Algorithm for FCFS (First Come First Serve) Scheduling

1. **Input:** Number of processes, arrival time, and burst time of each process.
2. **Sort** all processes based on their **arrival time** (if two processes have the same arrival time, maintain their original order).
3. **Initialize:**
   - `current_time = 0`
   - `completion_time = 0`
4. **For each process in the sorted order:**
   - If `current_time < arrival_time`, wait until the process arrives.
   - Process starts execution at `max(current_time, arrival_time)`.
   - Compute **completion time**:

     Completion Time=Start Time+Burst Time\text{Completion Time} = \text{Start Time} + \text{Burst Time}

   - Compute **Turnaround Time (TAT)**:

     TAT=Completion Time−Arrival TimeTAT = \text{Completion Time} - \text{Arrival Time}

   - Compute **Waiting Time (WT)**:

     WT=TAT−Burst TimeWT = TAT - \text{Burst Time}

   - Update `current_time` to the completion time of the current process.
5. **Output:** Average waiting time and turnaround time.

Algorithm for SJF (Shortest Job First) Scheduling – Non-Preemptive

1. **Input:** Number of processes, arrival time, and burst time of each process.
2. **Sort** all processes based on **arrival time**.
3. **Initialize:**
   - `current_time = 0`
   - `completed = 0` (to track completed processes)
4. **Repeat until all processes are completed:**
   - Select the process with the **shortest burst time** among the available (arrived) processes.
   - If two processes have the same burst time, choose the one that arrived first.
   - Process starts execution at `max(current_time, arrival_time)`.
   - Compute **completion time**:

     Completion Time=Start Time+Burst Time\text{Completion Time} = \text{Start Time} + \text{Burst Time}

   - Compute **Turnaround Time (TAT)**:

     TAT=Completion Time−Arrival TimeTAT = \text{Completion Time} - \text{Arrival Time}

   - Compute **Waiting Time (WT)**:

     WT=TAT−Burst TimeWT = TAT - \text{Burst Time}

    - Update `current_time` to the completion time of the current process.
    - Mark the process as completed (`completed += 1`).
5. **Output:** Average waiting time and turnaround time.

Algorithm for SRTF (Shortest Remaining Time First)

1. **Input:** Number of processes, arrival time, and burst time of each process.
2. **Initialize:**
    - Current time = 0
    - Remaining burst time array (same as burst time initially)
    - Keep track of completed processes (count = 0)
    - Maintain a flag for process completion
3. **Repeat until all processes are completed:**
    - Find the process with the shortest remaining time among the arrived processes.
    - If two processes have the same remaining time, choose the one that arrived first.
    - Execute the selected process for **one unit of time**.
    - Decrease its remaining burst time by 1.
    - If the process completes (remaining time = 0), mark it as finished, record completion time, and increase the completed count.
    - Increment the current time.
4. **Compute the Turnaround Time (TAT) and Waiting Time (WT):**
    - TAT=CompletionTime−ArrivalTimeTAT = Completion Time - Arrival Time
    - WT=TAT−BurstTimeWT = TAT - Burst Time
5. **Output:** Average waiting time and turnaround time.

Algorithm for Non-Preemptive Priority Scheduling

1. **Input:** Number of processes, arrival time, burst time, and priority of each process.
2. **Sort** all processes based on **arrival time**.
3. **Initialize:**
    - `current_time = 0`
    - `completed = 0` (to track completed processes)
4. **Repeat until all processes are completed:**
    - Select the process with the **highest priority** (lowest priority number) among the available (arrived) processes.
    - If two processes have the same priority, choose the one that arrived first.
    - Process starts execution at `max(current_time, arrival_time)`.
    - Compute **completion time**:

      Completion Time=Start Time+Burst Time\text{Completion Time} = \text{Start Time} + \text{Burst Time}

    - Compute **Turnaround Time (TAT)**:

      TAT=Completion Time−Arrival TimeTAT = \text{Completion Time} - \text{Arrival Time}

    - Compute **Waiting Time (WT)**:

      WT=TAT−Burst TimeWT = TAT - \text{Burst Time}

    - Update `current_time` to the completion time of the current process.
    - Mark the process as completed (`completed += 1`).
5. **Output:** Average waiting time and turnaround time.

Algorithm for IPC Using Shared Memory

1. **Generate a Unique Key:**
   - Use `ftok("shmfile", 65)` to generate a unique key for shared memory identification.
2. **Create/Access Shared Memory Segment:**
   - Writer: `shmget(key, 1024, 0666 | IPC_CREAT)` to create a shared memory segment of size 1024 bytes.
   - Reader: `shmget(key, 1024, 0666)` to access the existing shared memory.
3. **Attach Shared Memory to Process:**
   - Use `shmat(shmid, NULL, 0)` to attach the shared memory segment to the process's address space.
4. **Write Data (Writer Process):**
   - Prompt the user to enter a string.
   - Store the string in shared memory using `fgets(data, 1024, stdin)`.
5. **Read Data (Reader Process):**
   - Access the shared memory and retrieve the stored data.
   - Print the received string using `printf("Data from writer: %s", data)`.
6. **Detach Shared Memory:**
   - Use `shmdt(data)` to detach the shared memory segment from the process.
7. **Destroy Shared Memory (Optional):**
   - Use `shmctl(shmid, IPC_RMID, NULL)` to remove the shared memory segment after usage.

