Started on	Wednesday, 20 October 2021, 9:41 AM
State	Finished
Completed on	Wednesday, 20 October 2021, 4:24 PM
Time taken	6 hours 42 mins
Grade	**100.00** out of 100.00

Question **1**

Correct

Mark 40.00 out of 40.00

RDT 3.0, also known as **alternating-bit protocol**, uses ACK, timer and one-bit sequence number to achieve reliable data transfer.

When data are available from the above layer, the sender sends a packet (with data as payload) with a sequence number 0; it also starts a timer;

When it receives an ACK with a sequence number 0, it stops the timer and waits for another invocation from above;

When data become available again, the sender sends a packet with a sequence number 1 and starts a timer again;

When it receives an ACK with a sequence number 1, it stops the timer and waits for another invocation from above;

Whenever there is a timeout, the sender resends its last unAck'ed packet;

Whenever it receives a packet unexpected (e.g., a wrong ACK), it ignores it.

Here we ask you to implement a **simplified** version of the RDT 3.0 for its sender side. This sender side protocol can be described by the following FSM.

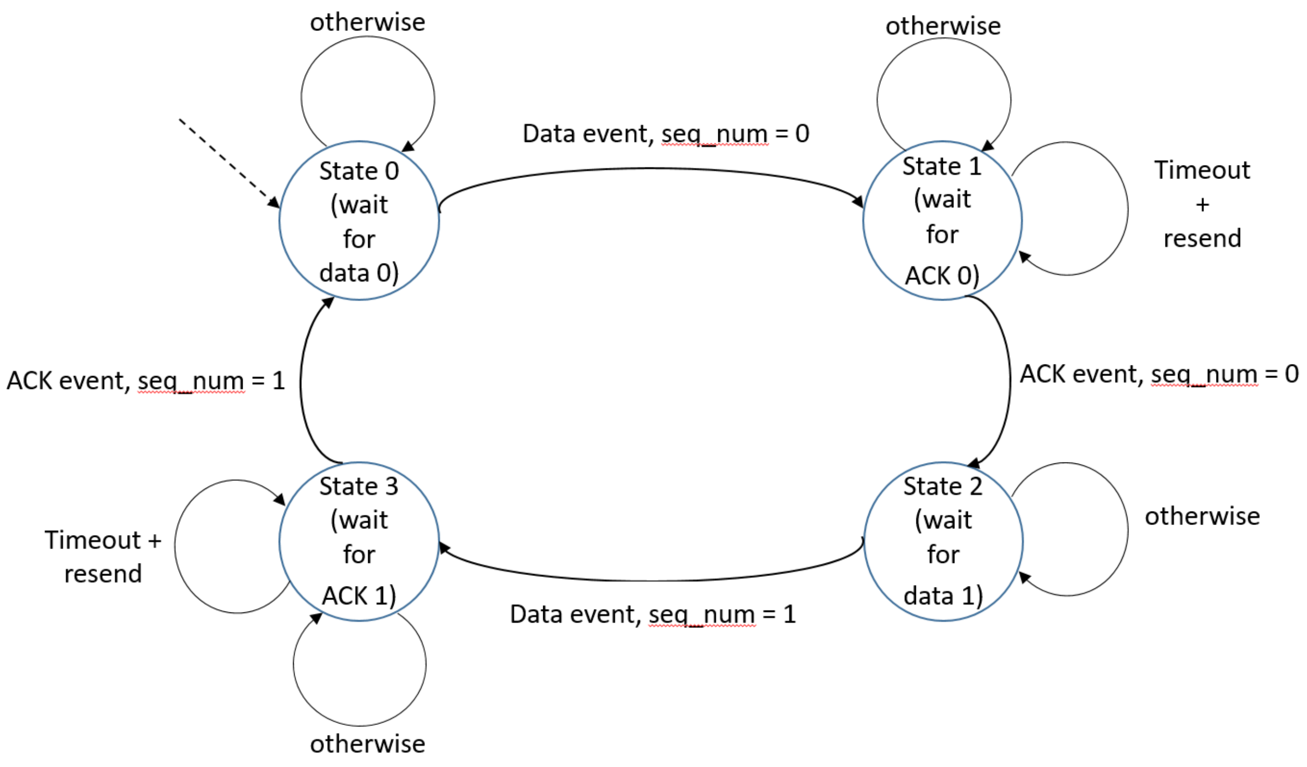
We define **four states** for the sender as follows (shown in the FSM):

