1. List Seven Layers of ISO/OSI Reference Model and Five Layers of Internet Protocol Stack and explain how Internet Protocol Stack model implements the function of ISO/OSI Reference Model.

The seven layers of the ISO/OSI Reference Model are:
1. Application
2. Presentation
3. Session
4. Transport
5. Network
6. Link
7. Physical

The five layers of the Internet Protocol Stack are:
1. Application
2. Transport
3. Network
4. Link
5. Physical

Before answering our second part, let's make some clarifying assumptions:

- The seven layers of the ISO/OSI Reference Model refer to Computer Networks in general, not for the Internet
- The Internet Protocol Stack has to do with the Internet

The internet is one type of Computer Network and as such is justified providing only a subset (that it needs to) of the services that ISO/OSI demands.
Assuming the ideas within the Transport Layer through to the Physical Layers are the same in both ISO/OSI model and Internet Protocol Stack, the difference will need to be made up in the application layer (in the Internet Protocol Stack).

That is, within the Internet Protocol Stack, 'Presentation' and 'Session' ideas from ISO/OSI will have to be implemented in the application stack as and when they are needed.

2. Explain the difference between Flow Control and Congestion Control

Congestion Control is responsible for being polite to the network as a whole
whereas Flow Control is the sender being polite to the receiver.

Reliable data delivery can be viewed as the transport of ALL bits from the sender application process to the receiver application process and the delivery of these bits IN ORDER.

On the receiver side of the TCP connection, there is a buffer that stores a stream of bits in the order they were sent for the consumption of the receiver's application process. Since buffer sizes are finite and owing to the fact that the application process may choose when to consume bits from the buffer, the buffer may fill up (remaining capacity dwindles).

If the receiver buffer is full, the receiver TCP side will have to drop incoming bits.
Because it is TCP's job to facilitate reliable data delivery, it cannot allow this.

Hence for outgoing acknowledgements from the receiver to the sender, it includes the capacity it has in its buffer (rwnd). The sender side TCP protocol uses the receiver's rwnd to understand how much space is left in the receiver buffer and the rate at which the receiver application process consumes bits.
From this understanding it matches, the sender side matches the rate at which it sends bits to the rate at which they are consumed by the receiver side.

This form of decreasing and increasing rate at which bits are sent is known as Flow Control.

Flow Control is the left arm of TCP and Congestion control is the right arm.
Since the TCP buffer situation is handled by Flow Control, then what does Congestion Control do?

The similarity that both have is they dictate fluctuations in the rate of bits sent by the sending side.

Recall that network congestion involves packet loss or increasing RTT times for acknowledgements of sent bytes.

And in congestion control, sending rate is governed by inferred increasing or decreasing network congestion.

TCP may infer network congestion by keeping a weighted running SRTT and from instances of packet loss (no ack arrives).

3. Explain why TCP connection needs two double handshakes instead of one double handshake.

With respect to the TCP protocol, we conduct a handshake(s) on the occurrence of one of two events:

1. A connection setup
2. A connection teardown

A connection 'setup' involves one 3-way handshake, whereas a connection 'teardown' involves two double handshakes.

Before proceeding to explain why a TCP connection needs two double handshakes instead of one double handshake, I make the clarifying assumption that my answer will be given in the context of a connection teardown.

When a TCP connection has been established between two end-systems A and B, both A and B allocate TCP variables and buffers. Let's call these variables and buffers and the port numbers reserved by A and B for the connection collectively as resources.

If A wishes to close the TCP connection with B, it participates in two double-handshakes.
The first part: 1_ A transmits FIN bit set segment to B AND 2) waits for B to send an acknowledgement suggesting it understands that A wishes to close the connection.
(*)
The second part: 1) B transmits FIN bit set segment to A AND 2) waits for A to send an acknowledgement suggesting it understands that B wishes to close the connection.

After both parts, A and B deallocate their resources from the connection.

The answer to the question of why two double handshakes rests in asking the following question:

Q. "Why do we need the second part, isn't 2. in the first part enough?"

- The answer rests on the premise that the first end system to send the FIN bit segment can no longer send data (not acknowledgements but payload messages).

But the other end system can finish sending the data (*) it needed to before sending it's FIN.
B can still send data (application messages) from A (only receives now).

This allows for a controlled way to close the connection - consider some application protocol that needs to take some steps on connection teardown.

Once B is done sending data, it initiates the second handshake.

