4a.FCFS – First Come First Served

PROGRAM:

```
#include <stdio.h>

int main() {
    int i, n;
    int burst[10], wait[10], turn[10];
    float w = 0, t = 0;

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

    printf("Enter the burst time for each process:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &burst[i]);
    }

    // Waiting time for first process is 0
    wait[0] = 0;

    // Calculate waiting times
    for (i = 1; i < n; i++) {
        wait[i] = wait[i - 1] + burst[i - 1];
    }

    // Calculate turnaround times
    for (i = 0; i < n; i++) {
        turn[i] = wait[i] + burst[i];
    }

    // Print Gantt chart
    printf("\nGantt Chart:\n");
    for (i = 0; i < n; i++)
        printf("\tP%d\t|", i + 1);
    printf("\n");

    int current = 0;
    for (i = 0; i < n; i++) {
        current += burst[i];
        printf("%d\t", current);
    }
    printf("\n");

    // Calculate averages
    for (i = 0; i < n; i++) {
        w += wait[i];
        t += turn[i];
    }
```

```
    w /= n;
    t /= n;

    printf("\nAverage waiting time: %.2f", w);
    printf("\nAverage turnaround time: %.2f\n", t);

    return 0;
}
```

4b.SJF (SHORTESTJOB FIRST)
PROGRAM

```
#include <stdio.h>

int main() {
    int n, i, j;
    int burst[10], wait[10], turn[10], process[10];
    int temp;
    float w = 0, t = 0;

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

    printf("Enter the burst time for each process:\n");
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &burst[i]);
        process[i] = i + 1;   // Store process numbers
    }

    // Sort burst times using simple bubble sort
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (burst[i] > burst[j]) {
                // Swap burst time
                temp = burst[i];
                burst[i] = burst[j];
                burst[j] = temp;

                // Swap process numbers
                temp = process[i];
                process[i] = process[j];
                process[j] = temp;
            }
        }
    }

    // First process waiting time is 0
    wait[0] = 0;
```

```
    // Calculate waiting time
    for (i = 1; i < n; i++) {
        wait[i] = wait[i - 1] + burst[i - 1];
    }

    // Calculate turnaround time
    for (i = 0; i < n; i++) {
        turn[i] = wait[i] + burst[i];
    }

    // Print Gantt Chart
    printf("\nGantt Chart:\n");
    for (i = 0; i < n; i++) {
        printf("P%d | ", process[i]);
    }

    printf("\n0 ");
    for (i = 0; i < n; i++) {
        printf("  %d", turn[i]);
    }

    // Calculate averages
    for (i = 0; i < n; i++) {
        w += wait[i];
        t += turn[i];
    }

    w /= n;
    t /= n;

    printf("\n\nAverage Waiting Time: %.2f", w);
    printf("\nAverage Turnaround Time: %.2f\n", t);

    return 0;
}
```

4c) Priority Scheduling
ALGORITHM

```
#include <stdio.h>

struct process {
    int id;         // Process ID
    int bt;         // Burst Time
    int wt;          // Waiting Time
    int tat;        // Turnaround Time
    int priority;   // Priority (Lower number = Higher priority)
};

// Function to calculate waiting time
```

```
void findWaitingTime(struct process proc[], int n) {
    proc[0].wt = 0;  // First process waiting time = 0

    for (int i = 1; i < n; i++) {
        proc[i].wt = proc[i - 1].wt + proc[i - 1].bt;
    }
}

// Function to calculate turnaround time
void findTurnaroundTime(struct process proc[], int n) {
    for (int i = 0; i < n; i++) {
        proc[i].tat = proc[i].wt + proc[i].bt;
    }
}

// Function to calculate and print averages
void findAverageTimes(struct process proc[], int n) {
    float totalWT = 0, totalTAT = 0;

    for (int i = 0; i < n; i++) {
        totalWT += proc[i].wt;
        totalTAT += proc[i].tat;
    }

    printf("\nAverage Waiting Time: %.2f", totalWT / n);
    printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
}

// Priority Scheduling (Non-Preemptive)
void priorityScheduling(struct process proc[], int n) {
    struct process temp;

    // Sort by priority (lower number = higher priority)
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (proc[i].priority > proc[j].priority ||
               (proc[i].priority == proc[j].priority && proc[i].bt > proc[j].bt)) {
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }

    findWaitingTime(proc, n);
    findTurnaroundTime(proc, n);

    printf("\nProcess\tBT\tPriority\tWT\tTAT\n");
    printf("-------------------------------------------------\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t\t%d\t%d\n",
               proc[i].id,
               proc[i].bt,
               proc[i].priority,
               proc[i].wt,
               proc[i].tat);
    }

    findAverageTimes(proc, n);
}

int main() {
    int n;

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

    if (n <= 0) {
        printf("Invalid number of processes.\n");
        return 1;
    }

    struct process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;

        printf("\nEnter Burst Time and Priority for Process P%d:\n", i + 1);

        printf("Burst Time: ");
        scanf("%d", &proc[i].bt);

        printf("Priority (Lower number = Higher priority): ");
        scanf("%d", &proc[i].priority);
    }

    priorityScheduling(proc, n);

    return 0;
}
```

4d RR Scheduling (Round Robin)

ALGORITHM:

```
#include <stdio.h>

// Function to calculate waiting time
void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {
    int rem_bt[n];
```

```
    // Initialize remaining burst times and waiting times
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        wt[i] = 0;
    }

    int time = 0;

    while (1) {
        int done = 1;

        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                done = 0;

                if (rem_bt[i] > quantum) {
                    time += quantum;
                    rem_bt[i] -= quantum;
                } else {
                    time += rem_bt[i];
                    wt[i] = time - bt[i];
                    rem_bt[i] = 0;
                }
            }
        }

        if (done)
            break;
    }
}

// Function to calculate turnaround time
void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

// Function to calculate and print averages
void findAverageTimes(int processes[], int n, int wt[], int tat[]) {
    float total_wt = 0, total_tat = 0;

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
}
```

```
// Round Robin Scheduling
void roundRobinScheduling(int processes[], int n, int bt[], int quantum) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, wt, quantum);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    printf("-------------------------------------------------------------\n");

    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n",
               processes[i], bt[i], wt[i], tat[i]);
    }

    findAverageTimes(processes, n, wt, tat);
}

int main() {
    int n, quantum;

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

    if (n <= 0) {
        printf("Invalid number of processes.\n");
        return 1;
    }

    int processes[n], burst_time[n];

    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("P%d: ", i + 1);
        scanf("%d", &burst_time[i]);
    }

    printf("Enter the time quantum: ");
    scanf("%d", &quantum);

    if (quantum <= 0) {
        printf("Invalid time quantum.\n");
        return 1;
    }

    roundRobinScheduling(processes, n, burst_time, quantum);

    return 0;
```

}