0 - wait for data 0 (data with sequence number 0);

1 - wait for ACK 0 (ACK with sequence number 0);

2 - wait for data 1;

3 - wait for ACK 1;



The function RDT_sender(event, state) which you will complete has two input parameters: **event** and **state** (defined above). The **state** is the current state of the sender before processing the **event**.

event - simulated event in the following format (a list):

[type, seq_num, data]

Data event: An event with *type* **0** means the incoming of *data* with corresponding *seq_num*; for example, an event [0, 0, 3] means the arrival of data 3 with sequence number 0 at the sender; an event [0,1,4] means the arrival of data 4 with sequence number 1 at the sender. In RDT 3.0, the sequence number for a proper packet can only be 0 or 1.

ACK event: An event with *type* **1** means the arrival of an ACK which acknowledges the reception of *data* with *seq_num* at the receiver side. For example, an event [1,0,3] means the arrival of an ACK packet. This ACK packet indicate that the receiver has successfully received data 3 (its sequence number is 0). An event [1,1,4] means the arrival of another ACK packet. This ACK packet indicate that the receiver has successfully received data 4 (its sequence number is 1).

Timeout event: An event with *type* **2** means there is a timeout at the sender side. Note that instead of using a real timer, we use timeout event to simulate the behavior of a timer. For such an event, the sender should ignore the *seq_num* and *data* fields if they are given in the event. In some test cases, a timeout event is just a list [2] (without the *seq_num* and *data* fields).

The output of the function RDT_sender is in the following format (a tuple):

state, [status, seq_num]

state - the current state of the sender after processing the event. If the sender is at state 0, then after it finishes process a data event, e.g., [0,0,3], its state changes to 1 (waiting for ACK 0).

A list **[status, seq_num]** tells the result of how the sender processes the incoming event. *Status* **-1** means the sender receives unexpected packet (event); when this happens, the returned *seq_num* should be -1 as well. For example, if the sender receives an event [0, 2, 3] (an event with *seq_num* 2), it should output its current *state* and [-1, -1].

Status **0** means a data event (type 0 event) has been processed successfully, the *seq_num* is the sequence number of the processed packet. For example, when the sender is waiting for the incoming of data 0 (sender is in state 0) and it receives an event [0,0,1]; after processing this event, it should output "1, [0,0]" - now the sender is in state 1.

Status **1** means the sender has just processed an ACK event successfully. The *seq_num* is the sequence number in the ACK packet. For example, if the sender is waiting for ACK 1 (in state 3) and it receives an ACK event [1,1,4], the output should be "0, [1,1]" - now the sender is in state 0.

Status **2** means the sender has just processed a timeout event successfully (resending a packet); The *seq_num* is the sequence number in the resent packet. For example, if the sender is waiting for ACK 1 (in state 3) and it receives a timeout event [2,1,4], the output should be "3, [2,1]" - now the sender is still in state 3.

The input of our test function *sndr_test* is a list of events (*event_list*). The initial state of the sender is state 0. The initial sequence number of data is 0.

For example:

Test	Result
`sndr_test([[0, 0, 1], [2, 0, 1]])`	`[[0, 0], [2, 0]]`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
"""
Sender side simulation of RDT 3.0;

Input packets are formatted
[type, seq_num, message]
0 message with seq_num to be send;
1 ACK received; acking seq_num;
2 timeout event; resend last packet;

Output packets are formatted
[status, seq_num]
-1 unexpected packet, -1 as seq_num;
0 message sent successfully, seq_num is the seq # of the message;
1 ACK processed; seq_num is the ACk seq_num;
2 resending finished; seq_num is the seq_num of the resent message;

Four states as described in the FSM
0 - wait for data 0;
1 - wait for ack 0;
2 - wait for data 1;
3 - wait for ack 1;

"""

def RDT_sender(event,state):
    #Adding your own code below

    event_type = event[0]

    if len(event) > 1:
        seq_num = event[1]
        if len(event) > 2:
            data = event[2]

    has_error = False

    #Initial state; only accepts data 0; return error otherwise;
    if state == 0:
        #Incoming of data with corresponding seq_num
        if event_type == 0 and seq_num == 0:
            state = 1
            status = 0
        else:
            has_error = True

