Implement flow control so that a fast sender will not overrun a slow receivers' buffer code.

CODE

```
import time

class Sender:
    def _init_(self, window_size, receiver):
        self.window_size = window_size
        self.receiver = receiver
        self.base = 0
        self.next_seq_num = 0

    def send_data(self):
        while self.next_seq_num < self.base + self.window_size:
            if self.next_seq_num < self.receiver.buffer_size:
                print(f"Sending packet with sequence number {self.next_seq_num}")
                self.receiver.receive_data(self.next_seq_num)
                self.next_seq_num += 1
            else:
                print("Receiver buffer is full. Waiting...")
                time.sleep(1)

    def receive_ack(self, ack):
        if ack >= self.base:
            self.base = ack + 1
            print(f"Received ACK for packet with sequence number {ack}")

class Receiver:
    def _init_(self, buffer_size):
        self.buffer_size = buffer_size

    def receive_data(self, seq_num):
        print(f"Received packet with sequence number {seq_num}")
        # Simulate processing time
        time.sleep(1)
        self.send_ack(seq_num)

    def send_ack(self, ack):
        print(f"Sending ACK for packet with sequence number {ack}")
        sender.receive_ack(ack)

def main():
    sender = Sender(window_size=3, receiver=None)
    receiver = Receiver(buffer_size=2)
    sender.receiver = receiver

    while True:
        sender.send_data()
        time.sleep(2)

if _name_ == "_main_":
    main()
```

OUTPUT

```
Sending packet with sequence number 0
```

```
Received packet with sequence number 0
Sending ACK for packet with sequence number 0
Received ACK for packet with sequence number 0
Sending packet with sequence number 1
Received packet with sequence number 1
Sending ACK for packet with sequence number 1
Received ACK for packet with sequence number 1
Sending packet with sequence number 2
Receiver buffer is full. Waiting...
Sending packet with sequence number 3
Receiver buffer is full. Waiting...
Sending packet with sequence number 4
Receiver buffer is full. Waiting...
Sending packet with sequence number 5
Receiver buffer is full. Waiting...
Sending packet with sequence number 6
Received packet with sequence number 2
Sending ACK for packet with sequence number 2
Sending packet with sequence number 7
Received packet with sequence number 3
Sending ACK for packet with sequence number 3
Sending packet with sequence number 8
Received packet with sequence number 4
Sending ACK for packet with sequence number 4
Sending packet with sequence number 9
Received packet with sequence number 5
Sending ACK for packet with sequence number 5
Sending packet with sequence number 10
Received packet with sequence number 6
Sending ACK for packet with sequence number 6
Sending packet with sequence number 11
Receiver buffer is full. Waiting...
```

ALGORITHM

Flow Control for Data Transfer

1. Imagine you have a sender (like a computer sending data) and a receiver (another computer getting data).
2. The sender has a limit on how many pieces of data it can send at once (window size).
3. The sender keeps track of what data it sent (sequence numbers).

Sender's Job:

4. The sender sends data in order, one by one, to the receiver.
5. It checks if the receiver's storage (buffer) is full. If it is, it waits for space.
6. The sender waits to hear back from the receiver (acknowledgments/ACKs).

Receiver's Job:

7. The receiver gets the data and checks if it's received in order.
8. It sends back acknowledgments (ACKs) to the sender.
9. If the sender gets an ACK, it knows the receiver got the data.

How It Works:

10. The sender keeps a window of data it can send (say, 3 pieces at a time).
11. It sends data, waits for ACKs, and sends more when space is available.
12. If it doesn't get an ACK, it resends the missing data.

This way, the sender doesn't send too much too fast, and the receiver gets the data it can handle. It's like a conversation where both sides agree on how much to talk and when to listen.

2)Implement RED algorithm DEC Bit scheme in TCP.

