PRACTICAL –02

AIM – To implement FCFS Algorithm in C/C++

CODE -

```
#include <iostream>

#include <iomanip>

#include <vector>


using namespace std;


struct Process {

    int pid;

    int arrival_time;

    int burst_time;

    int completion_time;

    int turn_around_time;

    int waiting_time;

};


void findCompletionTime(Process P[], int n) {

    P[0].completion_time = P[0].burst_time;

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

        P[i].completion_time = P[i - 1].completion_time + P[i].burst_time;

    }

}


void findTurnAroundTime(Process P[], int n) {

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

```
        P[i].turn_around_time = P[i].completion_time - P[i].arrival_time;
    }
}

void findWaitingTime(Process P[], int n) {
    P[0].waiting_time = 0;
    for(int i = 1; i < n; i++) {
        P[i].waiting_time = P[i].turn_around_time - P[i].burst_time;
    }
}
void findFCFS(Process P[], int n) {
    findCompletionTime(P, n);
    findTurnAroundTime(P, n);
    findWaitingTime(P, n);
}

void printFCFS(Process P[], int n) {
    cout << "FCFS SCHEDULING ALGORITHM: " << endl;
    cout << "PID\tArrival Time\tBurst Time\tCompletion Time\tTurn Around Time\tWaiting 
Time" << endl;
    for(int i = 0; i < n; i++) {
        cout << P[i].pid << "\t\t" << P[i].arrival_time << "\t\t" << P[i].burst_time << "\t\t" << 
P[i].completion_time << "\t\t" << P[i].turn_around_time << "\t\t" << P[i].waiting_time << endl;
    }
}
 void printGanttChart(Process P[], int n) {
    cout << "\nGantt Chart:" << endl;
    cout << "+------+";
```

```
    for(int i = 0; i < n; i++) {
        cout << "------";
    }
    cout << "+" << endl << "|";
    for(int i = 0; i < n; i++) {
        cout << "  P" << P[i].pid << "   |";
    }
    cout << endl << "+------+";
    for(int i = 0; i < n; i++) {
        cout << "------";
    }
    cout << "+" << endl;
}


int main() {
    int n = 5;
    Process P[n] = {{1, 0, 5}, {2, 1, 3}, {3, 2, 2}, {4, 3, 4}, {5, 4, 1}};
    findFCFS(P, n);
    printFCFS(P, n);
    printGanttChart(P, n);

    // Calculate average turn-around-time
    float total_turnaround_time = 0;
    for(int i = 0; i < n; i++) {
        total_turnaround_time += P[i].turn_around_time;
    }
    float average_turnaround_time = total_turnaround_time / n;
```

```
    cout << "Average Turnaround Time: " << average_turnaround_time << endl;

    // Calculate average waiting time
    float total_waiting_time = 0;
    for(int i = 0; i < n; i++) {
        total_waiting_time += P[i].waiting_time;
    }
    float average_waiting_time = total_waiting_time / n;
    cout << "Average Waiting Time: " << average_waiting_time << endl;

    // Calculate Scheduling Length
    int scheduling_length = P[n - 1].completion_time;
    cout << "Scheduling Length: " << scheduling_length << " time units" << endl;

    // Calculate Throughput
    float throughput = (float)n / scheduling_length;
    cout << "Throughput: " << throughput << " processes per time unit" << endl;

    return 0;
}
```

OUTPUT -

```
FCFS SCHEDULING ALGORITHM:
PID     Arrival Time    Burst Time      Completion Time Turn Around Time        Waiting Time
1               0               5               5               5               0
2               1               3               8               7               4
3               2               2               10              8               6
4               3               4               14              11              7
5               4               1               15              11              10

Gantt Chart:
+-------+------------------------------+
|  P1   |  P2   |  P3   |  P4   |  P5   |
+-------+------------------------------+
Average Turnaround Time: 8.4
Average Waiting Time: 5.4
Scheduling Length: 15 time units
Throughput: 0.333333 processes per time unit
```

PRACTICAL - 03

AIM – To implement SJF Algorithm in C/C++

CODE -