    #Wait for ACK 0
    elif state == 1:
        #ACK which acknowledges the reception of data with seq_num at the receiver side
        if event_type == 1 and seq_num == 0:
            state = 2
            status = 1
        #Timeout
        elif event_type == 2:
            status = 2
            seq_num = 0
        else:
            has_error = True

    #Wait for data 1
    elif state == 2:
        #Incoming of data with corresponding seq_num
        if event_type == 0 and seq_num == 1:
            state = 3
            status = 0
        else:
            has_error = True

    #Wait for ACK 1
    elif state == 3:
        #ACK which acknowledges the reception of data with seq_num at the receiver side
        if event_type == 1 and seq_num == 1:
            state = 0
            status = 1
        #Timeout
        elif event_type == 2:
            status = 2
            seq_num = 1
        else:
            has_error = True

    #Sender receives unexpected packet (event)
    if has_error:
        status = -1
        seq_num = -1

    return (state, [status, seq_num])


#Do not modify the following lines
def sndr_test(event_list):
    state = 0
    action_list = []

    for event in event_list:
        state, action = RDT_sender(event,state)
        action_list.append(action)
    print(f'{action_list}')
```

	Test	Expected	Got	
✓	sndr_test([[0, 0, 1], [1, 0, 1], [0, 1, 3], [2],[1,1,3]])	[[0, 0], [1, 0], [0, 1], [2, 1], [1, 1]]	[[0, 0], [1, 0], [0, 1], [2, 1], [1, 1]]	✓

	Test	Expected	Got	
✓	`sndr_test([[0, 0, 1], [2, 0, 1]])`	`[[0, 0], [2, 0]]`	`[[0, 0], [2, 0]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 0, 3], [0, 1, 4], [0, 0, 5]])`	`[[0, 0], [-1, -1], [-1, -1], [-1, -1], [-1, -1]]`	`[[0, 0], [-1, -1], [-1, -1], [-1, -1], [-1, -1]]`	✓
✓	`sndr_test([[0, 0, 1], [1, 0, 1], [0, 1, 3], [1, 1, 3]])`	`[[0, 0], [1, 0], [0, 1], [1, 1]]`	`[[0, 0], [1, 0], [0, 1], [1, 1]]`	✓
✓	`sndr_test([[0, 0, 1], [1, 0], [0, 1, 3], [1, 1]])`	`[[0, 0], [1, 0], [0, 1], [1, 1]]`	`[[0, 0], [1, 0], [0, 1], [1, 1]]`	✓
✓	`sndr_test([[0, 0, 1], [1, 1, 1], [0, 1, 3], [1, 1, 3]])`	`[[0, 0], [-1, -1], [-1, -1], [-1, -1]]`	`[[0, 0], [-1, -1], [-1, -1], [-1, -1]]`	✓
✓	`sndr_test([[0, 0, 1], [1, 0, 1], [0, 1, 3], [1, 1, 3],[0,1,4]])`	`[[0, 0], [1, 0], [0, 1], [1, 1], [-1, -1]]`	`[[0, 0], [1, 0], [0, 1], [1, 1], [-1, -1]]`	✓

Passed all tests! ✓

Correct

Marks for this submission: 40.00/40.00.

Question **2**

Correct

Mark 10.00 out of 10.00

RDT 3.0 for the receiver is much simpler. When it receives a packet, it checks its sequence number. If the sequence number is neither 0 nor 1, it outputs an error messages. Otherwise, it sends back an ACK message.

Please fill the following function **RDT_Receiver(packet)** which simulates the protocol for the receiver.

The packet is formatted as (a list):

[*seq_num*, *data*].

For example, [0,1] is a packet with sequence number 0 and its payload is 1. But [2,1] is not a valid packet.

The output of this function **RDT_Receiver(packet)** is a list:

[*status*, *seq_num*].

If the receiver receives a valid packet, it outputs a status code of 0 and the sequence number of the packet (simulating sending back an ACK); otherwise it outputs a status code of -1 and seq_num of -1.

For example, RDT_Receiver([0,1]) should return [0,0].

We call another function rcvr_test(packet_list) which calls the function RDT_Receiver().

The input of rcvr_test is a list of packets defined above.

You only need to complete the function RDT_Receiver().

For example:

Test	Result
`rcvr_test([[0, 1]])`	`[[0, 0]]`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
def RDT_Receiver(packet):
    #Your code here:

    seq_num, data = packet

    if seq_num != 0 and seq_num != 1:
        status = -1
        seq_num = -1

    else: #valid packet, sends back ACK
        status = 0

    return [status, seq_num]





#Do NOT modify the following lines
def rcvr_test(packet_list):
    action_list = []

    for packet in packet_list:
        action = RDT_Receiver(packet)
        action_list.append(action)

    print(f'{action_list}')
```

