FCFS

```c
#include <stdio.h>

// Structure to store process details
struct process {
    int pid;   // Process ID
    int at;    // Arrival Time
    int tat;   // Turnaround Time
    int ct;    // Completion Time
    int bt;    // Burst Time
    int wt;    // Waiting Time
};

int main() {
    int n, i, j, temp;
    float avg1, avg2;

    // Asking the user to input the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    // Declare an array of processes
    struct process p[n];

    // Input the details of each process
    for (i = 0; i < n; i++) {
        printf("Enter Pid: ");
        scanf("%d", &p[i].pid);  // Process ID input
        printf("Enter AT (Arrival Time): ");
        scanf("%d", &p[i].at);   // Arrival Time input
        printf("Enter BT (Burst Time): ");
```

```
        scanf("%d", &p[i].bt);   // Burst Time input
    }

    // Sorting the processes based on Arrival Time using
Bubble Sort
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at) {
                // Swap Arrival Time (AT)
                temp = p[j].at;
                p[j].at = p[j + 1].at;
                p[j + 1].at = temp;

                // Swap Burst Time (BT)
                temp = p[j].bt;
                p[j].bt = p[j + 1].bt;
                p[j + 1].bt = temp;

                // Swap Process ID (PID)
                temp = p[j].pid;
                p[j].pid = p[j + 1].pid;
                p[j + 1].pid = temp;
            }
        }
    }

    // Print out the processes after sorting by Arrival
Time
    printf("Processes are: \n");
    for (i = 0; i < n; i++) {
        printf("Pid: %d, AT: %d, BT: %d\n", p[i].pid,
p[i].at, p[i].bt);
```

```
    }

    // Initialize current time (curt) and sum variables
for average calculations
    int curt = 0, sum1 = 0, sum2 = 0;
    for (i = 0; i < n; i++) {
        // If the current time is less than the arrival
time of the process,
        // set the current time to the arrival time (curt
needs to wait for the process to arrive)
        if (curt < p[i].at) {
            curt = p[i].at;
        }

        // Completion Time (CT) = current time + Burst
Time
        p[i].ct = curt + p[i].bt;

        // Update the current time to the completion time
of this process
        curt = p[i].ct;

        // Turnaround Time (TAT) = Completion Time (CT) -
Arrival Time (AT)
        p[i].tat = p[i].ct - p[i].at;

        // Waiting Time (WT) = Turnaround Time (TAT) -
Burst Time (BT)
        p[i].wt = p[i].tat - p[i].bt;
    }
```

```
    // Print the process details after calculating CT,
TAT, and WT
    printf("After calculating:\n");
    for (i = 0; i < n; i++) {
        printf("Pid: %d, AT: %d, BT: %d, CT: %d, TAT: %d,
WT: %d\n\n",
               p[i].pid, p[i].at, p[i].bt, p[i].ct,
p[i].tat, p[i].wt);

        // Accumulate the Turnaround Time and Waiting
Time for average calculation
        sum1 += p[i].tat;
        sum2 += p[i].wt;
    }

    // Calculate the average Turnaround Time (TAT) and
Waiting Time (WT)
    avg1 = (float)sum1 / n;
    avg2 = (float)sum2 / n;

    // Print the average Turnaround Time and Waiting Time
    printf("AVG TAT: %f\n", avg1);
    printf("AVG WT: %f\n", avg2);

    return 0;
}
```

SJF(Non-Preemtive)

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

// Structure to store process information
struct process {
    int pid;    // Process ID
    int at;     // Arrival Time
    int bt;     // Burst Time
    int tat;    // Turnaround Time
    int wt;     // Waiting Time
    int ct;     // Completion Time
    int st;     // Start Time
};

int main() {
    int n, i;

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

    // Declare an array of structures to store process
information
    struct process p[n];

    // Input arrival time for each process
    printf("Enter arrival time:\n");
    for(i = 0; i < n; i++) {
```

```
        printf("AT of P%d: ", i);
        scanf("%d", &p[i].at);
        p[i].pid = i;  // Set the process ID to the index
    }