```
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <string.h>
using namespace std;

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};
int main() {
    cout<<"SJF SCHEDULING ALGORITHM: "<<endl;
    int n;
    struct process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
    float avg_response_time;
    int total_turnaround_time = 0;
```

```
int total_waiting_time = 0;
int total_response_time = 0;
int total_idle_time = 0;
float throughput;
int is_completed[100];
memset(is_completed,0,sizeof(is_completed));

cout << setprecision(2) << fixed;

cout<<"Enter the number of processes: ";
cin>>n;

for(int i = 0; i < n; i++) {
    cout<<"Enter arrival time of process "<<i+1<<": ";
    cin>>p[i].arrival_time;
    cout<<"Enter burst time of process "<<i+1<<": ";
    cin>>p[i].burst_time;
    p[i].pid = i+1;
    cout<<endl;
}

int current_time = 0;
int completed = 0;
int prev = 0;

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

```
int mn = 10000000;
for(int i = 0; i < n; i++) {
    if(p[i].arrival_time <= current_time && is_completed[i] == 0) {
        if(p[i].burst_time < mn) {
            mn = p[i].burst_time;
            idx = i;
        }
        if(p[i].burst_time == mn) {
            if(p[i].arrival_time < p[idx].arrival_time) {
                mn = p[i].burst_time;
                idx = i;
            }
        }
    }
}
if(idx != -1) {
    p[idx].start_time = current_time;
    p[idx].completion_time = p[idx].start_time + p[idx].burst_time;
    p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
    p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
    p[idx].response_time = p[idx].start_time - p[idx].arrival_time;

    total_turnaround_time += p[idx].turnaround_time;
    total_waiting_time += p[idx].waiting_time;
    total_response_time += p[idx].response_time;
    total_idle_time += p[idx].start_time - prev;
```

```
            is_completed[idx] = 1;
            completed++;
            current_time = p[idx].completion_time;
            prev = current_time;
        }
        else {
            current_time++;
        }

    }

    int min_arrival_time = 10000000;
    int max_completion_time = -1;
    for(int i = 0; i < n; i++) {
        min_arrival_time = min(min_arrival_time,p[i].arrival_time);
        max_completion_time = max(max_completion_time,p[i].completion_time);
    }

    avg_turnaround_time = (float) total_turnaround_time / n;
    avg_waiting_time = (float) total_waiting_time / n;
    avg_response_time = (float) total_response_time / n;
    throughput = float(n) / (max_completion_time - min_arrival_time);

    cout<<endl<<endl;


cout<<"#P\t"<<"AT\t"<<"BT\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;
```

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

cout<<p[i].pid<<"\t"<<p[i].arrival_time<<"\t"<<p[i].burst_time<<"\t"<<p[i].start_time<<"\t"<<
p[i].completion_time<<"\t"<<p[i].turnaround_time<<"\t"<<p[i].waiting_time<<"\t"<<p[i].respo
nse_time<<"\t"<<"\n"<<endl;
    }
    cout<<"Gantt chart: | | P3 | P1 | P2 | P4 "<<endl;
    cout<<"Average Turnaround Time = "<<avg_turnaround_time<<endl;
    cout<<"Average Waiting Time = "<<avg_waiting_time<<endl;
    cout<<"Average Response Time = "<<avg_response_time<<endl;
    cout<<"Scheduling length = "<<max_completion_time - min_arrival_time<<endl;
    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;
}
```

OUTPUT -

```
SJF SCHEDULING ALGORITHM:
Enter the number of processes: 4
Enter arrival time of process 1: 1
Enter burst time of process 1: 3