All in all, the double handshake for TCP teardown allows B to finish sending the data it needs to to A before signalling that it also wishes to close the connection from its end.

4. Differentiate between Non-persistent HTTP and Persistent HTTP

First, HTTP is an application layer protocol that has to do with requests and responses. Second, a TCP connection requires one port number to be reserved on the Client and one on the server.

Accordingly, an HTTP request/response pair between a client and a server will require allocated resources on both sides.

Application developers may choose, depending on the requirements of their networking application, where they want port numbers (and other resources) allocated for a long duration of time or solely to service one request/response need.

If they find it fit to reserve the port numbers (and other resources) for a long duration, they may opt for Persistent HTTP.

If the application only needs to probe for some insignificant amount of information from a server, they may opt for non-persistent HTTP.

This choice is governed by the fact that allocating Transport layer resources is expensive computationally.

With persistent HTTP, if you make many requests and receive many responses, it helps to only allocate and deallocate TCP resources once.

For the other case, the tradeoff involves hoarding operating system resources like ports if the application is idle for large periods of time.

So we note here that the difference between persistent HTTP and non-persistent HTTP is that:

- in persistent HTTP, many request/response pairs can be handled through one TCP connection.
- in non-persistent HTTP, a request/response pair requires the creation of a new TCP connection.

5. Differentiate between circuit-switched network and packet-switched network

Let's assume we have a network of links and end systems A,B, and C on the network edge.

- The connection between A and B uses packet switching
- The connection between A and C uses circuit switching

When a packet with destination B propagates to a packet switch (router) between the path from A to B, it may suffer from queueing delays at that packet switch. i.e Packets in a connection that uses packet-switching wait their turn in queues and get transmitted when they are at the front of the queue.

In contrast, when a packet with destination C propagates to a packet switch (router) between the path from A to C, it will not wait in the same way that the packet with dest B did. In circuit-switching, a connection is established from A to C where bandwidth is reserved for the connection between A and C on every link on the path from A to C.

- Through time division multiplexing, a packet with src A and dest C may wait for its upcoming slot on the next frame (if this frame's slot for A-C consumed)

but this waiting is different from d_queue in that there is a reserved connection where A-C packet will always be allowed to pass regardless of the traffic.

- Through frequency division multiplexing, a packet is transmitted on the frequency reserved for A-C connection on each link on the path from A to C; it does not suffer from d_queue in the same way that an A-B packet might. Sure it may have to wait for a previous A-C packet to complete transmitting, but it doesn't wait on some arbitrary X-Y packet in a first come first serve manner.

^ With both TDM and FDM, the router with the bottleneck link may see a buildup of packets from A-C if previous routers' links had greater throughput.

But even with this buildup, the distinction is that packet n from A to C will wait solely on packet(0 ... n-1) from A to C and not for a packet from some X to some Y.

More generally, the links from A to C will allocate bandwidth solely for the A-C connection. The same is not the case with the A-B connection which does NOT use circuit switching.

There are two problems with circuit-switching however:

- elasticity (spikes in usage requirements for a dedicated connection)
- If the rate of packets sent from src A to dest C increases dramatically, a link will only use the reserved bandwidth to transmit A-C packets. This is the case even if the router which transmitted the A-C packet has no other traffic.
- silent times (idle times when dedicated bandwidth is not being used)
    - Suppose that A-C connection represents a call where Alice is put on hold by Chris and no packets are transmitted.
    - The bandwidth reserved on the links is going unused: other traffic cannot make use of the idle bandwidth since it is reserved.

One pro of circuit-switching is that since bandwidth is reserved for A-C connection, a throughput can be guaranteed. Namely the bottleneck link's transmission rate (reserved bandwidth) serves as a good approximation for the time it takes for sender A to get a packet to receiver C.

6. List Key Differences between Go-Back N and Selective Repeat

The similarity in both is that they are sliding window protocols.

The differences:

Go-Back-N protocol discards out-of-order packets. In a window where seq 5 - 10 packets (1 byte packets) are expected, if a packet with seq 8 comes before 6, seq 8 packet is discarded.

Selective repeat on the other hand will buffer the packet with seq 8.

Accordingly, there is more retransmission of bytes in Go-Back-N than there is in Selective-Repeat.

Go-Back-N works on the assumption of cumulative acknowledgements.

- Suppose packets with seq 5-15 are received in order (window size = 10).

