CODE:

SJF

```c
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

// Structure to hold process information
struct Process {
    int process_id;       // Process ID
    int arrival_time;     // Arrival time
    int burst_time;       // Burst time
    int completion_time;  // Completion time
    int turnaround_time;  // Turnaround time
    int waiting_time;     // Waiting time
    int remaining_time;   // Remaining burst time for preemption
    int is_completed;     // Flag to check if process is completed
};

// SJF scheduling (Non-Preemptive)
void sjf_schedule(struct Process processes[], int n) {
    int current_time = 0; // Current time
    int completed = 0;    // Number of processes completed
    int arr[n]; // To store Order of processes executed
    int i=0;
    while (completed < n) {
        int shortest_index = -1;
        int shortest_burst = INT_MAX;

        // Find the shortest job that has arrived
        for (int i = 0; i < n; ++i) {
            if (processes[i].arrival_time <= current_time &&
                processes[i].is_completed == 0 &&
                processes[i].burst_time < shortest_burst) {
                shortest_index = i;
                shortest_burst = processes[i].burst_time;
            }
        }

        if (shortest_index == -1) {
            // If no process is available to execute, move to next moment
            current_time++;
        } else {
            // Execute the shortest job
            processes[shortest_index].completion_time = current_time +
processes[shortest_index].burst_time;
            processes[shortest_index].turnaround_time =
processes[shortest_index].completion_time -
processes[shortest_index].arrival_time;
            processes[shortest_index].waiting_time =
processes[shortest_index].turnaround_time -
processes[shortest_index].burst_time;
            processes[shortest_index].is_completed = 1;
            current_time = processes[shortest_index].completion_time;
            arr[i]=shortest_index+1;
```

```c
            completed++;
            i++;
        }
    }

    // Calculate total waiting time and turnaround time
    double total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; ++i) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Print average waiting time and average turnaround time
    printf("Average Waiting Time: %f\n", total_waiting_time / n);
    printf("Average Turnaround Time: %f\n", total_turnaround_time / n);

    printf("\n");

    printf("Gantt chart: \n\n");
    for (int i=0; i<n; i++){
        if(i==0){
            printf("| P%d |",arr[i]);
        }
        else{
            printf(" P%d |",arr[i]);
        }
    }
    printf("\n");
    for (int i=0; i<n; i++){
        if(i==0){
            printf("0    %d    ",processes[arr[i]-1].completion_time);
        }
        else{
             printf("%d    ",processes[arr[i]-1].completion_time);
        }
    }
    printf("\n\n");
}

// SJF scheduling (Preemptive)
void srtf_schedule(struct Process proc[], int n) {
    int rt[n];
    int arr[1000]; // To store the order of execution
    int gantt_time[1000]; // To store the time at each switch
    int exec_idx = 0; // Index to track gantt chart entries

    // Copy the burst time into rt[]
    for (int i = 0; i < n; i++)
        rt[i] = proc[i].burst_time;

    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    int check = 0;
```

```
    // Process until all processes get completed
    while (complete != n) {
        // Find process with minimum remaining time among the processes
that arrive till the current time
        for (int j = 0; j < n; j++) {
            if ((proc[j].arrival_time <= t) &&
                (rt[j] < minm) && rt[j] > 0) {
                minm = rt[j];
                shortest = j;
                check = 1;
            }
        }

        if (check == 0) {
            t++;
            continue;
        }

