1. FCFS (First Come, First Served) Scheduling Algorithm:

1. Input: Array of processes with Arrival Time and Burst Time.
2. Initialize:
   - Set current_time = 0
   - Create an array completion_time to store completion times of each process.
3. For each process in the array:
   a. Set completion_time[i] = current_time + burst_time[i]
      - where burst_time[i] is the burst time of the ith process.
   b. Update current_time = completion_time[i]
4. Calculate Turnaround Time and Waiting Time for each process:
   - Turnaround Time = completion_time[i] - arrival_time[i]
     - where arrival_time[i] is the arrival time of the ith process.
   - Waiting Time = Turnaround Time - burst_time[i]
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

2. SJF (Shortest Job First) Preemptive Scheduling Algorithm:

1. Input: Array of processes with Arrival Time and Burst Time.
2. Initialize:
   - Set current_time = 0
   - Create a priority queue to store processes based on their burst time.
3. While there are processes to execute:
   a. Select the process with the shortest remaining burst time from the priority queue.
   b. Execute the process for a time quantum or until completion if shorter.
   c. Update current_time and remaining burst times of processes accordingly.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

3. SJF (Shortest Job First) Non-preemptive Scheduling Algorithm:

1. Input: Array of processes with Arrival Time and Burst Time.
2. Initialize:
   - Set current_time = 0
   - Sort processes based on their burst time (shortest to longest).
3. For each process in the sorted array:
   a. Set completion_time[i] = current_time + burst_time[i]
      - where burst_time[i] is the burst time of the ith process.
   b. Update current_time = completion_time[i]
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

4. Priority Scheduling (Preemptive and Non-preemptive) Algorithm:

1. Input: Array of processes with Arrival Time, Burst Time, and Priority.
2. Initialize:
   - Set current_time = 0
   - Create a priority queue or sort processes based on their priority (higher number indicates higher priority).
3. For each process in the priority queue or sorted array:
   a. Preemptive:
      - Select the highest priority process that has arrived and execute it.
      - Adjust current_time and remaining burst times as needed.
   b. Non-preemptive:
      - Execute each process in order of their priority until completion.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

5. Round Robin Scheduling Algorithm (using ready queue):

1. Input: Array of processes with Arrival Time and Burst Time, Time Quantum.
2. Initialize:
   - Set current_time = 0
   - Create a ready queue to store processes.
3. While there are processes in the ready queue:
   a. Enqueue arriving processes into the ready queue.
   b. Dequeue the front process from the ready queue and execute it for the time quantum.
   c. If the process still has burst time left, enqueue it back into the ready queue.
   d. Update current_time and remaining burst times of processes accordingly.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

6. Multi-level Queue Scheduling with FCFS Algorithm:

1. Input: Multiple queues with different scheduling algorithms (e.g., FCFS for each queue).
2. Initialize:
   - Set current_time = 0
   - Create multiple queues with different priorities or characteristics.
3. While there are processes in any of the queues:
   a. Select a process from the highest priority queue that has arrived.
   b. Execute the process according to the scheduling algorithm of that queue (e.g., FCFS).
   c. Update current_time and process queues accordingly.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

7. Rate-Monotonic Scheduling Algorithm:

1. Input: Array of processes with Periods (or deadlines) and Execution Times (or CPU bursts).
2. Initialize:
   - Assign priorities inversely proportional to the period (shorter period = higher priority).
3. Execute processes in priority order (higher priority first) using a preemptive scheduling algorithm.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

8. Earliest-Deadline First (EDF) Scheduling Algorithm:

1. Input: Array of processes with Deadlines and Execution Times.
2. Initialize:
   - Set current_time = 0
   - Sort processes based on their deadlines (earliest deadline first).
3. For each process in the sorted array:
   a. Execute the process until completion or until its deadline, whichever comes first.
   b. Update current_time and remaining burst times accordingly.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

9. Proportional Scheduling Algorithm:

1. Input: Array of processes with Arrival Time, Burst Time, and a weight (or priority).
2. Initialize:
   - Set current_time = 0
   - Assign priorities or weights to processes based on their provided weight.
3. Execute processes in priority order (higher weight or priority first) using a preemptive or non-preemptive scheduling algorithm.
4. Calculate Turnaround Time and Waiting Time for each process as in FCFS.
5. Output: completion_time, Turnaround Time, Waiting Time for each process.