The receiver sends an acknowledgement for each packet it received (5...15).

If the acknowledgements for packets 5 through 14 inclusive are lost, when the acknowledgement for packet 15 arrives, the server will know that the receiver got packets 5-14 as well.

Accordingly the sender/server can move its window forward by 10 units.

Selective Repeat protocol uses selective acknowledgements.

- For the same seq 5-15 packets that are received, Selective repeat will send one acknowledgement for each (just like Go Back N).

However, if acknowledgements for 5-14 are lost in the network and the acknowledgement for 15 arrives, the sender will retransmit 5-14 after timeout because Selective Repeat does not use cumulative acknowledgements.

7. How can one tell there is network congestion?

TCP maintains a running weighted average of RTTs.
Using alpha, beta and initial value, our computation of SRTT = initial value * alpha + newRTT measure (1 - alpha) gives us a dynamic and changing RTT after receiving acknowledgements.

beta * SRTT = timeout governs how long TCP waits till retransmitting due to timeout, based on the running SRTT.

When these SRTT values shift due to variations in acknowledgement receive times since time packet sent, TCP can get insight into network congestion.

If the SRTT is constantly increasing, TCP knows the network traffic is increasing.
If the SRTT is decreasing, TCP knows the network is becoming less congested.

But what if an acknowledgement is not received at all?

I go into detail of what TCP can observe when no acknowledge is received through the context of a packet of n bytes (we'll call it PACKY) to be transported from Alice to Bob.

Assume the network core is the packet switches on the route PACKY takes from Alice to Bob exclusive. Alice and Bob are endsystems on the network edge.

Recall that d_nodal = d_proc + d_queue + d_trans + d_prop

Among these, the only non-constant contributor to d_nodal is d_queue.

That is, d_proc, d_trans, and d_prop remain constant between every pair of packet switches in the network core.

In an ideal network with no other traffic in the network core, when Alice sends PACKY to Bob, the time it takes Bob to receive PACKY is exactly: Sum [d_nodal] for every node in the network core.

This sum, by definition of 'no other traffic' implies d_queue = 0 for every node in the network core.

Let's say in the ideal network, it takes PACKY 5 seconds to arrive at Bob since Alice transmitted PACKY.

In the event that it takes PACKY more than 5+x seconds where x>0 to arrive at Bob, there must exist at least one d_nodal which adds to the sum more than it did in the ideal network scenario.

- Because d_proc, d_trans, and d_prop remain constant for PACKY, it is certain this one node consists of d_queue > 0.

Hence when the time it takes for PACKY to get to Bob after being transmitted by Alice exceeds 5 seconds, PACKY had to wait in some queue.

We can tell if network traffic is increasing or decreasing by comparing against some baseline time to get a packet from Alice to Bob (say 5 seconds).

On these grounds it follows that:

if it takes PACKY an infinite amount of time to get from Alice to Bob (x = infinity), there exists some d_nodal that has a queueing delay of infinity.

Since queue sizes are finite, we can be certain that some node in the network core has dropped PACKY because it's queue was full.

Since the queue of some node in the network core did not have enough space for PACKY, we can conclude that it had to buffer other packets.

By definition, that node suffered from a traffic intensity approaching 1 (# incoming bits > # outgoing i.e transmitted bits) and so the queue filled up.

Given this traffic intensity, we conclude there was network congestion leading to PACKY being dropped.

In conclusion, we can tell if there is network congestion if there exists packet loss and SRTT increases over time (since the other factors which contribute to RTT are constant).

As a complement to all this, TCP from the sender side keeps track of the last byte sent vs the last byte acknowledged.

The congestion window, cwnd >= Last byte sent - last byte acknowledged. As the right hand side converges onto cwnd, we know that there is congestion since cwnd allowance is decreasing.

That is, as the disparity between last byte sent and last byte acknowledged grows, our ability to send more bytes is restricted since we wish to maintain the inequality.

8. How can a protocol keep the unknown network bottleneck link busy?

Let's answer this question first trivially via UDP and then via TCP since both transport on the network.

The implication of 'keeping network bottleneck link busy' means we wish to transmit at the maximum throughput at all times.
How? Note that throughput = min{R_1, ..., R_n}. For some x s.t. R_x is a minimum, we know that the throughput of the connection is the transmission rate of node x.