Algorithm for Producer-Consumer Problem Using Semaphores

1. **Initialize Semaphores and Variables:**
   - `mutex = 1` → Controls **mutual exclusion**.
   - `full = 0` → Tracks **filled buffer slots**.
   - `empty = N` → Tracks **empty buffer slots** (where `N` is the buffer size).
   - `x = 0` → Item count.
2. **Define Semaphore Operations:**
   - `wait(s)` → Decrements semaphore `s` (`s--`).
   - `signal(s)` → Increments semaphore `s` (`s++`).
3. **Producer Process:**
   - Check if the buffer has **empty slots** (`empty != 0`).
   - Acquire `mutex` (`wait(mutex)`).
   - Produce an item (`x++`).
   - Release `mutex` (`signal(mutex)`).
   - Increment `full` (`signal(full)`).
4. **Consumer Process:**
   - Check if the buffer has **filled slots** (`full != 0`).
   - Acquire `mutex` (`wait(mutex)`).
   - Consume an item (`x--`).
   - Release `mutex` (`signal(mutex)`).
   - Increment `empty` (`signal(empty)`).
5. **Repeat Until User Exits:**
   - Display options:
     - `1` → Producer produces an item.
     - `2` → Consumer consumes an item.
     - `3` → Exit.
   - If the buffer is **full**, the producer waits.
   - If the buffer is **empty**, the consumer waits.

Algorithm for Banker's Algorithm

1. **Input the Number of Processes and Resources:**
   - Read the total number of processes (`pno`).
   - Read the total number of resource types (`r`).
2. **Input the Available Resources:**
   - Read the available instances for each resource type into `aval[r]`.
3. **Input Process Details:**
   - For each process `Pi`:
     - Read the **Allocation Matrix** (`all[i][r]`) – resources currently allocated.
     - Read the **Maximum Matrix** (`max[i][r]`) – maximum resources required.
     - Compute the **Need Matrix**:

       need[i][j]=max[i][j]−all[i][j]need[i][j] = max[i][j] - all[i][j]

     - Mark all processes as **not yet executed** (`flag = 1`).
4. **Display Process Details:**
   - Print the **Allocation, Maximum, and Need matrices**.
5. **Check for a Safe Sequence (Safety Algorithm):**
   - Initialize `count = 0` (number of completed processes).
   - **Repeat until all processes are executed (`count != pno`)**:
     - Find a process `Pi` that is not yet executed (`flag = 1`) and has `need[i] ≤ available` for all resources.
     - If found:
       - Allocate its resources to `available[]`:

         available[j]+=all[i][j]available[j] += all[i][j]

       - Mark process `Pi` as executed (`flag = 0`).
       - Add `Pi` to the **safe sequence**.
       - Increment `count`.
     - If no process is found, declare **unsafe state** and terminate.
6. **Print the Safe Sequence (If Exists):**
   - If all processes are executed, print the safe sequence.
   - Otherwise, declare the system as **unsafe**.

Algorithm for Least Recently Used (LRU) Page Replacement

1. **Input the Number of Frames & Pages:**
   - Read `frames` (number of available page frames).
   - Read `n` (number of pages in the reference string).
2. **Initialize Data Structures:**
   - Maintain a **page table** (stores pages currently in memory).
   - Use a **stack or queue** (to track page usage order).
   - Initialize all frames as **empty (-1)**.
3. **Process Each Page Request:**
   - If the page is **already in memory**, move it to the **most recently used** position.
   - If the page is **not in memory (page fault)**:
     - If a free frame is available, load the page into the frame.
     - If all frames are full, **replace the least recently used (LRU) page**:
       - Identify the page that **has not been used for the longest time**.
       - Remove it and load the new page.

    - Update the **usage order** after each access.
4. **Repeat Until All Pages Are Processed.**
5. **Compute Page Faults & Hits:**
    - Count the total **page faults** (number of times a new page is loaded).
    - Count the **page hits** (number of times a requested page is already in memory).
6. **Output the Page Replacement Steps & Page Fault Count.**

Algorithm for Best Fit Memory Allocation

1. **Input the Number of Memory Blocks & Processes:**
    - Read `m` (number of memory blocks).
    - Read `n` (number of processes).
2. **Initialize Memory Blocks & Processes:**
    - Store **size of each memory block** in `blockSize[m]`.
    - Store **size of each process** in `processSize[n]`.
    - Maintain an **allocation array** to track assigned blocks (`-1` means unallocated).
3. **For Each Process, Find the Best Fit Block:**
    - Set `bestIndex = -1` (to track the best-fitting block).
    - **Loop through all memory blocks**:
        - If the block is **large enough** for the process (`blockSize[j] >= processSize[i]`):
            - If `bestIndex == -1` OR `blockSize[j] < blockSize[bestIndex]`, update `bestIndex`.
    - If a suitable block is found (`bestIndex ≠ -1`):
        - Allocate the process to `blockSize[bestIndex]`.
        - Reduce the available size of that block.
4. **Repeat Until All Processes Are Allocated or No Suitable Blocks Exist.**
5. **Output the Memory Allocation Table:**
    - Display the **process number, size, and allocated block** (or `Not Allocated` if no block fits).