	Test	Expected	Got	
✓	`rcvr_test([[0, 1], [2, 1],[0, 2],[1, 3],[0, 4]])`	`[[0, 0], [-1, -1], [0, 0], [0, 1], [0, 0]]`	`[[0, 0], [-1, -1], [0, 0], [0, 1], [0, 0]]`	✓

	Test	Expected	Got	
✓	`rcvr_test([[0, 1]])`	`[[0, 0]]`	`[[0, 0]]`	✓
✓	`rcvr_test([[0, 1], [0, 1],[0, 2],[1, 3],[0, 4]])`	`[[0, 0], [0, 0], [0, 0], [0, 1], [0, 0]]`	`[[0, 0], [0, 0], [0, 0], [0, 1], [0, 0]]`	✓
✓	`rcvr_test([[0, 1], [1, 1],[0, 2],[1, 3],[0, 4]])`	`[[0, 0], [0, 1], [0, 0], [0, 1], [0, 0]]`	`[[0, 0], [0, 1], [0, 0], [0, 1], [0, 0]]`	✓

Passed all tests! ✓

Correct

Marks for this submission: 10.00/10.00.

Question **3**

Correct

Mark 40.00 out of 40.00

Go-Back-N (GBN) is a sliding window protocol for reliable data transfer at transport layer. The sender side protocol of GBN defines rules for three different events, namely, *invocation from above* (when there is a data call from the above layer), *receipt of ACK* and *timeout*.

We ask you to complete the function **GBN_sender**(*event*, *base*, *next_seq*) which simulates a simplified version of the sender side protocol of GBN. There are 3 input parameters. The parameter *event* simulates 3 types of events. The *base* and *next_seq* parameters are the current base of the sending window and the next available sequence number inside the window respectively. The function keeps track of these two variables. The window size N is fixed to 4.

The *event* parameter is given as a list [*type, seq_num, data*]. There are 3 types of events.

Data event: Type 0 (type = 0) event indicates the sender receivers a call from above and data is ready. For example one type 0 event [0,1,2] means there are data available to be sent at the sender side. For this type of event, you do not need to check *seq_num* and *data* fields in your function. Your function should only check whether the window is full; if not, you should update the variable next_seq (next available sequence number in the window) and return a list consisting of *status* code, the current *base* (of the window), and *next_seq*, i.e., [*status, base, next_seq*]. The definition of status code will be given later.

ACK event: Type 1 event means that the sender receives an ACK packet which acknowledges the reception of data at the receiver side. The data are associated with the *seq_num*. For example, a type 1 event [1,2,3] means the sender receives an ACK which acknowledge the reception of data 3 (its sequence number is 2) at the receiver side. Your function should check whether *seq_num* is in the range [*base*, *next_seq*-1]. If not, it returns a list consisting of a status code of -1, the current *base* and *next_seq*. Otherwise, it should update the variables base and next_seq and return a list [1, *base*, *next_seq*]. 1 is the status code. You should ignore the *data* field in your code.

Timeout event: Type 2 event is a timeout event. In practice, GBN resends all unAck'ed packets and restart a timer. To simplify this, your function should just return a list [2, *base*, *next_seq*]. 2 is the status code. There is no need for you to check *seq_num* and *data* in this case.

As we have mentioned above, your function should return a list [*status, base, next_seq*]. Status code has been defined below. *base* and *next_seq* are the current base of the sending window and the next available sequence number. Their initial values are both 0.

The status code is defined as follows.

Status **-1:** the sender receives unexpected event or the window is full;

status **0:** data have been sent successfully;

status **1:** ACK is received and processed;

status **2:** the sender receives a timeout event and retransmission is finished.

We call a function **sndr_test**(*event_list)* to test your function.
event_list is a list of consecutive events to simulate possible data arrival, ACK arrival or timeout events.
It then prints out a list of output produced by the function **GBN_sender**.