Assume that the receiver application process consumes bits as they arrive in the receiver TCP buffer; thus the receiver application process consumes exactly as fast as the sender sends.
This suggests that the maximum throughput (highest achievable constant rate of sending) between sender and receiver will be bounded by R_x.

The question asks how we can keep R_x busy, and therefore achieve maximum throughput by fully utilizing that link.

With regards to UDP, we know there is no form of congestion control. Hence UDP trivially keeps the network bottleneck link busy by sending packets as soon as they have been encapsulated with UDP header information. Therefore, it can output packets at a rate > R_x and keep n.b. link busy.

With regards to TCP:

- Given our assumption that receiver application buffer consumes bits instantly as they fill up, this example's sending rate will be solely dictated by congestional control (no flow control sender throttling).

The question now becomes, how TCP's congestion control can optimize network usage for full utilization of the network bottleneck link.

- The answer lies in shifting between the Slow Start, congestion avoidance, and fast recovery modes.

In the slow start mode, TCP doubles cwnd window size, thereby increasing sending rate, to find the slow start threshold. The slow start threshold is found if packet loss is detected. After which TCP resets cwnd to 1. Then as the doubling the approaches previous slow start threshold, TCP shifts to the congestion avoid mode to linearly increase the sending rate until a triple duplicate ACK is detected.

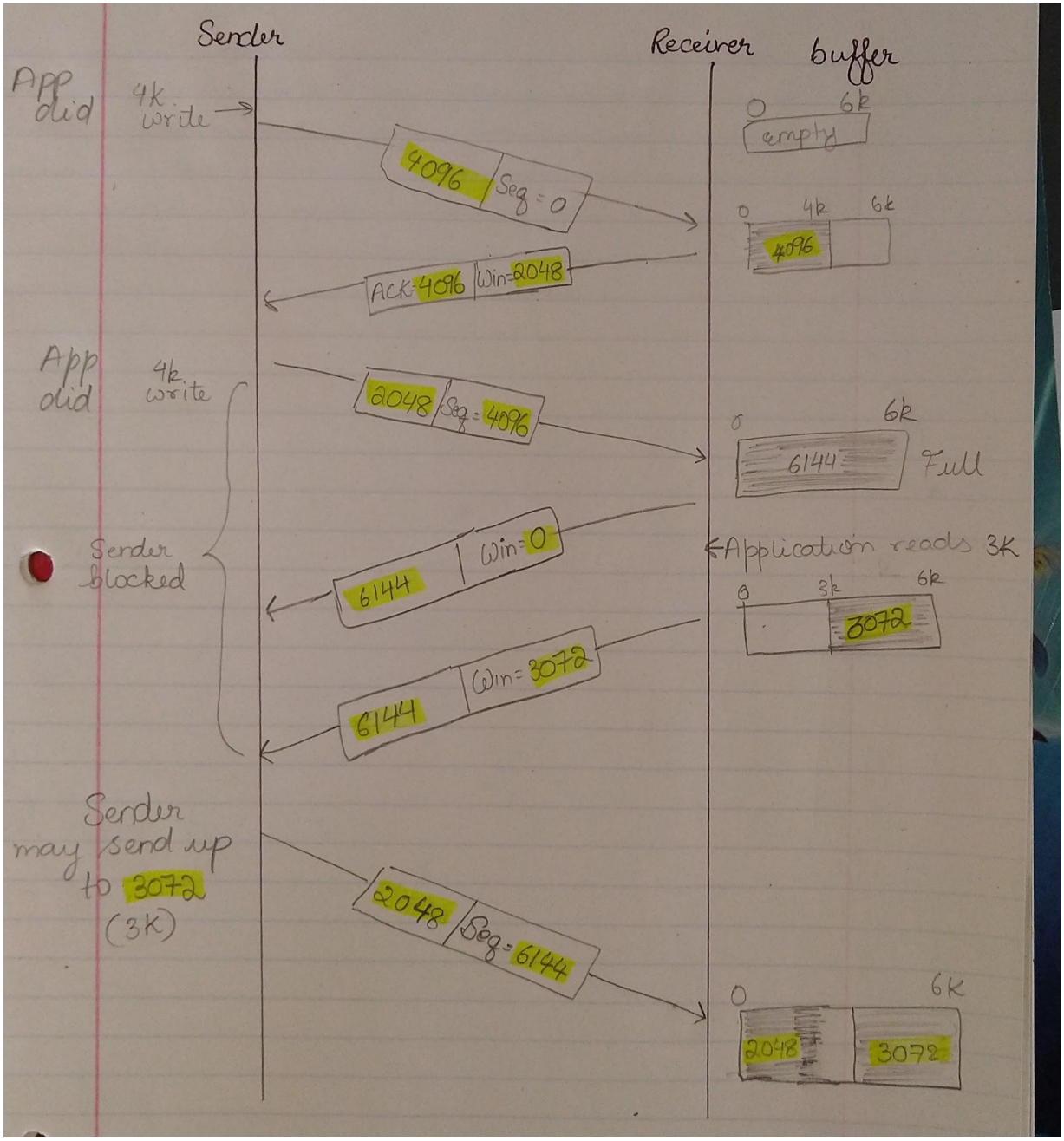
In trying to optimize sending rate through additive increase, Congestion avoidance mode, attempts to approach the rate at which packet loss may occur, that is, the rate at which it network congestion occurs; and when it does happen, TCP reduces cwnd to either 1 (Tahoe) or 3/4(cwnd @ time of packet loss) (Reno).

