Q1
Deadlocks in the Problem Setup:
Deadlocks can occur in the dining philosophers problem due to resource contention. Each philosopher requires two forks to eat, and if each picks up one fork, they might reach a deadlock situation where no philosopher can proceed. For instance, if all philosophers pick up the fork on their left simultaneously, they will be stuck waiting for the fork on their right, leading to a deadlock where no one can eat.

Solution to Avoid Deadlocks:
The proposed solution uses mutexes to ensure exclusive access to forks. It includes a logic where the first two philosophers are allowed to eat initially to avoid potential deadlocks caused by all philosophers trying to pick up their left fork simultaneously. Then, the philosophers enter a loop where they think and eat in a controlled manner, avoiding deadlock scenarios by only locking both forks when ready to eat.

Fairness of the Solution:
By utilising a loop where all philosophers get chances to eat and think in a controlled manner, the solution aims to ensure fairness over multiple cycles. Each philosopher is given an opportunity to eat and increment their meal count, and the code ensures they stop when they reach the maximum meal count specified.

Fairness regarding meal consumption heavily relies on sleep durations, the number of cycles, and the number of philosophers. A balanced execution allows each philosopher to eat a similar number of meals over time.

Q2
Code Logic:
1. Initialization:Initialize semaphores for synchronization and prompt the user to input the car's capacity and the total number of passengers.

2. Passenger and Car Threads: The main function creates two types of threads: the car thread and multiple passenger threads.

3. Passenger Logic:
   - Passengers are created as individual threads.
   - Each passenger waits on the `boardable_mutex` semaphore until signaled by the car that boarding is possible.
   - Upon boarding, a passenger increments the `passengers_boarded` count, allowing the car to start the ride once fully boarded.

- After the ride, passengers wait for the car to signal the end of the journey (`end_journey_mutex`) before disembarking. They decrement the `passengers_boarded` count.

4. Car Logic:
   - The car thread loops indefinitely to simulate continuous rides.
   - It loads passengers by signaling the `boardable_mutex` semaphore for each seat.
   - After all seats are filled, the car signals `fullyboarded`, indicating it's ready to start the ride.
   - After the ride, the car unloads passengers by signaling the `end_journey_mutex`. Once all passengers are off, it signals `fullyunboarded`.

Concurrency Bug Avoidance:
1. Mutual Exclusion:
   - The `boardedcountmutex` semaphore ensures that only one thread can access `passengers_boarded` at a time to prevent race conditions during boarding and offboarding.

2. Semaphore Use:
   - Semaphores such as `boardable_mutex`, `end_journey_mutex`, `boardedcountmutex`, `fullyboarded`, and `fullyunboarded` control access to critical sections of the code, preventing multiple threads from accessing shared resources simultaneously.

3. Synchronization:
   - Proper synchronization via semaphores ensures that passengers only board when the car is ready and can only disembark when the ride ends, preventing concurrency issues like over-boarding or unloading before the ride finishes.

4. Avoidance of Deadlocks:
   - The code has been structured to avoid deadlocks by ensuring proper semaphore usage and avoiding nested lock conditions that could lead to resource contention.

Q3)
Semaphore Initialization:

left_go and right_go: Signals whether a car can proceed from the left or right side.
bridge_avail: Indicates if the bridge is available for traversal.
bridgecountvaravail: Manages the count of cars on the bridge.

Passing Function: Increments the count of cars on the bridge before passage.

Simulates a delay to mimic the passage time. Prints the direction of the cars moving across the bridge.Decrements the count of cars on the bridge after passage.Signals if the bridge is available after all cars have passed.

Thread Functions (left and right):
left and right threads represent cars on their respective sides.
Wait for signals (sem_wait) before proceeding to cross the bridge.

Main Function:
Inputs the number of cars on the left and right sides.
Creates threads for left and right side cars.
Controls the flow of cars by allowing groups of cars (defined by max_cars) to cross the bridge at a time.
Ensures that all cars are able to cross the bridge by managing the number of cars allowed to move based on the set limit (max_cars).

Concurrency Bugs Avoidance:

1. Semaphores are used to control the flow of cars, ensuring that only a limited number of cars (determined by max_cars) can access the bridge at a time.
2. Proper semaphore signals are used to manage entry to the bridge from both the left and right sides, preventing multiple cars from trying to cross simultaneously.