This program is meant to open/read a pcap file and analyze its contents for the first three flows, each containing transaction information, total time to run, number of bytes, throughput, estimated congestion window sizes, and retransmission statistics. The first thing the program will do is read the argument in the user input (the pcap file) and attempt to open/read the pcap file. If unsuccessful, the program will print “Invalid File” and will proceed to exit. Otherwise, the getFlows() function will be called taking in the pcap file that was read.

For part a, in the getFlows() function, it starts off by creating a list for all of the flows, then loops through each buffer to organize the information into flows to add to the list. Each is organized with a timestamp, a list containing all packets, the starting sequence number, and a scale used to shift bytes to calculate the receive window size. The timestamp indicates the start of the flow and will be used to calculate the total time and throughput. Once all flows are organized in the list, the program goes into the printContents() function, which prints all information for each flow until the maximum number of flows is analyzed. The function kept track of whether or not the triple handshake was completed because the start of the first two transactions began immediately after that. So once it was complete, the transactions would start being printed. The number of bytes were calculated at the start of each transaction (coming from the source ip only) by taking the length of the tcp data plus the offset, then adding it to the total bytes variable for the flow. Once the fin packet was encountered, the timestamp was taken from the last packet, then both the start and end times were converted to a number so they can be subtracted from each other. The total time was based on the difference of those numbers and the total number of bytes was the final sum of the lengths and offsets from each packet. The throughput was then calculated by the number of bytes divided by the total time taken in units bytes/second.

For part b, the congestion window sizes were estimated based on AIMD and the formula, newRTT = (1-a)*oldRTT + a*(newSample) where “a” signifies the weight (the recommended weight is 0.125), oldRTT is the previous RTT calculated, and newSample is the current RTT. For AIMD, the congestion window size will start off small right after the tcp connection is established. It grows exponentially (basically doubling the previous window each time based on the results you will see) and will eventually hit the slow start phase when reaching the ssthreshold. Once it reaches the threshold, the size increase will linearize, but only having three congestion windows printed made it not reach that threshold. For the formula, the program kept track of new and old RTT times (indicated by the newRTT and oldRTT variables respectively) so that the formula would function. Every transaction, the oldRTT time would become the current newRTT time, then the newSample time would be the current transaction time by subtracting the start time from the end time of the transaction. Once the newRTT time was calculated from that formula, it was then checked to see if two times that value (2 * RTT’ = RTO) was close enough to the current transaction timestamp minus the start of the flow. If it is within 0.001, since both values would almost never be the exact same and the number of congestion windows is less than the total number of congestion windows allowed (three), then it would update the congestion window size. The current congestion window would take the previous size, if the current index is greater than zero, and would add the number of packets sent from the source during the current transaction. After each transaction finished, each estimated window size was stored in a list that was printed at the end of the flow. For retransmission statistics, triple duplicates specifically, were found by checking if the current sequence number of an ack packet coming from the sender is found in the current state of the ack dictionary. The ack dictionary grows every time there is a new ack packet coming from the receiver and it stores that ack number. So, if there is a sequence number of a packet from a sender that encounters an ack number in the dictionary, then it must be a duplicate. A dictionary is used instead of a list because there needs to be a count for how many times the packet was encountered since it needs to be hit three times for it to be considered a triple duplicate ack. The retransmits variable in the flowDict dictionary would be updated to be the current ack number at the end of the transaction that found a duplicate packet where the sender is the destination ip address. For timeouts, it was checked if the actual RTT for the current transaction was greater than the estimated RTT since that would produce a timeout as it went over the expected time limit. The way this worked was by adding the timestamps of ack packets to a dictionary from the source ip address. Every time an ack packet is sent from the receiver, the ack packet key and the timestamp of that ack packet are stored in the timeDict dictionary. So, because the ack packet coming from the receiver is the sender's packet data length plus the sequence number, the program can check if the current sequence number was already used prior. If it was, then that could mean there was a timeout and the sender is trying to give the receiver another packet. If that conditional passed the check, then the program would check if the current transaction minus the packet timestamp (which is one round trip) is greater than the estimated RTT. This would indicate a timeout since the packet was not received within the specified time. Any timeout or duplicate added one to the respective variables. All data was printed at the end of each flow