This mechanism describes an attempt to 'probe' for the optimal sending rate because TCP uses these modes to climb to the highest rate it can.

TCP uses the dynamically shifting ssthresh value to try and get the sending rate as close to the max send rate before hitting the roadblock of network congestion.

The search for an optimal sending rate implies the desire for maximum utilization of network bottleneck link. TCP is actively in this search and that is how it attempts to keep the unknown n.b. link busy.

9. 16 '?' fields.


Sender
Receiver
buffer
App did
4k write
4096
Seq = 0
0
6k
empty
0
4k
6k
4096
ACK=4096
Win=2048
App did
4k write
2048
Seq = 4096
0
6k
6144
Full
Sender blocked
Win=0
6144
←Application reads 3K
0
3k
6k
3072
Win=3072
6144
Sender may send up to 3072 (3K)
2048
Seq=6144
0
6K
2048
3072

10. Please calculate the SRTT and Timout Interval based on the Initial SRTT and parameters listed.

a = 0.5, (1-a) = 0.5; b = 3
all units are seconds:

RTT Measurement	SRTT	Timeout
2	= **2**	3*2 = **6**
3	= 2*0.5+ 3*0.5 = **2.5**	3*2.5 = **7.5**
4	= 2.5*0.5 + 4*0.5 = **3.25**	3*3.25 = **9.75**
1	= 3.25*0.5 + 1*0.5 = **2.13**	3*2.13 = **6.39**
5	= 2.12*0.5 + 5*0.5 = **3.57**	3*3.57 = **10.71**

11. Suppose you click on a link to obtain a Web page in your Web browser.

...

(i) we use Persistent HTTP connection without pipelining or

(ii) we use non-persistent HTTP with 5 parallel connections?

==========

* To get the IP address of the server hosting web page, IPsearch = RTT1 + ... + RTT6
* Let RTT0 be the round-trip-time from my web browser to the server hosting the web page and the 11 objects.

i. RTT1 + ... + RTT6 + RTT0 {establish TCP conn} + RTT0 {get HTML file} + 11 * RTT0 {get the 11 objects}

= RTT1 + ... + RTT6 + 2*RTT0 + 11*RTT0

ii. RTT1 + ... + RTT6

+ RTT0 + RTT0 {TCP conn + request/response for HTML file}

{with 5 non-persistent & parallel connections we can get set up 5 TCP connections every RTT0 and get 5 objects for each TCP connection }

+ RTT0 + RTT0 {this nets us 5 objects}

+ RTT0 + RTT0 {non persistent, so we had to make 5 more TCP connections in 1 RTT (parallel). Total object's web browser has now is 10}

+ RTT0 + RTT0 {1 more TCP connection + 1 more object needed}

==========

RTT1 + ... + RTT6 + 2RTT0 + 3*2RTT0

12. A TCP connection is established between two hosts A and B connected over 5 links in tandem. The bandwidth of the first link is 1 Mbps (bps=bits per sec, M = 10^6), and the bandwidth of the next 3 links is ½ of the previous link, and the bandwidth of the last link is ¼ of the first link. What is the maximum bandwidth of the connection?

Prompt inferred as:

Network: A ---------- + ---------- + ---------- + ---------- + ---------- B
Throughput: 1 Mbps 0.5 Mbps 0.25 Mbps 0.125 Mbps 0.25 Mbps

By assuming no other traffic in the network except between A and B, the maximum bandwidth of the connection is the throughput of the bottleneck link in the route for our connection.