        // Track the process execution in Gantt chart
        arr[exec_idx] = proc[shortest].process_id;
        gantt_time[exec_idx] = t;
        exec_idx++;

        // Reduce remaining time by one
        rt[shortest]--;

        // Update minimum
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;

        // If a process gets completely executed
        if (rt[shortest] == 0) {
            complete++;
            check = 0;

            // Find finish time of current process
            finish_time = t + 1;

            // Calculate waiting time
            proc[shortest].waiting_time = finish_time -
proc[shortest].burst_time - proc[shortest].arrival_time;

            if (proc[shortest].waiting_time < 0)
                proc[shortest].waiting_time = 0;

            // Calculate turnaround time
            proc[shortest].turnaround_time = proc[shortest].burst_time +
proc[shortest].waiting_time;

            // Store the completion time
            proc[shortest].completion_time = finish_time;
        }
        // Increment time
```

```
        t++;
    }

    // Calculate total waiting time and turnaround time
    double total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; ++i) {
        total_waiting_time += proc[i].waiting_time;
        total_turnaround_time += proc[i].turnaround_time;
    }

    // Print average waiting time and average turnaround time
    printf("Average Waiting Time: %f\n", total_waiting_time / n);
    printf("Average Turnaround Time: %f\n", total_turnaround_time / n);

    // Print Gantt chart for SRTF
    printf("\nGantt chart: \n\n");
    for (int i = 0; i < exec_idx; i++) {
        if (i == 0 || arr[i] != arr[i - 1]) {
            printf("| P%d ", arr[i]);
        }
    }
    printf("|\n");

    for (int i = 0; i < exec_idx; i++) {
        if (i == 0 || arr[i] != arr[i - 1]) {
            printf("%d   ", gantt_time[i]);
        }
    }
    printf("%d\n\n", t);
}

int main() {
    int n;
    int choice;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; ++i) {
        processes[i].process_id = i + 1;
        printf("Enter arrival time and burst time for process %d: ", i +
1);
        scanf("%d %d", &processes[i].arrival_time,
&processes[i].burst_time);
        processes[i].is_completed = 0;
        processes[i].remaining_time = processes[i].burst_time; //
Initialize remaining time for SRTF
    }

    printf("Choose the scheduling algorithm:\n");
    printf("1. Shortest Job First (Non-Preemptive)\n");
```

```
    printf("2. Shortest Remaining Time First (Preemptive)\n");

    printf("Enter your choice (1 or 2): ");
    scanf("%d", &choice);

    // Perform the chosen scheduling algorithm
    if (choice == 1) {
        sjf_schedule(processes, n);
    } else if (choice == 2) {
        srtf_schedule(processes, n);
    } else {
        printf("Invalid choice!\n");
        return 1;
    }

    // Display process details along with completion time
    printf("Process-ID\tArrival-Time\tBurst-Time\tTurnaround-
Time\tWaiting-Time\n");

    for (int i = 0; i < n; ++i) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
               processes[i].process_id,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].turnaround_time,
               processes[i].waiting_time);
    }

    return 0;
}
```

```
OUTPUT:

SJF(non-preemptive):

Enter the number of processes: 5
Enter arrival time and burst time for process 1: 2 6
Enter arrival time and burst time for process 2: 5 2
Enter arrival time and burst time for process 3: 1 8
Enter arrival time and burst time for process 4: 0 3
Enter arrival time and burst time for process 5: 4 4

Choose the scheduling algorithm:
1. Shortest Job First (Non-Preemptive)
2. Shortest Remaining Time First (Preemptive)
Enter your choice (1 or 2): 1
Average Waiting Time: 5.200000
Average Turnaround Time: 9.800000

Gannt chart:

| P4 | P1 | P2 | P5 | P3 |
0    3    9    11   15   23
```