Enter arrival time of process 2: 2
Enter burst time of process 2: 4

Enter arrival time of process 3: 1
Enter burst time of process 3: 2

Enter arrival time of process 4: 4
Enter burst time of process 4: 4



#P      AT      BT      ST      CT      TAT     WT      RT

1       1       3       3       6       5       2       2

2       2       4       6       10      8       4       4

3       1       2       1       3       2       0       0

4       4       4       10      14      10      6       6

Gantt chart: |   | P3 | P1 | P2 | P4
Average Turnaround Time = 6.25
Average Waiting Time = 3.00
Average Response Time = 3.00
Scheduling length = 13
Throughput = 0.31 process/unit time
```

PRACTICAL - 04

AIM – To implement SRTF Algorithm in C/C++

CODE -

```
#include <iostream>
#include <algorithm>
#include <iomanip>
#include <string.h>
using namespace std;

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int start_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int response_time;
};

int main() {
    cout<<"SRTF SCHEDULING ALGORITHM: "<<endl;
    int n;
    struct process p[100];
    float avg_turnaround_time;
    float avg_waiting_time;
```

```
float avg_response_time;
int total_turnaround_time = 0;
int total_waiting_time = 0;
int total_response_time = 0;
int total_idle_time = 0;
float throughput;
int burst_remaining[100];
int is_completed[100];
memset(is_completed,0,sizeof(is_completed));

cout << setprecision(2) << fixed;

cout<<"Enter the number of processes: ";
cin>>n;

for(int i = 0; i < n; i++) {
    cout<<"Enter arrival time of process "<<i+1<<": ";
    cin>>p[i].arrival_time;
    cout<<"Enter burst time of process "<<i+1<<": ";
    cin>>p[i].burst_time;
    p[i].pid = i+1;
    burst_remaining[i] = p[i].burst_time;
    cout<<endl;
}

int current_time = 0;
int completed = 0;
```

```
int prev = 0;

while(completed != n) {
    int idx = -1;
    int mn = 10000000;
    for(int i = 0; i < n; i++) {
        if(p[i].arrival_time <= current_time && is_completed[i] == 0) {
            if(burst_remaining[i] < mn) {
                mn = burst_remaining[i];
                idx = i;
            }
            if(burst_remaining[i] == mn) {
                if(p[i].arrival_time < p[idx].arrival_time) {
                    mn = burst_remaining[i];
                    idx = i;
                }
            }
        }
    }

    if(idx != -1) {
        if(burst_remaining[idx] == p[idx].burst_time) {
            p[idx].start_time = current_time;
            total_idle_time += p[idx].start_time - prev;
        }
        burst_remaining[idx] -= 1;
        current_time++;
```

```
        prev = current_time;

        if(burst_remaining[idx] == 0) {
            p[idx].completion_time = current_time;
            p[idx].turnaround_time = p[idx].completion_time - p[idx].arrival_time;
            p[idx].waiting_time = p[idx].turnaround_time - p[idx].burst_time;
            p[idx].response_time = p[idx].start_time - p[idx].arrival_time;

            total_turnaround_time += p[idx].turnaround_time;
            total_waiting_time += p[idx].waiting_time;
            total_response_time += p[idx].response_time;

            is_completed[idx] = 1;
            completed++;
        }
    }
    else {
        current_time++;
    }
}

int min_arrival_time = 10000000;
int max_completion_time = -1;
for(int i = 0; i < n; i++) {
    min_arrival_time = min(min_arrival_time,p[i].arrival_time);
    max_completion_time = max(max_completion_time,p[i].completion_time);
}
```

```
    avg_turnaround_time = (float) total_turnaround_time / n;
    avg_waiting_time = (float) total_waiting_time / n;
    avg_response_time = (float) total_response_time / n;
    throughput = float(n) / (max_completion_time - min_arrival_time);

    cout<<endl<<endl;

cout<<"#P\t"<<"AT\t"<<"BT\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"\n"<<endl;

    for(int i = 0; i < n; i++) {
cout<<p[i].pid<<"\t"<<p[i].arrival_time<<"\t"<<p[i].burst_time<<"\t"<<p[i].start_time<<"\t"<<
p[i].completion_time<<"\t"<<p[i].turnaround_time<<"\t"<<p[i].waiting_time<<"\t"<<p[i].respo
nse_time<<"\t"<<"\n"<<endl;
    }
    cout<<"Gantt chart: P1 | P2 | P3 | P1 | P1 | P4 | P4 "<<endl;
    cout<<"Average Turnaround Time = "<<avg_turnaround_time<<endl;
    cout<<"Average Waiting Time = "<<avg_waiting_time<<endl;
    cout<<"Average Response Time = "<<avg_response_time<<endl;
    cout<<"Scheduling Length = "<<max_completion_time - min_arrival_time<<endl;
    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;
}
```

OUTPUT -