Throughput of bottleneck link = min {1, 0.5, 0.25, 0.125, 0.25} * 10^6 bps.
Throughput of bottleneck link = 0.125 * 10^6 bps = 125,000 bps.

Maximum bandwidth of the connection is therefore 125,000 bits per sec.

13. Consider the GO back N protocol with a sender window size of 5 and a sequence number starting from 1. At some time t, the receiver sends an acknowledgment for 10 (received all packets up to 10). What are the possible sequence numbers of packets in the sender's window at time t?

Here, we are not asked for the number of possible distinct sender windows at time t, only for the range of possible sequence numbers that window can take on at time t.
Thus we do not concern ourselves with events regarding transmission of packets 1-5.

Note that since the window size is 5, we may give a range of possible sequence numbers in the window, but they may not all be in the same window.
In order to derive the possible sequence numbers of packets in the sender's window at time t, we need to look at cases where acknowledgements are received and lost.

Facts:

1. The window may slide over by x units if acknowldgements for the first x sequence numbers are obtained.

- Go-Back-N may obtain acks for the first x even if acks for 1..x-1 get lost because of cumulative acks.

2. From time = 0 to time = t:

a) The window size will always remain 5.

Propositions:

b) If window starts at 10, then max seq number in sender's window (that hasn't been sent) can go up to 14.

c) If the window ends at 10, then the min seq number in the sender's' window (that has been sent) can be 6.

3. The only way the receiver can send the acknowledgment for 10, is if seq packets 1 through 5 have been ack-d at the sender.

- We do not consider these packets here on out.

Claim:

From 2b we derive possible sequence numbers to be 10, 11, 12, 13, and 14.

From 2c we derive possible sequence numbers to be 6, 7, 8, 9, and 10.

2b Union 2c yields possible sequence numbers from 6 to 14 (both inclusive).

Therefore, sequence numbers 6,7,8,9,10,11,12,13, and 14 are candidates to be in the sender's window (though only a contiguous subset due to 2a).

Assume that 2b and 2c are sound propositions, then the claim is trivially valid.
Now we show that 2b and 2c are indeed sound:

2b.

- How can we achieve a case where the window starts at 10?
- Initially window is [1,5]. Suppose all packets are sent in order and ack for 5 is received.
- Window is now [6,10].
    - There exists a case where window starts at 6, and packets 6 through 10 (inclusive) are transmitted.
    - packets with seq 6,7,8,9 reach at the receiver in order. That is, packet with seq 10 is lost in network core.
    - When the ack for packet 9 reaches the receiver, the window slides 4 places forward from the starting value of 6, since 6,7,8,9 were ack-d.
    - now sender window is 10-14 inclusive.

- Due to timeout, packet 10 will be resent, along with packets with seq 11,12,13, and 14.
- Now when packet with seq 10 arrives at the receiver, it can send an ack with seq = 10.
- Therefore at this time t, sender window is waiting for acknowledgements for 10,11,12,13,14.
- So we have a max value of 14.

- There exists another case where window starts at 6 and packets 6 through 10 are transmitted and all received at sender.
- But in this case, the ack for packet 10 not received (but ack with seq-9 received).
- Therefore, window moves forward by 4 to have a new start at 10 and end at 14.

Now we show why window can start at 6 and show soundness of 2c.

2c:

- In order for the sender to have sent packet with seq = 10, it must have first sent packets with seq 6 through to 9 (inclusive).
- In order for the receiver to send an ack of seq=10 in Go-Back-N, it neccessarily must have received packets 6 through 9 {cumulative acknowldgements}.
- It IS possible, that cumulative acks 6,7,8, and 9 are LOST in network core.
- Thus sender is not guaranteed to have delivered packets 6-9. Accordingly, it does not move its window forward from base seq of 6.
    - In the event that the base wasn't 6 to begin with, we can assume that it received an ack of 5 but not for anything after and including 6.
- The existence of packet with seq 10 sent but packet with seq 6-9 ack not received at sender implies that the window is [6,10].
- Therefore, given this ack packet loss, we have the sender's window starting at 6.

14.

Let us that each packet is 1 segment.
We need the RTTs required to send 25 packets. And initial ss-thresh = 40.

We need 9, since linear increase after RTT 5.