Process-ID	Arrival-Time	Burst-Time	Turnaround-Time	Waiting-Time
1	2	6	7	1
2	5	2	6	4
3	1	8	22	14
4	0	3	3	0
5	4	4	11	7

SJF(preemptive):

Enter the number of processes: 5
Enter arrival time and burst time for process 1: 2 6
Enter arrival time and burst time for process 2: 5 2
Enter arrival time and burst time for process 3: 1 8
Enter arrival time and burst time for process 4: 0 3
Enter arrival time and burst time for process 5: 4 4

Choose the scheduling algorithm:
1. Shortest Job First (Non-Preemptive)
2. Shortest Remaining Time First (Preemptive)
Enter your choice (1 or 2): 2

Average Waiting Time: 4.600000
Average Turnaround Time: 9.200000

Gantt chart:

```
| P4 | P1 | P5 | P2 | P5 | P1 | P3 |
0    3    4    5    7    10   15   23
```

Process-ID	Arrival-Time	Burst-Time	Turnaround-Time	Waiting-Time
1	2	6	13	7
2	5	2	2	0
3	1	8	22	14
4	0	3	3	0
5	4	4	6	2

RoundRobin:
CODE:

```
#include <stdio.h>
#include <stdlib.h>
struct Process
{
      int pid;
      int at;
      int bt;
      int rt;
      int wt;
      int tat;
};
struct Node
{
      struct Process *data;
```

```
	struct Node *next;
};
struct Queue
{
	struct Node *front;
	struct Node *rear;
	int size;
};
struct Process* create_process(int pid, int at, int bt)
{
	struct Process *p = (struct Process*)malloc(sizeof(struct
Process));
	p->pid = pid;
	p->at = at;
	p->bt = bt;
	p->rt = bt;
	p->wt = 0;
	p->tat = 0;
	return p;
}
struct Node* create_node(struct Process *p)
{
	struct Node *node = (struct Node*)malloc(sizeof(struct Node));
	node->data = p;
	node->next = NULL;
	return node;
}
void init_queue(struct Queue *q)
{
	q->front = q->rear = NULL;
	q->size = 0;
}
int is_empty(struct Queue *q)
{
	return q->size == 0;
}
void enqueue(struct Queue *q, struct Process *p) {
	struct Node *node = create_node(p);
	if (is_empty(q))
	{
		q->front = q->rear = node;
	}
	else
	{
		q->rear->next = node;
		q->rear = node;
	}
	q->size++;
}
struct Process* dequeue(struct Queue *q)
{
	if (is_empty(q))
	{
		return NULL;
```

```
    }
    struct Node *temp = q->front;
    struct Process *p = temp->data;
    q->front = q->front->next;
    if (q->front == NULL)
    {
        q->rear = NULL;
    }
    free(temp);
    q->size--;
    return p;
}
void store_gantt_chart(int gantt_chart[][2], int *gantt_index, int time,
int pid)
{
    gantt_chart[*gantt_index][0] = time;
    gantt_chart[*gantt_index][1] = pid;
    (*gantt_index)++;
}
void print_gantt_chart(int gantt_chart[][2], int gantt_index)
{
    printf("\nGantt Chart:\n");
    printf("|");
    for (int i = 0; i < gantt_index; i++)
    {

        printf(" P%d |", gantt_chart[i][1]);
    }
    printf("\n");

    printf("0 ");
    for (int i = 0; i < gantt_index; i++)
    {

      printf("  %d  ", gantt_chart[i][0]);


    }
    printf("\n");
}
int main()
{
    int n, quant, i, time = 0, total_wt = 0, total_tat = 0;
    printf("Total number of processes in the system: ");
    scanf("%d", &n);
    struct Process *processes[n];
    for (i = 0; i < n; i++) {
        int at, bt;
        printf("\nEnter the Arrival and Burst time of Process[%d]\n",
i + 1);
        printf("Arrival time: ");
        scanf("%d", &at);
        printf("Burst time: ");
        scanf("%d", &bt);
```

```
            processes[i] = create_process(i + 1, at, bt);
      }
      printf("Enter the Time Quantum for the process: ");
      scanf("%d", &quant);
      struct Queue q;
      init_queue(&q);
      for (i = 0; i < n; i++)
      {
            if (processes[i]->at == 0)
            {
                  enqueue(&q, processes[i]);
            }
      }
      int gantt_chart[100][2];
      int gantt_index = 0;
      printf("\nProcess No\tBurst Time\tTAT\tWaiting Time\n");
      while (!is_empty(&q))
      {
            struct Process *current = dequeue(&q);
            if (current->rt > quant)
            {
                  time += quant;
                  current->rt -= quant;
                  store_gantt_chart(gantt_chart, &gantt_index, time,
current->pid);
            }
            else
            {
                  time += current->rt;
                  current->rt = 0;
                  current->tat = time - current->at;
                  current->wt = current->tat - current->bt;
                  printf("%d\t\t%d\t\t%d\t%d\n", current->pid, current-
>bt, current->tat, current->wt);
                  store_gantt_chart(gantt_chart, &gantt_index, time,
current->pid);
                  total_wt += current->wt;
                  total_tat += current->tat;
            }
            for (i = 0; i < n; i++)
            {
                  if (processes[i]->at <= time && processes[i]->rt > 0 &&
processes[i] != current)
                  {
                        int already_in_queue = 0;
                        struct Node *node = q.front;
                        while (node != NULL) {
                              if (node->data == processes[i])
                              {
                                    already_in_queue = 1;
                                    break;
                              }
                              node = node->next;
                        }
```

```
                    if (!already_in_queue)
                    {
                         enqueue(&q, processes[i]);
                    }
               }
          }
          if (current->rt > 0)
          {
               enqueue(&q, current);
          }
     }
     print_gantt_chart(gantt_chart, gantt_index);
     float avg_wt = (float)total_wt / n;
     float avg_tat = (float)total_tat / n;
     printf("\nAverage Turn Around Time: %.2f", avg_tat);
     printf("\nAverage Waiting Time: %.2f", avg_wt);
     return 0;
}
```

```
OUTPUT:
Total number of processes in the system: 5

Enter the Arrival and Burst time of Process[1]
Arrival time: 2
Burst time: 6

Enter the Arrival and Burst time of Process[2]
Arrival time: 5
Burst time: 2

Enter the Arrival and Burst time of Process[3]
Arrival time: 1
Burst time: 8

Enter the Arrival and Burst time of Process[4]
Arrival time: 0
Burst time: 3

Enter the Arrival and Burst time of Process[5]
Arrival time: 4
Burst time: 4
Enter the Time Quantum for the process: 2

Process No  Burst Time  TAT   Waiting Time
4           3           7     4
2           2           8     6
5           4           13    9
1           6           17    11
3           8           22    14

Gantt Chart:
| P4 | P1 | P3 | P4 | P5 | P1 | P2 | P3 | P5 | P1 | P3 | P3 |
0    2    4    6    7    9    11   13   15   17   19   21   23
```