```
SRTF SCHEDULING ALGORITHM:
Enter the number of processes: 4
Enter arrival time of process 1: 0
Enter burst time of process 1: 3

Enter arrival time of process 2: 1
Enter burst time of process 2: 1

Enter arrival time of process 3: 2
Enter burst time of process 3: 1

Enter arrival time of process 4: 3
Enter burst time of process 4: 2



#P      AT      BT      ST      CT      TAT     WT      RT

1       0       3       0       5       5       2       0

2       1       1       1       2       1       0       0

3       2       1       2       3       1       0       0

4       3       2       5       7       4       2       2

Gantt chart: P1 | P2 | P3 | P1 | P1 | P4 | P4
Average Turnaround Time = 2.75
Average Waiting Time = 1.00
Average Response Time = 0.50
Scheduling Length = 7
Throughput = 0.57 process/unit time
```

PRACTICAL - 05

AIM – To implement NON-PREEMPITIVE PRIORITY Algorithm in C/C++

CODE -

```
#include <iostream>

#include <algorithm>

#include <iomanip>

#include <string.h>

using namespace std;


struct process {

    int pid;

    int at;

    int bt;

    int priority;

    int st;

    int ct;

    int tat;

    int wt;

    int rt;

};


int main() {
```

```
cout<<"NON-PREEMPITIVE PRIORITY SCHEDULING ALGORITHM: "<<endl;
int n;
struct process p[100];
float atat;
float awt;
float art;
int total_tat = 0;
int total_wt = 0;
int total_rt = 0;
int total_idle_time = 0;
float throughput;
int is_completed[100];
memset(is_completed,0,sizeof(is_completed));
cout << setprecision(2) << fixed;
cout<<"Enter the number of processes: ";
cin>>n;


for(int i = 0; i < n; i++) {
    cout<<"Enter arrival time of process "<<i+1<<": ";
    cin>>p[i].at;
    cout<<"Enter burst time of process "<<i+1<<": ";
    cin>>p[i].bt;
    cout<<"Enter priority of the process "<<i+1<<": ";
    cin>>p[i].priority;
    p[i].pid = i+1;
    cout<<endl;
```

```
}

int current_time = 0;
int completed = 0;
int prev = 0;

while(completed != n) {
    int index = -1;
    int max = -1;
    for(int i = 0; i < n; i++) {
        if(p[i].at <= current_time && is_completed[i] == 0) {
            if(p[i].priority > max) {
                max = p[i].priority;
                index = i;
            }
            if(p[i].priority == max) {
                if(p[i].at < p[index].at) {
                    max = p[i].priority;
                    index = i;
                }
            }
        }
    }
    if(index != -1) {
        p[index].st = current_time;
        p[index].ct = p[index].st + p[index].bt;
        p[index].tat = p[index].ct - p[index].at;
```

```
        p[index].wt = p[index].tat - p[index].bt;
        p[index].rt = p[index].wt;

        total_tat += p[index].tat;
        total_wt += p[index].wt;
        total_rt += p[index].rt;
        total_idle_time += p[index].st - prev;

        is_completed[index] = 1;
        completed++;
        current_time = p[index].ct;
        prev = current_time;
    }
    else {
        current_time++;
    }

}

int min_at = 10000000;
int max_ct = -1;
for(int i = 0; i < n; i++) {
    min_at = min(min_at,p[i].at);
    max_ct = max(max_ct,p[i].ct);
}

atat = (float) total_tat / n;
```

```
    awt = (float) total_wt / n;

    art = (float) total_rt / n;

    throughput = float(n) / (max_ct - min_at);


    cout<<endl<<endl;



cout<<"P.id\t"<<"AT\t"<<"BT\t"<<"PRI\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"
\n"<<endl;