    // Input burst time for each process
    printf("Enter burst time:\n");
    for(i = 0; i < n; i++) {
        printf("BT of P%d: ", i);
        scanf("%d", &p[i].bt);
    }

    // Print the input process details
    printf("\nPID-------AT--------BT\n");
    for(i = 0; i < n; i++) {
        printf("P%d--------%d---------%d\n", p[i].pid,
p[i].at, p[i].bt);
    }

    // Variable initialization for the scheduling process
    int complete = 0;  // To count how many processes are
completed
    bool is_completed[100] = {false};  // Boolean array
to track if the process is completed
    int curtime = 0;  // Current time initialization
    int sum1 = 0, sum2 = 0;  // Variables to accumulate
Turnaround Time and Waiting Time

    // Main scheduling loop
    while(complete != n) {
        int min_idx = -1;  // Index of the process with
the shortest burst time
```

```
        int mini_bust = INT_MAX;  // Initially set to
maximum possible value for comparison

        // Loop through all processes to find the next
process to execute
        for(i = 0; i < n; i++) {
            // Process must have arrived before or at the
current time and not be completed
            if(p[i].at <= curtime && !is_completed[i]) {
                // Select process with the shortest burst
time
                if(p[i].bt < mini_bust) {
                    mini_bust = p[i].bt;
                    min_idx = i;
                }
                // If burst time is the same, choose the
process with the earliest arrival time
                if(p[i].bt == mini_bust && p[i].at <
p[min_idx].at) {
                    mini_bust = p[i].bt;
                    min_idx = i;
                }
            }
        }

        // If no process is ready to be executed, move
the time forward
        if(min_idx == -1) {
            curtime++;
        } else {
            // Update the process details
```

```
            p[min_idx].st = curtime;  // Start time of
the process
            p[min_idx].ct = p[min_idx].st +
p[min_idx].bt;  // Completion time (start time + burst
time)
            p[min_idx].tat = p[min_idx].ct -
p[min_idx].at;  // Turnaround time (completion time -
arrival time)
            p[min_idx].wt = p[min_idx].tat -
p[min_idx].bt;  // Waiting time (turnaround time - burst
time)

            // Mark the process as completed
            is_completed[min_idx] = true;
            complete++;  // Increment the count of
completed processes
            curtime = p[min_idx].ct;  // Update the
current time to the process completion time
        }
    }

    // Print the results after all processes are
completed

printf("\nPID-------AT--------BT--------CT--------TAT----
---WT\n");
    for(i = 0; i < n; i++) {

printf("P%d--------%d---------%d---------%d---------%d---
------%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].ct,
p[i].tat, p[i].wt);
```

```
        sum1 += p[i].tat;  // Add Turnaround Time to the
total
        sum2 += p[i].wt;   // Add Waiting Time to the
total
    }

    // Calculate and print the average Turnaround Time
and Waiting Time
    float avg1 = (float)sum1 / n;  // Average Turnaround
Time
    float avg2 = (float)sum2 / n;  // Average Waiting
Time
    printf("AVG TAT: %f\nAVG WT: %f\n", avg1, avg2);