Average Turn Around Time: 13.40
Average Waiting Time: 8.80

FCFS:
CODE:

```
#include <stdio.h>
void fcfs(int arr[][5], int n) {
    int i, j, temp;
    int current_time = 0;
    int total_wt = 0, total_tat = 0;
    float avg_wt, avg_tat;

    // Bubble sort based on Arrival Time
    for(i = 0; i < n-1; i++) {
        for(j = 0; j < n-i-1; j++) {
            if (arr[j][0] > arr[j+1][0]) {
                // Swap Arrival Time
                temp = arr[j][0];
                arr[j][0] = arr[j+1][0];
                arr[j+1][0] = temp;

                // Swap Burst Time
                temp = arr[j][1];
                arr[j][1] = arr[j+1][1];
                arr[j+1][1] = temp;

                //Swap process number
                temp = arr[j][4];
                arr[j][4] = arr[j+1][4];
                arr[j+1][4] = temp;
            }
        }
    }

    // Calculate Turnaround Time and Waiting Time
    for(i = 0; i < n; i++) {
        if(current_time < arr[i][0]) {
            current_time = arr[i][0];
        }

        current_time += arr[i][1];

        // TAT=CT-AT
        arr[i][2] = current_time - arr[i][0];

        // WT=TAT-BT
        arr[i][3] = arr[i][2] - arr[i][1];

        total_wt += arr[i][3];
        total_tat += arr[i][2];
    }
```

```
    avg_wt = (float)total_wt / n;
    avg_tat = (float)total_tat / n;

    // Print results
    printf("\nProcess\t Arrival Time \tBurst Time \tTurnaround Time 
\tWaiting Time\n");
    for(i = 0; i < n; i++) {
        printf("%d\t\t %d\t\t %d\t\t %d\t\t\t %d\t\t\t\n", arr[i][4], 
arr[i][0], arr[i][1], arr[i][2], arr[i][3]);
    }

    printf("\nAverage Waiting Time = %f", avg_wt);
    printf("\nAverage Turnaround Time = %f", avg_tat);
    printf("\n");
}



int main() {
    int n, i;


    printf("Enter number of processes: ");
    scanf("%d", &n);


    int arr[n][5];
    int arr1[n+1];// used for gannt chart



    printf("\nEnter Arrival Time and Burst Time:\n");
    for(i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        arr[i][4]=i+1;  //process number array initialisation
        printf("Arrival Time: ");
        scanf("%d", &arr[i][0]);
        printf("Burst Time: ");
        scanf("%d",&arr[i][1]);
    }

    fcfs(arr, n);
    int sum=0;
    for(int i=0; i<=n; i++){
        if(i==0){
           arr1[i]=0;
           sum+=arr[i][1];
        }
        else{

                arr1[i]=sum;
                sum+=arr[i][1];
        }
```

```
    }

    for(int i=0; i<n; i++){
      if(i==0){
            printf("| P%d |",arr[i][4]);
      }
      else{
            printf(" P%d |",arr[i][4]);
      }


    }
    printf("\n");

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

        if(i==0){
            printf("%d    ", arr1[i]);
        }
      else{
         printf("%d    ",arr1[i]);

      }




    }
    printf("\n");

    return 0;
}
```

```
OUTPUT:
Enter number of processes: 5

Enter Arrival Time and Burst Time:
Process 1:
Arrival Time: 2
Burst Time: 6
Process 2:
Arrival Time: 5
Burst Time: 2
Process 3:
Arrival Time: 1
Burst Time: 8
Process 4:
Arrival Time: 0
Burst Time: 3
Process 5:
Arrival Time: 4
```

```
Burst Time: 4

Process      Arrival Time     Burst Time  Turnaround Time  Waiting Time
    4       0           3          3                0
    3       1           8          10               2
    1       2           6          15               9
    5       4           4          17               13
    2       5           2          18               16

Average Waiting Time = 8.000000
Average Turnaround Time = 12.600000
| P4 | P3 | P1 | P5 | P2 |
0    3    11   17   21   23
```