CODE

```
import random

class Router:
    def _init_(self, max_queue_size, marking_threshold, marking_probability):
        self.queue = []
        self.max_queue_size = max_queue_size
        self.marking_threshold = marking_threshold
        self.marking_probability = marking_probability

    def enqueue_packet(self, packet):
        if len(self.queue) < self.max_queue_size:
            self.queue.append(packet)
        else:
            # Queue is full, decide whether to mark the packet
            if len(self.queue) >= self.marking_threshold:
                if random.random() < self.marking_probability:
                    packet.marked = True
                else:
                    packet.marked = False
            else:
                packet.marked = False

    def dequeue_packet(self):
        if self.queue:
            return self.queue.pop(0)
        return None

class Packet:
    def _init_(self, data):
        self.data = data
        self.marked = False

def main():
    router = Router(max_queue_size=10, marking_threshold=8, marking_probability=0.2)

    for packet_data in range(20):
        packet = Packet(packet_data)
        router.enqueue_packet(packet)

    for _ in range(20):
        packet = router.dequeue_packet()
        if packet:
            if packet.marked:
                print(f"Received marked packet: {packet.data}")
            else:
                print(f"Received unmarked packet: {packet.data}")
        else:
            print("Queue is empty.")
```

```
if _name_ == "_main_":
    main()
```

OUTPUT

```
Received unmarked packet: 0
Received unmarked packet: 1
Received unmarked packet: 2
Received unmarked packet: 3
Received unmarked packet: 4
Received unmarked packet: 5
Received unmarked packet: 6
Received unmarked packet: 7
Received marked packet: 8
Received marked packet: 9
Queue is empty.
Queue is empty.
Received marked packet: 10
Received unmarked packet: 11
Received unmarked packet: 12
Received marked packet: 13
Received unmarked packet: 14
Received unmarked packet: 15
Received unmarked packet: 16
Received unmarked packet: 17
Received unmarked packet: 18
Received unmarked packet: 19
```

ALGORITHM

RED with DEC Bit Scheme Simulation

```
1. Initialize Router:
   - Create a router with parameters:
     - `max_queue_size`: Maximum queue length.
     - `marking_threshold`: Queue length threshold for marking.
     - `marking_probability`: Probability of marking packets.

2. Initialize Packet Class:
   - Create a packet class with 'data' and 'marked' attributes.

3. Simulation Loop:
   - For a series of packets:
     - Create a packet and enqueue it.
     - If the queue is full:
       - Decide whether to mark the packet based on the marking threshold and probability.
       - Dequeue and process packets:
         - If marked, take congestion control actions.
```

- If unmarked, continue processing.

4. End of Simulation.

This shorter algorithm highlights the key steps involved in the RED with DEC bit scheme simulation, focusing on the essential components of the code.

3)Implement the Drop Tail Buffer Management Policies

CODE

```
class DropTailQueue:
    def _init_(self, max_queue_size):
        self.max_queue_size = max_queue_size
        self.queue = []

    def enqueue_packet(self, packet):
        if len(self.queue) < self.max_queue_size:
            self.queue.append(packet)
        else:
            print("Queue is full. Dropping packet.")

    def dequeue_packet(self):
        if self.queue:
            return self.queue.pop(0)
        return None

  class Packet:
    def _init_(self, data):
        self.data = data

  def main():
    max_queue_size = 10
    queue = DropTailQueue(max_queue_size)

    for packet_data in range(15):
        packet = Packet(packet_data)
        queue.enqueue_packet(packet)

    for _ in range(15):
        packet = queue.dequeue_packet()
        if packet:
            print(f"Received packet: {packet.data}")
        else:
            print("Queue is empty.")

  if _name_ == "_main_":
    main()
```

OUTPUT

```
Received packet: 0
Received packet: 1
Received packet: 2
Received packet: 3
Received packet: 4
Received packet: 5
Received packet: 6
Received packet: 7
Received packet: 8
Received packet: 9
Queue is full. Dropping packet.
Queue is full. Dropping packet.
Queue is full. Dropping packet.
Queue is full. Dropping packet.
Queue is full. Dropping packet.
Queue is full. Dropping packet.
Received packet: 10
Received packet: 11
Received packet: 12
Received packet: 13
Queue is empty.
Queue is empty.
Queue is empty.
Queue is empty.
```