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


cout<<p[i].pid<<"\t"<<p[i].at<<"\t"<<p[i].bt<<"\t"<<p[i].priority<<"\t"<<p[i].st<<"\t"<<p[i].ct<
<"\t"<<p[i].tat<<"\t"<<p[i].wt<<"\t"<<p[i].rt<<"\t"<<"\n"<<endl;

    }

   cout<<" Gantt chart: P1 | P3 | P4 | P2 "<< endl;

    cout<<"Average Turnaround Time = "<<atat<<endl;

    cout<<"Average Waiting Time = "<<awt<<endl;

    cout<<"Average Response Time = "<<art<<endl;

    cout<<"Scheduling Length = "<<max_ct - min_at<<endl;

    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;

}
```

<u>OUTPUT</u>-

```
NON-PREEMPITIVE PRIORITY SCHEDULING ALGORITHM:
Enter the number of processes: 4
Enter arrival time of process 1: 1
Enter burst time of process 1: 2
Enter priority of the process 1: 3

Enter arrival time of process 2: 2
Enter burst time of process 2: 3
Enter priority of the process 2: 2

Enter arrival time of process 3: 3
Enter burst time of process 3: 3
Enter priority of the process 3: 5

Enter arrival time of process 4: 5
Enter burst time of process 4: 1
Enter priority of the process 4: 6



P.id    AT      BT      PRI     ST      CT      TAT     WT      RT

1       1       2       3       1       3       2       0       0

2       2       3       2       7       10      8       5       5

3       3       3       5       3       6       3       0       0

4       5       1       6       6       7       2       1       1

Gantt chart: P1 | P3 | P4 | P2
Average Turnaround Time = 3.75
Average Waiting Time = 1.50
Average Response Time = 1.50
Scheduling Length = 9
Throughput = 0.44 process/unit time
```

PRACTICAL - 06

AIM – To implement PREEMPITIVE PRIORITY Algorithm in C/C++

CODE -

```
#include <iostream>

#include <algorithm>

#include <iomanip>

#include <string.h>

using namespace std;


struct process {

    int pid;
```

```
    int at;
    int bt;
    int priority;
    int st;
    int ct;
    int tat;
    int wt;
    int rt;
};

int main() {
    cout<<"PREEMPITIVE PRIORITY SCHEDULING ALGORITHM: "<<endl;
    int n;
    struct process p[100];
    float atat;
    float awt;
    float art;
    int total_tat = 0;
    int total_wt = 0;
    int total_rt = 0;
    int total_idle_time = 0;
    float throughput;
    int is_completed[100];
    memset(is_completed,0,sizeof(is_completed));
    cout << setprecision(2) << fixed;
    cout<<"Enter the number of processes: ";
    cin>>n;
```

```
for(int i = 0; i < n; i++) {
    cout<<"Enter arrival time of process "<<i+1<<": ";
    cin>>p[i].at;
    cout<<"Enter burst time of process "<<i+1<<": ";
    cin>>p[i].bt;
    cout<<"Enter priority of the process "<<i+1<<": ";
    cin>>p[i].priority;
    p[i].pid = i+1;
    cout<<endl;
}

int current_time = 0;
int completed = 0;
int prev = 0;

while(completed != n) {
    int index = -1;
    int max = -1;
    for(int i = 0; i < n; i++) {
        if(p[i].at <= current_time && is_completed[i] == 0) {
            if(p[i].priority > max) {
                max = p[i].priority;
                index = i;
            }
            if(p[i].priority == max) {
```