    return 0;
}
```

Pre-emtive SJF

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

// Structure to represent a process
struct process {
    int pid, at, bt, tat, wt, ct;  // PID, Arrival Time,
Burst Time, Turnaround Time, Waiting Time, Completion
Time
};

int main() {
```

```
    int n, i;

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

    // Declaring an array of 'n' processes and array to
store remaining burst times
    struct process p[n];
    int rem_bst[n];  // Array to keep track of remaining
burst time for each process

    // Input arrival times for each process
    printf("Enter arrival times:\n");
    for (i = 0; i < n; i++) {
        printf("AT of P%d: ", i);
        scanf("%d", &p[i].at);
        p[i].pid = i;  // Assigning Process ID (starting
from 0)
    }

    // Input burst times for each process
    printf("Enter burst times:\n");
    for (i = 0; i < n; i++) {
        printf("BT of P%d: ", i);
        scanf("%d", &p[i].bt);
        rem_bst[i] = p[i].bt;  // Initializing the
remaining burst time
    }

    // Display the entered processes (PID, AT, BT)
    printf("\nPID-------AT--------BT\n");
```

```
    for (i = 0; i < n; i++) {
        printf("P%d--------%d---------%d\n", p[i].pid,
p[i].at, p[i].bt);
    }

    // Initializing variables for process completion
tracking
    int complete = 0;  // Number of completed processes
    bool is_completed[n];  // Array to track whether a
process is completed
    for (i = 0; i < n; i++) {
        is_completed[i] = false;  // Initially, no
process is completed
    }

    int curtime = 0;  // Start time of the scheduler
(initially 0)

    // Main scheduling loop to simulate the execution of
processes
    while (complete != n) {  // Keep running until all
processes are completed
        int min_burst = INT_MAX;  // Variable to keep
track of the minimum remaining burst time
        int min_idx = -1;  // Index of the process with
the minimum burst time

        // Loop through all processes to find the process
with the smallest remaining burst time
        for (i = 0; i < n; i++) {
            // Process is eligible if its arrival time is
<= current time and it's not yet completed
```

```
            if (p[i].at <= curtime && !is_completed[i]) {
                // If a process has a smaller remaining
burst time, select it
                if (rem_bst[i] < min_burst) {
                    min_burst = rem_bst[i];
                    min_idx = i;
                }
                // If burst times are equal, choose the
one with the earliest arrival time
                else if (rem_bst[i] == min_burst &&
p[i].at < p[min_idx].at) {
                    min_burst = rem_bst[i];
                    min_idx = i;
                }
            }
        }

        // If no process is eligible (min_idx == -1),
increment the current time (curtime)
        if (min_idx == -1) {
            curtime++;
        } else {
            // Process the selected process by reducing
its remaining burst time
            rem_bst[min_idx]--;
            curtime++;  // Increment current time
(simulating the passage of time)

            // If the process is completed (remaining
burst time reaches 0), calculate completion, turnaround,
and waiting times
            if (rem_bst[min_idx] == 0) {
```

```
                p[min_idx].ct = curtime;  // Completion
time is the current time
                p[min_idx].tat = p[min_idx].ct -
p[min_idx].at;  // Turnaround time: CT - AT
                p[min_idx].wt = p[min_idx].tat -
p[min_idx].bt;  // Waiting time: TAT - BT
                complete++;  // Increment the number of
completed processes
                is_completed[min_idx] = true;  // Mark
the process as completed
            }
        }
    }

    // Output the results (Completion Time, Turnaround
Time, and Waiting Time for each process)

printf("\nPID-------AT--------BT--------CT--------TAT----
---WT\n");
    int sum_tat = 0, sum_wt = 0;  // Variables to
calculate average turnaround time and waiting time

    // Loop through all processes and print their details
    for (i = 0; i < n; i++) {

printf("P%d--------%d---------%d---------%d---------%d---
------%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].ct,
p[i].tat, p[i].wt);
        sum_tat += p[i].tat;  // Accumulate the
turnaround time
```

```
        sum_wt += p[i].wt;     // Accumulate the waiting
time
    }

    // Calculate the average turnaround time and average
waiting time
    float avg_tat = (float)sum_tat / n;  // Average
Turnaround Time
    float avg_wt = (float)sum_wt / n;    // Average
Waiting Time

    // Print the averages
    printf("AVG TAT: %.2f\n", avg_tat);  // Average
Turnaround Time
    printf("AVG WT: %.2f\n", avg_wt);    // Average
Waiting Time

    return 0;
}
```

