processes are sorted by arrival time. The currentTime starts at 0 and done keeps track of which processes have finished. The order will store the gantt chart sequence. The CPU looks for the shortest job that has already arrived. If no process has arrived yet, we just increase the time by 1 unit until something arrives. Lastly, it will calculate all once a process is selected. The average waiting time for each process is 1, 7, 0, and 3 which when divided by 4 you'll get 2.75. The SJF chooses the process with the shortest bust time first which is the order is P3 then P4 then P1, and lastly the P2. When we total the TurnAround, 3+15+1+7 divided by 4 we get is 6.5 for the average turnaround time.

Punay: In this part, we worked on the SJF algorithm, which improves performance by choosing the shortest task that's ready. The code checks which process has the smallest burst time and runs it first. This reduces waiting time because short tasks finish quickly. We used a loop to find the shortest job among the arrived processes and tracked the Gantt chart order to see how the CPU scheduled them. The average waiting time dropped to 2.75, showing that SJF is more efficient than FCFS. This part helped us learn how smarter scheduling can make systems faster.

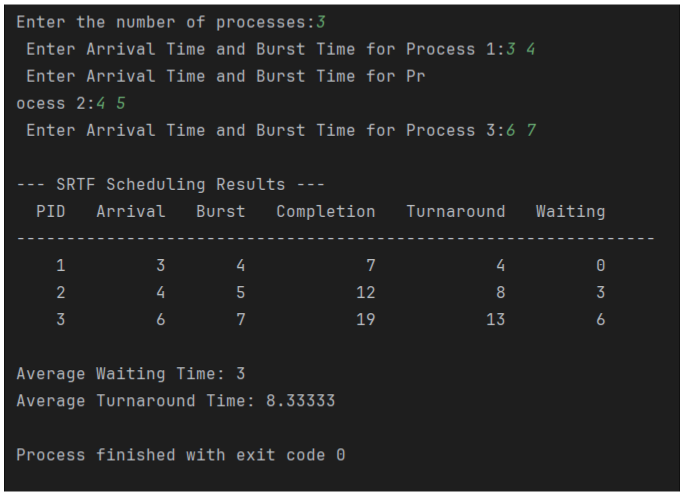
ILO B.1

Code



```
#include <iostream>
#include <vector>
#include <iomanip> // For std::setw
#include <limits> // For std::numeric_limits

// A structure to represent a process
struct Process {
   int pid;                // Process ID
   int arrival_time;       // Time at which
process arrives
   int burst_time;         // Total time
required for execution
   int remaining_time;     // Time remaining for
execution
   int completion_time;    // Time at which
process completes
   int turnaround_time;    // completion_time -
arrival_time
   int waiting_time;       // turnaround_time -
burst_time

   // Constructor
   Process(int id, int at, int bt)
```