For example:

Test	Result
`sndr_test([[2,0,0]])`	`[[2, 0, 0]]`
`sndr_test([[0, 0, 1]])`	`[[0, 0, 1]]`
`sndr_test([[0, 0, 1], [0, 1, 2]])`	`[[0, 0, 1], [0, 0, 2]]`
`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3]]`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```
"""
Sender side simulation of GBN;

An event is formatted as [type, seq_num, data]
0 data to send; no check on seq_num and data;
1 ACK received; acking seq_num;
2 timeout event; resend all outgoing unAck'ed events; no check on seq_num and data;

Output of function GBN_sender() is formatted as
```

```
[status, base, next_seq]
-1 unexpected event/window full
0 data sent successfully
1 ACK processed;
2 resending finished;

N - the window size
base - seq# of lower winder boundary (base)

"""

N = 4 #window size;

def GBN_sender(event,base,next_seq):
    """
    1. event of form [type, seq_num, data]
    2. base is the current  base of the sending window to keep track of
    3. next_seq is the next available sequence number inside the window to keep track of
    """

    if event[0] == 0: #Data to send; check whether the window is full;
        #Check whether the window is full, if we are allowed to send packets*
        if next_seq >= base + N:
            status = -1
        else: #update the next_seq (next available sequence number in the window)
            next_seq += 1
            status = 0

    if event[0] == 1: #ACK to process;
        #check whether seq_num is in the range [base, next_seq-1]. TODO IS THIS INCLUDING??
        seq_num = event[1]
        if seq_num in range(base, next_seq):
            base = seq_num + 1
            status = 1
        else:
            status = -1

    if event[0] == 2: #Timeout
        #In practice, GBN resends all unAck'ed packets and restart a timer.
        status = 2

    return [status, base, next_seq]

#Do NOT modify the following code
def sndr_test(event_list):
    base = 0
    next_seq = 0
    action_list = []

    for event in event_list:
        action = GBN_sender(event,base,next_seq)
        base = action[1]
        next_seq = action[2]
        action_list.append(action)

    print(f'{action_list}')
```

	Test	Expected	Got	
✓	`sndr_test([[2,0,0]])`	`[[2, 0, 0]]`	`[[2, 0, 0]]`	✓
✓	`sndr_test([[0, 0, 1]])`	`[[0, 0, 1]]`	`[[0, 0, 1]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2]])`	`[[0, 0, 1], [0, 0, 2]]`	`[[0, 0, 1], [0, 0, 2]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3]]`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3], [0, 3, 4], [0, 4, 5]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [0, 0, 4], [-1, 0, 4]]`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [0, 0, 4], [-1, 0, 4]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 1], [1, 1, 3], [0, 1, 4]])`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [0, 2, 3]]`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [0, 2, 3]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 1], [1, 1, 3], [0, 1, 4]])`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [0, 2, 3]]`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [0, 2, 3]]`	✓

	Test	Expected	Got	
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3],[2]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [2, 0, 3]]`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [2, 0, 3]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3],[2,0,0]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [2, 0, 3]]`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [2, 0, 3]]`	✓
✓	`sndr_test([[1,0,0]])`	`[[-1, 0, 0]]`	`[[-1, 0, 0]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 1], [1, 1, 3],[1,1,3]])`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [-1, 2, 2]]`	`[[0, 0, 1], [0, 0, 2], [1, 2, 2], [-1, 2, 2]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 1], [1, 2, 3]])`	`[[0, 0, 1], [0, 0, 2], [-1, 0, 2]]`	`[[0, 0, 1], [0, 0, 2], [-1, 0, 2]]`	✓
✓	`sndr_test([[0, 0, 1], [0, 1, 2], [0, 2, 3], [0, 3, 4], [0, 4, 5], [1,2,3],[0,4,5],[0,5,6],[2,0,0], [1,4,5]])`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [0, 0, 4], [-1, 0, 4], [1, 3, 4], [0, 3, 5], [0, 3, 6], [2, 3, 6], [1, 5, 6]]`	`[[0, 0, 1], [0, 0, 2], [0, 0, 3], [0, 0, 4], [-1, 0, 4], [1, 3, 4], [0, 3, 5], [0, 3, 6], [2, 3, 6], [1, 5, 6]]`	✓

Passed all tests! ✓

Correct

Marks for this submission: 40.00/40.00.

Question **4**

Correct

Mark 10.00 out of 10.00

The receiver's actions in GBN are much simpler. If the packet the receiver receives is in order, the receiver sends an ACK for the packet, delivers the data to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. You are asked to implement a simplified version of the receiver side protocol in GBN.

The function you will complete is **GBN_Receiver**(*packet*, *exp_num*). Its first parameter is *packet* in the form of a list [*seq_num*, *data*]. To make it simple, you only need to check whether the sequence number is the expected one.

GBN_Receiver() should return a list [*status*, *exp_num*].

> If the sequence number of the received packet is equal to the expected sequence number, then *status* is assigned to 0 and the variable *exp_num* should be updated;
>
> otherwise it is set to -1 and *exp_num* is not changed.

In practice, the receiver should sent back an ACK in either case. Here we just want you to return a list [*status*, *exp_num*]. The initial value of *exp_num* is 1.

We use another function **rcvr_test**(*packet_list*) to test the function **GBN_Receiver().** The input of rcvr_test() is a list of *packets,* simulating the arrival of a few packets. The function **GBN_Receiver()** is then called on each of these *packets*.

You only need to complete the function **GBN_Receiver().**

For example:

Test	Result
`rcvr_test([[1,1]])`	`[[0, 2]]`
`rcvr_test([[1,1],[2,2]])`	`[[0, 2], [0, 3]]`
`rcvr_test([[1,1],[2,2],[3,3]])`	`[[0, 2], [0, 3], [0, 4]]`
`rcvr_test([[0,1]])`	`[[-1, 1]]`

Answer: (penalty regime: 0, 10, 20, ... %)

Reset answer