Round Robin

```
#include<stdio.h>
#include<stdbool.h>

// Structure to represent each process and its details
struct process {
    int pid, at, bt, ct, tat, wt, st, rem_bt;
```

```
    // pid: Process ID
    // at: Arrival Time
    // bt: Burst Time
    // ct: Completion Time
    // tat: Turnaround Time
    // wt: Waiting Time
    // st: Start Time
    // rem_bt: Remaining Burst Time (to keep track during
execution)
};

int main() {
    int i, n, j, front = -1, rear = -1, quantum;

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

    struct process p[n], temp;
    int queue[100]; // Queue for managing the order of
process execution

    // Input Arrival Times for each process
    printf("Enter Arrival time:\n");
    for(i = 0; i < n; i++) {
        printf("P%d AT:", i);
        scanf("%d", &p[i].at);
        p[i].pid = i; // Assigning process IDs
sequentially
    }

    // Input Burst Times for each process
```

```
    printf("Enter Burst time:\n");
    for(i = 0; i < n; i++) {
        printf("P%d BT:", i);
        scanf("%d", &p[i].bt);
        p[i].rem_bt = p[i].bt; // Initializing remaining
burst time with burst time
    }

    // Input time quantum for the round robin
    printf("Enter time quantum:");
    scanf("%d", &quantum);

    // Sorting processes based on arrival time using
Bubble Sort (for scheduling order)
    for(i = 0; i < n - 1; i++) {
        for(j = 0; j < n - 1 - i; j++) {
            if(p[j].at > p[j + 1].at) {
                temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    // Display sorted processes for reference
    printf("\nPID-------AT--------BT\n");
    for(i = 0; i < n; i++) {
        printf("P%d--------%d---------%d\n", p[i].pid,
p[i].at, p[i].bt);
    }

    int completed = 0; // Count of completed processes
```

```
    bool visited[100] = {false}; // Array to keep track
of processes added to queue

    // Initialize queue and add first process
    front = rear = 0;
    queue[rear] = 0;
    int curtime = 0, indx, max;

    visited[0] = true; // Mark first process as visited

    // Main loop to process all processes until
completion
    while(completed != n) {
        indx = queue[front]; // Get index of current
process
        front++; // Move front pointer

        // First-time execution of a process
        if(p[indx].rem_bt == p[indx].bt) {
            if(curtime < p[indx].at) // If CPU idle time
needed
                max = p[indx].at;
            else
                max = curtime;

            p[indx].st = max; // Set start time
            curtime = p[indx].st; // Update current time
        }

        // If remaining burst time is greater than the
time quantum, process partially
        if(p[indx].rem_bt - quantum > 0) {
```

```
            p[indx].rem_bt -= quantum; // Reduce
remaining burst time
            curtime += quantum; // Increment current time
by quantum
        }
        // If remaining burst time is less than or equal
to the time quantum, complete the process
        else {
            curtime += p[indx].rem_bt; // Move current
time by remaining burst time
            p[indx].rem_bt = 0; // Process is now
complete
            completed++; // Increment completed count
            p[indx].ct = curtime; // Set completion time
            p[indx].tat = p[indx].ct - p[indx].at; //
Calculate turnaround time
            p[indx].wt = p[indx].tat - p[indx].bt; //
Calculate waiting time
        }

        // Check for new processes arriving while current
process is running
        for(i = 1; i < n; i++) {
            if(p[i].rem_bt > 0 && p[i].at <= curtime &&
visited[i] == false) {
                queue[++rear] = i; // Add process to
queue
                visited[i] = true; // Mark process as
visited
            }
        }
```

```
        // If the current process is incomplete, re-add
it to the queue
        if(p[indx].rem_bt > 0) {
            queue[++rear] = indx;
        }

        // If queue is empty, find next process (to
handle idle CPU time)
        if(front > rear) {
            for(i = 1; i < n; i++) {
                if(p[i].rem_bt > 0) {
                    queue[rear++] = i; // Add next
available process to queue
                    visited[i] = true;
                    break;
                }
            }
        }
    }

    // Output the completion, turnaround, and waiting
times for each process
    printf("\n");
    int sum1 = 0;
    int sum2 = 0;


printf("PID-------AT--------BT--------CT--------TAT------
-WT\n");
    for(i = 0; i < n; i++) {
```

```
printf("P%d--------%d---------%d---------%d---------%d---
------%d\n",
            p[i].pid, p[i].at, p[i].bt, p[i].ct,
p[i].tat, p[i].wt);
        sum1 += p[i].tat;
        sum2 += p[i].wt;
    }

    // Calculate and print average Turnaround Time (TAT)
and Waiting Time (WT)
    float avg1 = (float)sum1 / n;
    float avg2 = (float)sum2 / n;
    printf("AVG TAT: %f\nAVG WT: %f", avg1, avg2);

    return 0;
}
```

