CS425A:Computer Networks, Assignment 1
2020-21-II

Some template codes are provided. Do NOT use any libraries beyond what is given in the template. Marks may be deducted based on the functions of the additional libraries used.

Problem 1:

In this Question, you are required to read an input image pixels row-wise, e.g., if there are 9 pixels they should be read in the following order.

1	2	3
4	5	6
7	8	9

For each retrieved pixel you need to perform the following tasks.

1. You need to separate them into Red, Green, and Blue components represented as an integer in the range 0 to 255. Follow strictly the order Red followed by Green followed by Blue.

P1R,P1G,P1B	P2R,P2G,P2B	...
...	...	...
...	...	...

   You need to write a function that converts each integer value into its equivalent 11-bit binary string by appending zeros at the left, i.e., the above image will be represented as 000<8 bit bin of P1R>000<8 bit bin of P1G> …
   Continue doing the same for other RGB values in the same order.
2. You need to write a function named **hamming_encode(bitstring)** to encode each 11-bit binary string into 15-bit bitstring using **(15,11) hamming code.**
3. Each (R, G, and B) 15-bit encoded bitstrings are flipped at zero or more places using the same 15-bit flip_bits array in the code template. You need to add this functionality into your code where the position of '1' bit in flip_bits array indicates that the corresponding bit is to be flipped in the 15-bit encoded bitstring. For the next pixel, this flip_bits array gets changed using LeftShiftRotate functionality in the code template. You need to write a function named **hamming_decode(bitstring)** to decode each bitstring in the same order they were encoded. In the hamming_decode function, your task is to check whether the 15-bit bitstring has an error, if so print the error bit, alongside the guessed 11-bit string. if there is no error, print 'Valid', followed by the decoded 11-bit string.

Processed 11-bit string: [d3, d5, d6, d7, d9, d10, d11, d12, d13, d14, d15]

Encoded 15-bit string: [p1, p2, d3, p4, d5, d6, d7, p8, d9, d10, d11, d12, d13, d14, d15]

Input-Output format: Your code will be tested against multiple images and against different values of flip_bits. Take this input as a command-line argument. For tasks (1) and (2), you should print the output in file `sender.txt` that contains integer values and encoded 15-bit bitstring corresponding to R, G, and B components as shown below. For task (3), you'll print the output in file `receiver.txt` that contains error bit position or valid if encoded bitstring is correctly received, along with 11-bit guessed bitstring separated by a space.

Input: as a command-line argument takes image name = example1.jpg, and starting flip_bits = 1,0,0,0,0,0,0,1,0,0,0,0,0,0,1

Output:
sender.txt
{5,3,0} {010000000000101,100000000000011,000000000000000}
{4,1,0} {100100010000100,110100010000001,000000000000000}
{5,1,2} {010000000000101,110100010000001,010100010000010}
…..so on

receiver.txt
d6 00100000100 d6 00100000010 d6 00100000001
d6 00110000111 d6 00110000010 d6 00110000011
d5 01100000011 d5 01100000111 d5 01100000100
….
Valid 00001000011 Valid 00001000010 Valid 00001000000
Valid 00010000001 Valid 00010000001 Valid 00010000101
….so on.

Problem 2
For this question, you need to implement **Stop and Wait ARQ protocol.**
The question is divided into two parts, **sender** and **receiver,** both are separate subquestions.

1. **Receiver** -

   Polynomial - $x^4 + x + 1$

message(any no. of bits)	crc_checkbits(4bits)	flag(1 bit)

   The input is given in the form of a string of 0's and 1's.
   Now you have to check whether the frame is **valid, corrupt, or duplicate** (same as previous frame received) , if the frame is **valid** then read the message in

chunks of 8 bits and convert the chunk to corresponding character (ascii code) and print the converted message in console, else if the frame is **duplicate** we will print “duplicate” on console (without quotes), else if the frame is **corrupt** we print “corrupt” on console (without quotes).

You can assume the flag bit used previously was ‘0’ therefore the first message which we get in the input file ideally should have the flag bit as ‘1’.

The input file consists of a sequence of bit strings in order separated by a newline**.**
“010101010101” - means that the receiver received a frame with data “010101010101” where the starting bits are the actual message itself which is followed by 4 bits of crc_checkbits followed by 1 bit of flag (note we use flag values 0 and 1 only and alternate between them to implement the receiver).

You have to take input from file “`input.txt`” and provide the output on the console.

Sample `input.txt`:

```
01101000011001010111100101101
0110100101010
0110100101010
011000010110110110101
011100110110000101101101011100000110110001100101100000
011101000110010101110011011101000010 1
011101000110010101110011011101000000 0
```

Sample output (on the console):
hey i duplicate am sample test corrupt

NOTE-

1. Do not use any library for checking CRC.

2. Sender-

Polynomial - $x^4 + x + 1$
Timeout = 2 seconds
Your job is to simulate working of stop and wait ARQ’s sender’s function.
A template is provided with two functions running on different threads concurrently, functions are sender() and receiver(), these two functions are connected by a socket for inter-communication, you need to **write** the sender function which takes input from the file “`input.txt`” line by line and sends all

the messages present in the file to receiver, you need to handle all cases like ACK getting lost, message getting lost, etc.
Sender must send the message with the 4 bit crc_checkbits and 1 bit flag(0 or 1) as explained in 1st part of the question,
The ACK received is a string that is “1000” + crc_checkbits =”`10001011”` for a message with flag ‘0’ and “1001” + crc_checkbits = “`10011000”` for a message with flag ‘1’.
The receiver() function is already written and should be considered as the sample test case.

To send data from one function to other we use the following -

```
s.sendall(bytes(ack, 'utf-8'))
```

bytes() function converts the string into byte format to be transferred.
To receive data we use the following -

```
msg = s.recv(1024)
msg = msg.decode('utf-8') #convert from byte format to string
```

Where argument 1024 means we will take no more than 1024 bytes, all the test cases will transfer less than 1024 bytes.

Explanation of sample input i.e receiver() function is provided in the template comments.

Output for sample Receiver function -
message sent
wrong ack
message sent
no ack
message sent
message sent
message sent

Output format : whenever you try to send a message and receive its ACK correctly you should print “message sent” in console.
Whenever we don’t receive an ACK we should print “no ack” and whenever the ACK is corrupt or wrong we should print “wrong ack”. You should print a new line after every print statement.

NOTE-

1. sockets used are blocking in nature, i.e., instructions are executed one after another -- only when the current instruction is complete the next instruction will start executing.
2. The python code is readable, you can replicate it in C (template for C is not provided).
3. It is assumed that either the ACK comes before timeout or else it is lost.