Output



```
Enter the number of processes:3
 Enter Arrival Time and Burst Time for Process 1:3 4
 Enter Arrival Time and Burst Time for Pr
ocess 2:4 5
 Enter Arrival Time and Burst Time for Process 3:6 7

--- SRTF Scheduling Results ---
  PID   Arrival   Burst   Completion   Turnaround   Waiting
-------------------------------------------------------------
    1         3      4            7            4         0
    2         4      5           12            8         3
    3         6      7           19           13         6

Average Waiting Time: 3
Average Turnaround Time: 8.33333

Process finished with exit code 0
```

```cpp
       : pid(id), arrival_time(at),
burst_time(bt),
         remaining_time(bt),
completion_time(0),
         turnaround_time(0), waiting_time(0)
{}
};

// Function to print the results in a table
void printResults(const std::vector<Process>&
processes, int n) {
   double total_waiting_time = 0;
   double total_turnaround_time = 0;

   std::cout << "\n--- SRTF Scheduling Results
---\n";
   std::cout << std::setw(5) << "PID"
             << std::setw(10) << "Arrival"
             << std::setw(8) << "Burst"
             << std::setw(13) << "Completion"
             << std::setw(13) << "Turnaround"
             << std::setw(10) << "Waiting" <<
"\n";
   std::cout <<
"----------------------------------------------
-------------------\n";

   for (int i = 0; i < n; ++i) {
       std::cout << std::setw(5) <<
processes[i].pid
                 << std::setw(10) <<
processes[i].arrival_time
                 << std::setw(8) <<
processes[i].burst_time
                 << std::setw(13) <<
processes[i].completion_time
                 << std::setw(13) <<
processes[i].turnaround_time
                 << std::setw(10) <<
processes[i].waiting_time << "\n";

       total_waiting_time +=
processes[i].waiting_time;
       total_turnaround_time +=
processes[i].turnaround_time;
   }

   std::cout << "\n";
   std::cout << "Average Waiting Time: " <<
(total_waiting_time / n) << "\n";
   std::cout << "Average Turnaround Time: " <<
(total_turnaround_time / n) << "\n";
}

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

   std::vector<Process> processes;
   processes.reserve(n); // Allocate space

   for (int i = 0; i < n; ++i) {
       int arrival, burst;
       std::cout << "Enter Arrival Time and
Burst Time for Process " << (i + 1) << ": ";
       std::cin >> arrival >> burst;
       processes.emplace_back(i + 1, arrival,
burst);
   }

   int completed_processes = 0;
   int current_time = 0;
   int max_int =
std::numeric_limits<int>::max();

   // Main simulation loop
   // Continues as long as not all processes
are completed
   while (completed_processes < n) {
       int shortest_job_index = -1;
       int min_remaining_time = max_int;

       // Find the process with the minimum
remaining time *that has arrived*
       for (int i = 0; i < n; ++i) {
           // Check if process has arrived and
is not yet complete
           if (processes[i].arrival_time <=
current_time && processes[i].remaining_time >
0) {
               // Check if this process has a
shorter remaining time
               if (processes[i].remaining_time
< min_remaining_time) {
                   min_remaining_time =
processes[i].remaining_time;
                   shortest_job_index = i;
               }
           }
       }

       if (shortest_job_index == -1) {
           // No process is available to run
(CPU is idle)
           // This happens if all arrived
processes are done, but new ones haven't
arrived yet
           current_time++;
       } else {
           // A process is found, execute it
for one time unit

processes[shortest_job_index].remaining_time--
;
           current_time++; // Advance the
system clock

           // Check if the process has
completed execution
           if
(processes[shortest_job_index].remaining_time
== 0) {
               completed_processes++;

               // Set completion time and
calculate other metrics

processes[shortest_job_index].completion_time
= current_time;

processes[shortest_job_index].turnaround_time
=

processes[shortest_job_index].completion_time
- processes[shortest_job_index].arrival_time;
processes[shortest_job_index].waiting_time =

processes[shortest_job_index].turnaround_time
- processes[shortest_job_index].burst_time;
            }
        }
    }

    // All processes are done, print the final
results
    printResults(processes, n);

    return 0;
}
```

Analysis

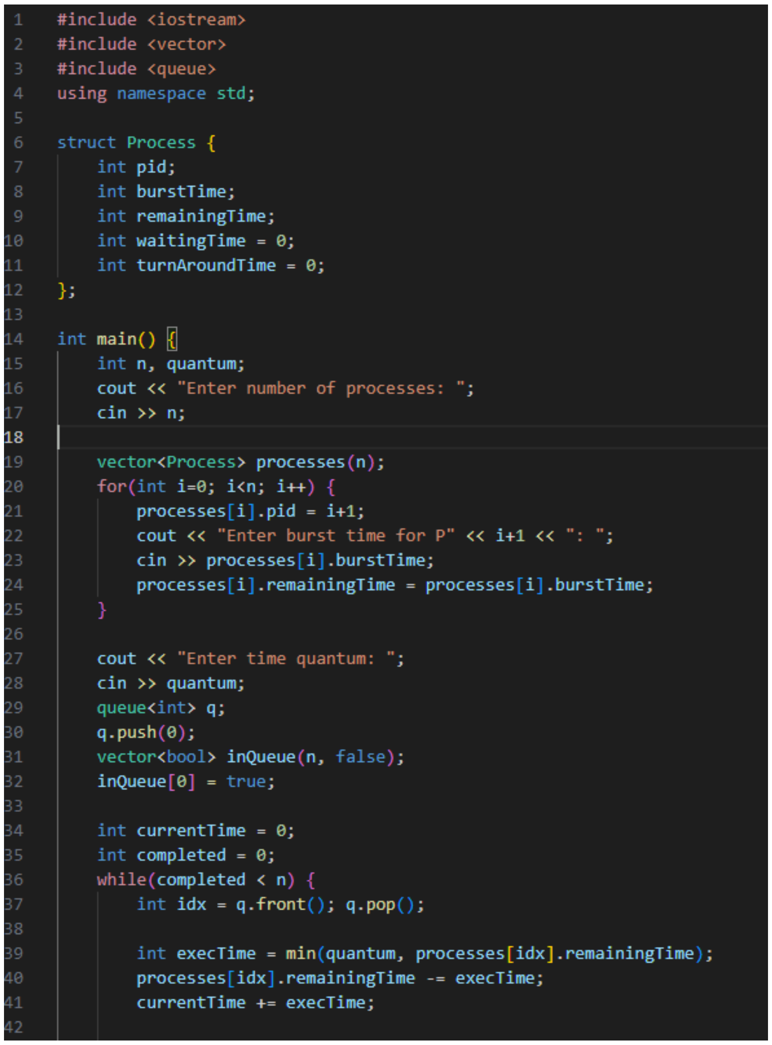
The output shows how the code simulates the SRTF concept using its current_time variable as a system clock. Initially, the CPU is idle because no processes have arrived. At T=2, P2 arrives and begins running immediately, as it is the only process available. This is why its final waiting_time is 0.