FIFO Page Replacement Algorithm

```
#include <stdio.h>
```

```c
int main() {
    // N: Total number of pages in the reference string
    // M: Total number of frames available in memory
    // pageFaults: Counter for the number of page faults
that occur
    // pos: Position pointer for FIFO replacement
    // flag: Indicator to check if a page is already in
memory
    int N, M, pageFaults = 0, pos = 0, flag = 0;

    // Input: Total number of pages
    printf("Enter the total number of pages: ");
    scanf("%d", &N);

    // Array to store the page reference string
    int pages[N];
    printf("Enter the page reference string: ");
    for (int i = 0; i < N; i++) {
        scanf("%d", &pages[i]);  // Input each page in
the reference string
    }

    // Input: Total number of frames (fixed memory slots)
    printf("Enter the total number of frames: ");
    scanf("%d", &M);

    // Array to represent frames; initially set to -1
indicating empty slots
    int frames[M];
    for (int i = 0; i < M; i++) {
        frames[i] = -1;  // -1 indicates an empty frame
```

```
    }

    // Processing each page in the reference string
    for (int i = 0; i < N; i++) {
        flag = 0;  // Reset flag for each page reference

        // Check if the page is already in one of the
frames (page hit)
        for (int j = 0; j < M; j++) {
            if (frames[j] == pages[i]) {
                flag = 1;  // Page is already in memory
(hit)
                break;     // Exit the loop since page is
found
            }
        }

        // If the page is not in memory (page fault)
        if (flag == 0) {
            // Replace the page at the 'pos' position
(FIFO replacement)
            frames[pos] = pages[i];
            pos = (pos + 1) % M;  // Update 'pos' to the
next frame, wrapping around using modulo
            pageFaults++;         // Increment the page
fault counter
        }
    }

    // Output the total number of page faults
    printf("Total number of page faults: %d\n",
pageFaults);
```

```
    return 0;
}
```

Optimal Page Replacement Algorithm

```
#include <stdio.h>

// Function to find the index of the frame to be replaced
based on the optimal page replacement policy
int find(int pages[], int frames[], int totalframe, int
totalpage, int curIndx) {
    int far = curIndx;  // Tracks the farthest future
occurrence of a page in frames
    int indx = -1;      // Holds the index of the frame
to replace, initially -1
    int i, j;

    // Iterate over each frame to check when each page in
frames is next used
    for (i = 0; i < totalframe; i++) {
        for (j = curIndx; j < totalpage; j++) {
            // If the current frame page is found in the
remaining pages
            if (frames[i] == pages[j]) {
                // Update 'far' and 'indx' if the page
appears later in the page stream
                if (j > far) {
```

```
                    far = j;
                    indx = i;
                }
                break;
            }
        }

        // If the frame page does not appear again in the
future pages, it should be replaced
        if (j == totalpage) {
            return i;
        }
    }
    // If no page found that will not be used in future,
return the index of the page farthest in future
    return (indx == -1) ? 0 : indx;
}

// Function to check if a page is already present in the
frames (page hit)
int isPresent(int frames[], int page, int totalframe) {
    for (int i = 0; i < totalframe; i++) {
        if (frames[i] == page) {
            return 1;  // Page is found in frames,
indicating a page hit
        }
    }
    return 0;  // Page is not found in frames, indicating
a page fault
}
```

```
// Function to perform the Optimal Page Replacement
algorithm
void optimalPage(int pages[], int totalframe, int
totalpage) {
    int i, j, curIndex = 0, fault = 0;  // curIndex
tracks filled frames, fault counts page faults
    int frames[totalframe];  // Array to hold the frames

    // Initialize all frames to -1 (empty)
    for (i = 0; i < totalframe; i++) {
        frames[i] = -1;
    }

    // Loop through each page in the page reference
string
    for (i = 0; i < totalpage; i++) {
        // If the page is not in frames (page fault)
        if (!isPresent(frames, pages[i], totalframe)) {
            // If there is an empty frame, place the page
there
            if (curIndex < totalframe) {
                frames[curIndex++] = pages[i];
            } else {
                // If no empty frame, find the optimal
frame to replace
                int optimalIndx = find(pages, frames,
totalframe, totalpage, i + 1);
                frames[optimalIndx] = pages[i];
            }
            fault++;  // Increment the page fault count
        }
    }
```

```
    // Print the total number of page faults and page
hits
    printf("\n\nThe total page faults: %d", fault);
    printf("\n\nThe total page hits: %d\n", totalpage -
fault);
}

int main() {
    int totalpage, totalframe, n, j, i;

    // Input total number of pages in the page reference
string
    printf("Enter number of pages: ");
    scanf("%d", &totalpage);

    // Input the number of frames
    printf("Enter frame number: ");
    scanf("%d", &totalframe);

    // Input the page reference string
    printf("Enter sequence stream: ");
    int pages[totalpage], frames[totalframe];
    for (i = 0; i < totalpage; i++) {
        scanf("%d", &pages[i]);
    }

    // Execute the Optimal Page Replacement algorithm
    optimalPage(pages, totalframe, totalpage);
    return 0;
}
```