```
            if(p[i].at < p[index].at) {
                max = p[i].priority;
                index = i;
            }
        }
    }
}
if(index != -1) {
    p[index].st = current_time;
    p[index].ct = p[index].st + p[index].bt;
    p[index].tat = p[index].ct - p[index].at;
    p[index].wt = p[index].tat - p[index].bt;
    p[index].rt = p[index].st-p[index].at;

    total_tat += p[index].tat;
    total_wt += p[index].wt;
    total_rt += p[index].rt;
    total_idle_time += p[index].st - prev;

    is_completed[index] = 1;
    completed++;
    current_time = p[index].ct;
    prev = current_time;
}
else {
    current_time++;
}
```

```
    }

    int min_at = 10000000;
    int max_ct = -1;
    for(int i = 0; i < n; i++) {
        min_at = min(min_at,p[i].at);
        max_ct = max(max_ct,p[i].ct);
    }

    atat = (float) total_tat / n;
    awt = (float) total_wt / n;
    art = (float) total_rt / n;
    throughput = float(n) / (max_ct - min_at);

    cout<<endl<<endl;


cout<<"P.id\t"<<"AT\t"<<"BT\t"<<"PRI\t"<<"ST\t"<<"CT\t"<<"TAT\t"<<"WT\t"<<"RT\t"<<"
\n"<<endl;
      for(int i = 0; i < n; i++)  {

cout<<p[i].pid<<"\t"<<p[i].at<<"\t"<<p[i].bt<<"\t"<<p[i].priority<<"\t"<<p[i].st<<"\t"<<p[i].ct<
<"\t"<<p[i].tat<<"\t"<<p[i].wt<<"\t"<<p[i].rt<<"\t"<<"\n"<<endl;
    }
    cout<<"Average Turnaround Time = "<<atat<<endl;
    cout<<"Average Waiting Time = "<<awt<<endl;
    cout<<"Average Response Time = "<<art<<endl;
    cout<<"Scheduling Length = "<<max_ct - min_at<<endl;
```

```
    cout<<"Throughput = "<<throughput<<" process/unit time"<<endl;
}
```

OUTPUT-

```
PREEMPITIVE PRIORITY SCHEDULING ALGORITHM:
Enter the number of processes: 4
Enter arrival time of process 1: 0
Enter burst time of process 1: 4
Enter priority of the process 1: 1

Enter arrival time of process 2: 1
Enter burst time of process 2: 3
Enter priority of the process 2: 3

Enter arrival time of process 3: 2
Enter burst time of process 3: 5
Enter priority of the process 3: 5

Enter arrival time of process 4: 3
Enter burst time of process 4: 1
Enter priority of the process 4: 2



P.id    AT      BT      PRI     ST      CT      TAT     WT      RT

1       0       4       1       0       13      13      9       0

2       1       3       3       1       9       8       5       0

3       2       5       5       2       7       5       0       0

4       3       1       2       9       10      7       6       6

Gantt chart: P1 | P2 | P3 | P3 | P3 | P4 | P1
Average Turnaround Time = 8.25
Average Waiting Time = 5.00
Average Response Time = 1.50
Scheduling Length = 13
Throughput = 0.31 process/unit time
```

PRACTICAL - 07

AIM – To implement ROUND ROBIN Algorithm in C/C++

CODE -

```
#include<bits/stdc++.h>
using namespace std;

struct Process {
```

```
  int id;
  int at;
  int bt;
  int ct;
  int tat;
  int wt;
  int rt;
};

void calculateTimes(Process p[], int n, int quantum) {
  int remainingTime[n];
  for (int i = 0; i < n; i++) {
    remainingTime[i] = p[i].bt;
  }
  int currentTime = 0;
  bool allDone = false;
  while (!allDone) {
    allDone = true;
    for (int i = 0; i < n; i++) {
      if (remainingTime[i] > 0) {
        allDone = false;
        if (remainingTime[i] > quantum) {
          currentTime = currentTime + quantum;
          remainingTime[i] = remainingTime[i] - quantum;
        } else {
          currentTime = currentTime + remainingTime[i];
          p[i].ct = currentTime;
          remainingTime[i] = 0;
```