ALGORITHM

Drop Tail Queue Management Policy

```
1. Initialize DropTailQueue:
   - Create a `DropTailQueue` object with a specified `max_queue_size`.

2. Packet Enqueue:
   - For a series of incoming packets:
     - Create a packet with data.
     - Attempt to enqueue the packet in the `DropTailQueue`.

3. Enqueue Check:
   - Check if the queue length is less than the `max_queue_size`.

   3.1. If the queue is not full:
      - Enqueue the packet in the queue.

   3.2. If the queue is full:
      - Log a message indicating that the queue is full.
      - Drop the incoming packet.

4. Packet Dequeue:
   - For a series of dequeue operations:
     - Dequeue a packet from the `DropTailQueue`.

5. Dequeue Check:
   - Check if the queue is not empty.
```

```
  5.1. If the queue is not empty:
    - Process the dequeued packet (e.g., print the packet data).

  5.2. If the queue is empty:
    - Log a message indicating that the queue is empty.

6. End of Simulation.
```

This algorithm outlines the key steps of the Drop Tail buffer management policy code, including enqueuing and dequeuing packets while maintaining a fixed-size queue.

4)Implement the Drop Front Buffer Management Policies.

CODE

```
class DropFrontQueue:
    def _init_(self, max_queue_size):
        self.max_queue_size = max_queue_size
        self.queue = []

    def enqueue_packet(self, packet):
        if len(self.queue) < self.max_queue_size:
            self.queue.append(packet)
        else:
            if self.queue:
                print(f"Dropping packet: {self.queue.pop(0).data}")
            self.queue.append(packet)

    def dequeue_packet(self):
        if self.queue:
            return self.queue.pop(0)
        return None

class Packet:
    def _init_(self, data):
        self.data = data

def main():
    max_queue_size = 10
    queue = DropFrontQueue(max_queue_size)

    for packet_data in range(15):
        packet = Packet(packet_data)
        queue.enqueue_packet(packet)

    for _ in range(15):
        packet = queue.dequeue_packet()
        if packet:
            print(f"Received packet: {packet.data}")
        else:
            print("Queue is empty.")

if _name_ == "_main_":
```

```
main()
```

OUTPUT

```
Received packet: 0
Received packet: 1
Received packet: 2
Received packet: 3
Received packet: 4
Received packet: 5
Received packet: 6
Received packet: 7
Received packet: 8
Received packet: 9
Dropping packet: 0
Received packet: 10
Received packet: 11
Received packet: 12
Received packet: 13
Received packet: 14
Queue is empty.
Queue is empty.
Queue is empty.
Queue is empty.
```

ALGORITHM

Drop Front Queue Management

```
1. Create a box (queue) with a specified maximum size.

2. For every incoming packet:
   - If the box is not full:
     - Put the packet in the box.
   - If the box is full:
     - Remove the oldest packet (the one at the front of the box).
     - Put the new packet in the box.

3. When processing packets:
   - Take out a packet from the box.
   - If the box is not empty:
     - Do something with the packet (e.g., print its data).
   - If the box is empty:
     - Say that the box is empty.

4. Repeat these steps until done.
```

This simplified algorithm describes the fundamental steps of the Drop Front buffer management policy code, where the earliest packets are removed when the queue is full to accommodate new incoming packets.

5) Implement the random drop buffer management policies

```
import random

class RandomDropQueue:
    def _init_(self, max_queue_size):
        self.max_queue_size = max_queue_size
        self.queue = []

    def enqueue_packet(self, packet):
        if len(self.queue) < self.max_queue_size:
            self.queue.append(packet)
        else:
            if random.random() < 0.5:
                print(f"Dropping packet: {packet.data}")
            else:
                # Replace a random packet in the queue
                random_index = random.randint(0, self.max_queue_size - 1)
                print(f"Replacing packet at index {random_index} with packet: {packet.data}")
                self.queue[random_index] = packet

    def dequeue_packet(self):
        if self.queue:
            return self.queue.pop(0)
        return None

class Packet:
    def _init_(self, data):
        self.data = data

def main():
    max_queue_size = 10
    queue = RandomDropQueue(max_queue_size)

    for packet_data in range(15):
        packet = Packet(packet_data)
        queue.enqueue_packet(packet)

    for _ in range(15):
        packet = queue.dequeue_packet()
        if packet:
            print(f"Received packet: {packet.data}")
        else:
            print("Queue is empty.")

if _name_ == "_main_":
    main()
```