LRU Page Replacement Algorithm

```c
#include<stdio.h>

// Function to find the position of the least recently
used page
int find(int time[], int m) {
    int pos = 0, i;
    int min = time[0];

    // Find the minimum time value in the array,
indicating the least recently used page
    for(i = 1; i < m; i++) {
        if(time[i] < min) {
            min = time[i];
            pos = i;
        }
    }

    // Return the position of the least recently used
page
    return pos;
}

int main() {
    int i, j, n, m;
    printf("Enter number of pages: "); // Total number of
pages
    scanf("%d", &n);
```

```
    printf("Enter number of frames: "); // Total number
of frames
    scanf("%d", &m);

    int fr[m], page[n], time[m];

    // Input the page reference string
    printf("Enter page reference string:");
    for(i = 0; i < n; i++) {
        scanf("%d", &page[i]);
    }

    // Initialize frames and time array with -1
(indicating empty frames initially)
    for(i = 0; i < m; i++) {
        fr[i] = -1;
        time[i] = 0;
    }

    int flag1 = 0, flag2 = 0, count = 0, fault = 0, pos;

    // Loop through each page in the reference string
    for(i = 0; i < n; i++) {
        flag1 = 0;
        flag2 = 0;

        // Check if the page is already in one of the
frames
        for(j = 0; j < m; j++) {
            if(fr[j] == page[i]) {
                count++; // Increment the time counter
```

```
                time[j] = count; // Update the usage time
for the page
                flag1 = 1; // Indicate that the page was
found (hit)
                break;
            }
        }

        // If the page was not found in frames (page
fault)
        if(flag1 == 0) {
            // Find an empty frame
            for(j = 0; j < m; j++) {
                if(fr[j] == -1) {
                    fr[j] = page[i]; // Place the page in
the empty frame
                    count++;
                    time[j] = count; // Update the time
                    flag2 = 1; // Indicate that an empty
frame was found
                    fault++; // Increment the page fault
count
                    break;
                }
            }

            // If no empty frame was found, we need to
replace the least recently used page
            if(flag2 == 0) {
                pos = find(time, m); // Find the LRU page
position using the `find` function
```

```
                fr[pos] = page[i]; // Replace the LRU
page with the current page
                count++;
                time[pos] = count; // Update the time for
the new page
                fault++; // Increment the page fault
count
            }
        }
    }

    // Output the total page faults and page hits
    printf("Faults: %d\n", fault);
    printf("Hits: %d\n", n - fault); // Total hits is the
difference between total pages and faults

    return 0;
}
```