The first key event happens at T=4, when P3 (Burst=3) arrives. At this exact moment, P2 is already running and has a remaining_time of only 1. The code, following the SRTF rule, compares P2's remaining time (1) with P3's total time (3). Since 1 is less than 3, P2 is allowed to continue running and is not preempted. P3 is forced to wait for one time unit (from T=4 to T=5) until P2 finishes. This single unit of waiting is what results in P3's waiting_time of 1.

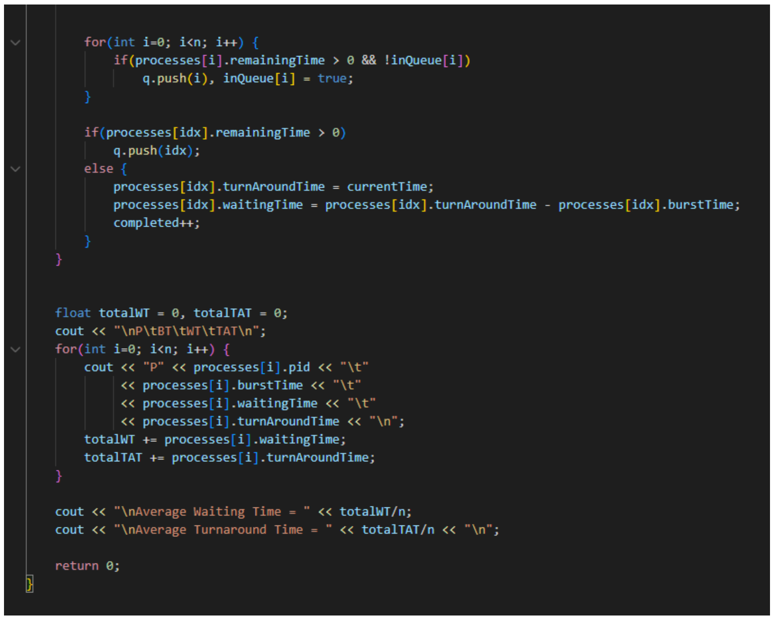
A very similar event occurs when P3 is running and P1 (Burst=3) arrives at T=7. At that moment, P3 has a remaining_time of 1. The code again compares the running process (P3, RT=1) with the new process (P1, RT=3) and sticks with P3 because it's closer to finishing. P1 is forced to wait for one time unit (from T=7 to T=8) until P3 completes, which explains P1's waiting_time of 1. This output clearly shows that even though the algorithm is preemptive, it will only preempt a running process if the newly arriving process has a shorter burst time than the remaining time of the one that is currently running.