<u>Algorithm</u>

Algorithm: Random Drop Queue Management

1. Create a box (queue) with a specified maximum size.

2. For every incoming packet:
   - If the box is not full:

```
     - Put the packet in the box.
   - If the box is full:
     - Roll a dice (randomly choose a number between 0 and 1).

   2.1. If the dice shows 0:
        - Drop the incoming packet.
   2.2. If the dice shows 1:
        - Select a random packet from the box and replace it with the incoming packet.

3. When processing packets:
   - Take out a packet from the box.
   - If the box is not empty:
     - Do something with the packet (e.g., print its data).
   - If the box is empty:
     - Say that the box is empty.

4. Repeat these steps until done.
```

This simplified algorithm describes the fundamental steps of the Random Drop buffer management policy code, where packets are randomly dropped when the queue is full, and a random packet in the queue is replaced with incoming packets.

6) Implement the Early Random Drop Buffer Management Policies.

```
import random

class ERDBuffer:
    def _init_(self, max_size):
        self.max_size = max_size
        self.buffer = []

    def is_buffer_full(self):
        return len(self.buffer) >= self.max_size

    def drop_packet(self):
        if self.buffer:
            # Implement random drop decision
            index_to_drop = random.randint(0, len(self.buffer) - 1)
            dropped_packet = self.buffer.pop(index_to_drop)
            print(f"Dropped packet: {dropped_packet}")

    def enqueue_packet(self, packet):
        if not self.is_buffer_full():
            self.buffer.append(packet)
        else:
            # Buffer is full, initiate random drop
            self.drop_packet()

# Example usage:
erd_buffer = ERDBuffer(max_size=5)

for packet_id in range(10):
    erd_buffer.enqueue_packet(packet_id)
```

Algorithm

the Early Random Drop algorithm works like this:

1. Imagine you have a box with a specific size, and you're putting packets in it.
2. If the box is not too full, you can put packets in without a problem.
3. But when the box gets quite full, you start to make decisions.
4. You roll a dice (random choice) to decide if you should drop a packet or not.
5. If the dice shows a certain number, you drop the new packet.
6. If not, you replace the oldest packet in the box with the new packet.
7. When you want to take out a packet, you get the oldest one from the box.
8. If the box is empty, you say it's empty.
9. You keep doing this for more packets.

So, it's like a game where you decide to keep a new packet, drop it, or replace an old one when the box gets crowded.

7) Implement RED algorithm

```
import random

class RED:
    def _init_(self, max_threshold, min_threshold, max_probability):
        self.max_threshold = max_threshold
        self.min_threshold = min_threshold
        self.max_probability = max_probability
        self.avg_queue_size = 0

    def mark_packet(self, queue_size):
        self.avg_queue_size = 0.1 * queue_size + 0.9 * self.avg_queue_size

    def should_drop(self):
        if self.avg_queue_size < self.min_threshold:
            return False
        elif self.avg_queue_size > self.max_threshold:
            return True
        else:
            drop_probability = (self.avg_queue_size - self.min_threshold) / \
                              (self.max_threshold - self.min_threshold)
            return random.uniform(0, 1) < min(self.max_probability, drop_probability)

# Example Usage:
red_instance = RED(max_threshold=10, min_threshold=5, max_probability=0.2)
current_queue_size = 8  # Replace this with your actual queue size

red_instance.mark_packet(current_queue_size)

if red_instance.should_drop():
    print("Drop packet")
else:
    print("Accept packet")
```

<u>Algorithm</u>

Imagine you're in charge of a line of people waiting for a bus:

1. *Few People in Line:*
   - If only a few people are in line, you let everyone get on the bus without any problem.

2. *Lots of People in Line:*
   - If the line is really long, you start telling people to wait for the next bus because the bus is too full.

3. *Somewhere in Between:*
   - If the line is not too short but not crazy long, you might occasionally ask someone to wait for the next bus, just to avoid the line getting too long.

4. *Decision Process:*
   - You make the decision based on how long the line has been on average, and if it's been consistently long, you might tell people to wait more often.

In technical terms, this is like the RED algorithm managing a network's traffic, preventing it from getting too crowded and slow.