10. Producer-Consumer Problem Algorithm:

1. Initialize:
   - Shared buffer between producer and consumer with a limited size.
   - Semaphores or locks for synchronization (e.g., mutex, empty, full).
2. Producer:
   a. Produce an item.
   b. Acquire mutex to enter the critical section.

c. Insert the item into the buffer.
d. Release mutex.
e. Signal full semaphore if buffer was empty.
3. Consumer:
a. Acquire full semaphore to consume from the buffer.
b. Acquire mutex to enter the critical section.
c. Remove an item from the buffer.
d. Release mutex.
e. Signal empty semaphore if buffer was full.
4. Repeat steps 2 and 3 in a loop.
5. Output: Produced and consumed items.

11. Dining Philosophers Problem Algorithm:

1. Initialize:
- Array of philosophers and their forks.
- Semaphores or mutexes for synchronization (one per fork).
2. Philosopher:
a. Think until hungry.
b. Pick up the left fork.
c. Pick up the right fork (if available).
d. Eat.
e. Put down the right fork.
f. Put down the left fork.
3. Repeat steps 2 in a loop.
4. Output: Philosophers eating in turns, avoiding deadlock.

12. Banker's Algorithm for Deadlock Avoidance:

1. Input: Allocation matrix, Max matrix, Available resources vector, and Need matrix.
2. Initialize:
- Work vector as available resources.
- Finish array to track processes that can complete.
3. While there are unfinished processes:
a. Find a process where Need[i] <= Work.
b. If such a process exists, allocate resources to it, mark it as finished, and release resources after completion.
c. If no such process exists, system is in an unsafe state (potential deadlock).
4. Output: Safe sequence or indicate deadlock.

13. Deadlock Detection Algorithm:

1. Input: Resource allocation graph or matrices (Allocation, Request, and Available).
2. Initialize:
   - Mark all processes as unfinished.
3. While there are unfinished processes:
   a. Find a process that can be completed with the available resources.
   b. Execute the process and release its allocated resources.
4. If all processes can complete, system is deadlock-free. Otherwise, deadlock detected.
5. Output: Processes involved in deadlock (if detected).

14. Worst Fit, Best Fit, First Fit Algorithms (for Memory Allocation):

1. Input: Memory block sizes and process sizes (requests).
2. Worst Fit:
   - Find the largest available memory block that can accommodate the process.
3. Best Fit:
   - Find the smallest available memory block that can accommodate the process.
4. First Fit:
   - Allocate the first available memory block that is large enough for the process.
5. Output: Allocated memory blocks for each process.

15. FIFO (First In, First Out) Page Replacement Algorithm:

1. Input: Page reference string and frame size.
2. Initialize:
   - Set of frames to store pages.
3. For each page reference in the string:
   a. If the page is already in a frame, continue.
   b. If a frame is available, allocate it to the page.
   c. If no frame is available, replace the page in the oldest frame (first-in).
4. Output: Page faults or hits for each reference.

16. LRU (Least Recently Used) Page Replacement Algorithm:

1. Input: Page reference string and frame size.
2. Initialize:
   - Set of frames to store pages.
   - Queue or stack to keep track of the order of page accesses.
3. For each page reference in the string:
   a. If the page is already in a frame, update its position in the queue or stack.
   b. If a frame is available, allocate it to the page and add it to the queue or stack.
   c. If no frame is available, replace the least recently used page (oldest page in the queue or stack).

4. Output: Page faults or hits for each reference.

17. Optimal Page Replacement Algorithm:

1. Input: Page reference string and frame size.
2. Initialize:
   - Set of frames to store pages.
   - Array or table to store future page references.
3. For each page reference in the string:
   a. If the page is already in a frame, continue.
   b. If a frame is available, allocate it to the page.
   c. If no frame is available, replace the page that will not be used for the longest time (future reference).
4. Output: Page faults or hits for each reference.