ILO B.2

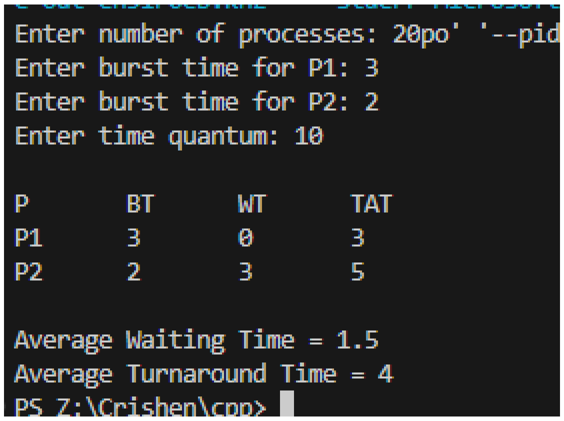
Code	Output



```cpp
#include <iostream>
#include <vector>
#include <queue>
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;
    q.push(0);
    vector<bool> inQueue(n, false);
    inQueue[0] = true;

    int currentTime = 0;
    int completed = 0;
    while(completed < n) {
        int idx = q.front(); q.pop();

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

```



```cpp
        for(int i=0; i<n; i++) {
            if(processes[i].remainingTime > 0 && !inQueue[i])
                q.push(i), inQueue[i] = true;
        }

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


    float totalWT = 0, totalTAT = 0;
    cout << "\nP\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";
        totalWT += processes[i].waitingTime;
        totalTAT += processes[i].turnAroundTime;
    }

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

    return 0;
}
```



```
Enter number of processes: 20po' '--pid
Enter burst time for P1: 3
Enter burst time for P2: 2
Enter time quantum: 10

P       BT      WT      TAT
P1      3       0       3
P2      2       3       5

Average Waiting Time = 1.5
Average Turnaround Time = 4
PS Z:\Crishen\cpp> 
```

Analysis

Pulgado: Under this code, a process is defined by a structure that has a process ID, burst time, internal time, waiting time, and turnaround time. The user would then key in the number of processes, their burst times and a constant time quantum that would dictate the maximum duration that each process would take before termination. The order of executing the processes is then handled by a queue where all processes can have an equal right to use the CPU. The program repeatedly repeats the queue with each process running throughout the entire quantum or until finished. Individual turnaround and waiting times of a process are determined when a process has completed and the overall time it has taken is taken into consideration. Lastly, the program provides the burst-time, waiting-time and turnaround-time of all processes and the average waiting and turnaround-time of all the processes.

B. Answers to Supplementary Activity

Question 1: For the implemented non-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.
Yes, the non-preemptive algorithm follows FCFS and SJF to generally run with similar time complexity. Both used sorting and looping, which gives them a time complexity of O(n log n) for sorting and O(n) for calculating waiting and turnaround times. In FCFS, we sort the processes by arrival time and process them in order, while in SJF, we sort by burst time among the arrived processes. When FCFS code is run, the average of waiting time is 8, and for SJF, it is 2.75, showing that SJF performs better in terms of scheduling efficiency.

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.
For the preemptive algorithm, SRTF does not run with the same time complexity as the non-preemptive ones. In Srtf

C. Conclusion & Lessons Learned

Santiago

In conclusion, preemptive and non-preemptive algorithms are core algorithm scheduling techniques that a CPU uses to complete its processes. A preemptive algorithm can halt its processes if a different process that can be completed faster is suddenly added to the queue. This optimization allows the CPU to prioritize shorter and simpler tasks allowing more tasks to be completed in total. On the other hand, non-preemptive algorithms cannot suddenly stop processes that are currently running; it completes one process at a time. In general, preemptive algorithms are more flexible because the CPU can optimize its performance through different prioritizations leading to a higher productivity because of more tasks completed. However, the single-process-at-a-time nature of non-preemptive algorithms are much easier to track and predict due to the fact that only one process is running at a time. Preemptive algorithms are still more flexible, reliable, and CPU efficient compared to the latter, but it requires more complex codes and algorithms which makes its implementation much harder compared to non-preemptive algorithms.

Pulgado:
This activity examined CPU scheduling algorithms, emphasizing both non-preemptive and preemptive approaches. computational performance and execution effectiveness. Through the use of theoretical runtime evaluation and sequential code analysis During the examination, I noticed how algorithms such as FCFS, SJF, SRTF, and Round Robin manage the selection of processes, delays, and response durations. The research emphasized the difference between straightforward, low-cost algorithms and more intricate, resource-demanding methods, offering a transparent insight into the compromises in effectiveness and equity.

D. Assessment Rubric

E. External References