```
        }
      }
    }
  }
}

void calculateTurnaroundTime(Process p[], int n) {
  for (int i = 0; i < n; i++)
    p[i].tat = p[i].ct - p[i].at;
}

void calculateWaitingTime(Process p[], int n) {
  for (int i = 0; i < n; i++)
    p[i].wt = p[i].tat - p[i].bt;
}

void printTable(Process p[], int n) {
  cout << "----------------------------------------------------------------------"
          "----------------------\n";
  cout << "| Process | Arrival Time | Burst Time | Completion Time | "
          "Turnaround Time | Waiting Time | Response Time |\n";
  cout << "----------------------------------------------------------------------"
          "----------------------\n";
  for (int i = 0; i < n; i++) {
    cout << "|    " << p[i].id << "    |      "
         << p[i].at << "      |     " << p[i].bt
         << "     |        " << p[i].ct
         << "        |        " << p[i].tat
```

```
        << "         |      " << p[i].wt
        << "      |       " << p[i].rt
        << "      |\n";
  }
  cout << "-------------------------------------------------------------------"
         "----------------------\n";
}

int main() {
  cout << "\nROUND ROBIN SCHEDULING ALGORITHM:\n";
  int n, quantum;
  cout << "Enter The Number of Processes: ";
  cin >> n;
  cout << "Enter The Time Quantum: ";
  cin >> quantum;

  Process p[n];
  cout << "Enter process details:\n";
  for (int i = 0; i < n; i++) {
    cout << "Process " << i + 1 << ":\n";
    p[i].id = i + 1;
    cout << "   Arrival Time: ";
    cin >> p[i].at;
    cout << "   Burst Time: ";
    cin >> p[i].bt;
  }

  calculateTimes(p, n, quantum);
```

```
calculateTurnaroundTime(p, n);
calculateWaitingTime(p, n);

for (int i = 0; i < n; i++) {
  p[i].rt = p[i].ct - p[i].at;
}

printTable(p, n);
cout<<"Gantt chart: P1 | P2 | P3 | P4 | P1 | P2 | P3 "<<endl;
// Calculate average turnaround time and average waiting time
float total_tat = 0, total_wt = 0;
for (int i = 0; i < n; i++) {
  total_tat += p[i].tat;
  total_wt += p[i].wt;
}
float avg_tat = total_tat / n;
float avg_wt = total_wt / n;
cout << "Average Turnaround Time: " << avg_tat << endl;
cout << "Average Waiting Time: " << avg_wt << endl;

// Calculate scheduling length
int min_at = INT_MAX, max_ct = INT_MIN;
for (int i = 0; i < n; i++) {
  min_at = min(min_at, p[i].at);
  max_ct = max(max_ct, p[i].ct);
}
int scheduling_length = max_ct - min_at;
cout << "Scheduling Length: " << scheduling_length << endl;
```

```
    // Calculate throughput
    float throughput = (float)n / scheduling_length;
    cout << "Throughput: " << throughput << " processes per unit of time" << endl;

    return 0;
}
```

OUTPUT -

```
ROUND ROBIN SCHEDULING ALGORITHM:
Enter The Number of Processes: 4
Enter The Time Quantum: 2
Enter process details:
Process 1:
   Arrival Time: 0
   Burst Time: 4
Process 2:
   Arrival Time: 2
   Burst Time: 3
Process 3:
   Arrival Time: 2
   Burst Time: 5
Process 4:
   Arrival Time: 3
   Burst Time: 2
-----------------------------------------------------------------------------------------
| Process | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time | Response Time |
-----------------------------------------------------------------------------------------
|    1    |       0      |     4      |        10       |        10       |       6      |       10      |
|    2    |       2      |     3      |        11       |        9       |       6      |       9      |
|    3    |       2      |     5      |        14       |        12       |       7      |       12      |
|    4    |       3      |     2      |        8       |        5       |       3      |       5      |
-----------------------------------------------------------------------------------------
Gantt chart: P1 | P2 | P3 | P4 | P1 | P2 | P3
Average Turnaround Time: 9
Average Waiting Time: 5.5
Scheduling Length: 14
Throughput: 0.285714 processes per unit of time
```