```python
"""
This is a program simulating the receiver side of GBN,
******************
Input packets are formatted as
[seq_num, data]

Output packets are formatted as
[status, exp_num]
0 - an ACK is sent;
-1 - unexpected packets received;
*******************
"""

def GBN_Receiver(packet,exp_num):
    """
    1. packet in form [seq_num, data]"""

    seq_num, data = packet
    #If the packet the receiver receives is in order,
    #the receiver sends an ACK for the packet, delivers the data to the upper layer.

    #In all other cases,
    #the receiver discards the packet and resends an ACK for the most recently received in-order packet.

    if seq_num == exp_num:
        status = 0
        exp_num += 1
    else:
        status = -1

    return [status, exp_num]

```

```
#Do NOT modify the following code
def rcvr_test(packet_list):
    action_list = []
    exp_num = 1

    for packet in packet_list:
        action = GBN_Receiver(packet,exp_num)
        exp_num = action[1]
        action_list.append(action)

    print(f'{action_list}')
```

	Test	Expected	Got	
✓	rcvr_test([[1,1]])	[[0, 2]]	[[0, 2]]	✓
✓	rcvr_test([[1,1],[2,2]])	[[0, 2], [0, 3]]	[[0, 2], [0, 3]]	✓
✓	rcvr_test([[1,1],[2,2],[3,3]])	[[0, 2], [0, 3], [0, 4]]	[[0, 2], [0, 3], [0, 4]]	✓
✓	rcvr_test([[0,1]])	[[-1, 1]]	[[-1, 1]]	✓
✓	rcvr_test([[1, 1], [2, 2],[3, 3],[4, 4],[5, 5]])	[[0, 2], [0, 3], [0, 4], [0, 5], [0, 6]]	[[0, 2], [0, 3], [0, 4], [0, 5], [0, 6]]	✓
✓	rcvr_test([[1, 1], [2, 2],[3, 3],[4, 4],[5, 5]])	[[0, 2], [0, 3], [0, 4], [0, 5], [0, 6]]	[[0, 2], [0, 3], [0, 4], [0, 5], [0, 6]]	✓
✓	rcvr_test([[1, 1], [2, 2],[4, 4],[5, 5]])	[[0, 2], [0, 3], [-1, 3], [-1, 3]]	[[0, 2], [0, 3], [-1, 3], [-1, 3]]	✓
✓	rcvr_test([[1, 1], [2, 2],[3, 3],[3, 3],[5, 5]])	[[0, 2], [0, 3], [0, 4], [-1, 4], [-1, 4]]	[[0, 2], [0, 3], [0, 4], [-1, 4], [-1, 4]]	✓
✓	rcvr_test([[1, 1], [2, 2],[4, 4],[3, 3],[5, 5]])	[[0, 2], [0, 3], [-1, 3], [0, 4], [-1, 4]]	[[0, 2], [0, 3], [-1, 3], [0, 4], [-1, 4]]	✓

Passed all tests! ✓

Correct

Marks for this submission: 10.00/10.00.

◄ Lab test 2020 (practice copy)

Jump to...

Superquiz (RDT protocols) (copy) ►