CS425
Programming Assignment No. 1

Due on February 14

General Instructions:

- You can write a program in C, C++, Java or python. (It must take input file name from command line argument)
- You must provide readme file which contains the language you used, command for compile if required, run command and the version of language you used like Python 2.7 or 3.0.

1. CRC checksum

For this program, you will be given a sample input file containing N number of frames in the following format that contains only 0/1's. The framing is done using a byte-oriented framing scheme with flag bytes and byte stuffing. Each frame starts and ends with a FLAG. There is an escape byte ESC which is used to insert "accidental" FLAG byte and ESC byte in the data. If we need to transmit data which has FLAG or ESC byte respectively, then we send it by using ESC followed by (0x20 XOR FLAG) or (0x20 XOR ESC) so that FLAG will never occur in the DATA part. The format of the frame is as follows. Minimum size of data will be 1B (byte).
FLAG = 10101001(®)
ESC = 10100101(¥)
Generator Polynomial = x^7+x+1

FLAG (1B)	DATA	CRC_CHECKBITS (1B)	FLAG (1B)

You will be given two sample input files and corresponding output files of that input. You have to take this filename as a command-line argument of the program and provide the output on the console.
Sample input files contain N (the actual value may differ for different test cases) number of frames containing only 0/1's in a single line without any space (you can read it as a chunk of 8 bits).
Your job is to find the total number of frames which are separated by FLAG, check invalid frames using CRC_CHECKBITS with the given generator polynomial and print the invalid frame numbers (in the order it appears in the input file starting from 1) separated by '**,**' (if no invalid frame is found leave it blank). If the frame is valid read it and print the character of that ASCII value in a single line.
NOTE: for generating 1 byte of the CRC_CHECKBITS from the generator polynomial, pad one zero at the end. While decoding, ignore the last bit and consider the rest as the CRC checkbits.

Sample output:
12
6,8,10
Cn Assignment

Explanation:
Total number of frames
Invalid frame no separated by **,** (frame no starts with 1)
Valid data in string format.

Grading Policy

You should not use any external libraries to solve this problem.
If program correctly handles test cases of sample input file (20 points)
If program correctly handles test cases of hidden input file (30 points)

2. Hamming Code

For this problem, you need to decode given binary string(s) to their corresponding characters' paragraph. That paragraph may or may not be correct, which depends on whether you found any incorrect characters in the paragraph. The encoding process employed is as follows :

Given paragraph $\Longrightarrow$ encode each character into 8 bits $\Longrightarrow$ Take each nibble (in order) and encode using Hamming(7,4) $\Longrightarrow$ Append all the 7bit encoded blocks in the same order as they occurred in the original paragraph.

You need to do the reverse of the above process and decode the bit string into actual para.

While decoding the string, you may come across these possible situations:

1st Nibble	**2nd Nibble**	**Char to be used**
No error	No error	correct char
No error	Recoverable	Possibly correct char
Irrecoverable	-----any-------	@
Recoverable	Recoverable	Possibly correct char
Note: Both the nibbles can be used interchangeably in the above table. It is just to give an idea that needs to be followed while decoding.		

There are several key points that you need to keep in mind:

- Char = 8 bits broken into 7+7 bits with hamming encoding of its constituent 4+4 bits
- The paragraph may contain 255 characters except @ (it was not there in the original text before encoding).
- We have introduced only 1 or 2 bit errors so as to enable you to detect all the errors present in the bit string. Also, as you might be aware that it is not possible to distinguish 1 bit and 2 bit errors as they both look the same. We can correct 1 bit errors in Hamming(7,4) successfully but not the 2 bit ones.
- "Recoverable" here means the error which vanishes after doing the suggested correction by the bit position whereas "Irrecoverable" refers to the error which persists even after performing the suggested bit manipulation in the word.
- You can easily keep track of the current nibble (if it is the first one of the current char or the second one) in programming.
- We have mentioned "*possibly correct char*" because it is impossible to know if the error we just corrected was 1 bit or 2 bit (If it was a 2 bit error, we were fortunate enough to get a char by following the bit manipulation suggestion fulfilling the hamming properties)
- The given paragraph contains *bits in multiples of 7* ! If not, output **"INVALID"**, followed by two newlines and the decoding of the following paragraphs (as usual).

INPUT-OUTPUT Format:
Your code will be tested against multiple test cases present in an input file named "input.txt" wherein there will be multiple bit strings (representing a paragraph) with a carriage return followed a blank line (so finally two carriage returns/newline) in between two paragraphs.
So for input, hardcode that name ("input.txt") in the code itself.

You need to print the output on the console in the following format:

```
This @s a sa@ple output wi@h di@fer@nt char@@ters s@ch as : ,./ blah blah u23 hello *&)$_@${}}\@/f, note that this text has @ also, since this is the decoded para and @ here signifies the irrecoverable char in the para. Original might have had some character we don't know about!!



This is the second decoded para after two newline/carriage returns. This is perfectly fine without any ampersand character since there wasn't even a single irrecoverable error in the whole para.



So on… (this is actually the 3rd paragraph :| )
```

Example 1:
Original text:
IITK, 1959

Encoded Text (sent) :
100**1**10000**11**001100110000**1**1001010010**1**10**0**11001001100011001101010100111100**010**
10100000000100001111010011**1**0000110011001100001101001 01100001100110**01**

Received Text (**Sample Input 1**):
100**0**10000**00**001100110000**0**1001010010**0**10**1**11001001100011001101010100111100**111**
1010000000010000111101001**0**0000110011001100001101001 01100001100110**10**
Sample output 1:
@ITK,�195:

Explanation:
Take for instance the first 14 bits 100**0**100 00**00**001. As we can see in the second received block, there is a 2 bit error which implies that either it can be recoverable or irrecoverable.
For a moment, after decoding 0000001, it seems that the nibble sent must have been 0000. But if you encode 0000 again, you'll find its corresponding hamming code to be 0000000 (different from 0000001). Therefore this error was "Irrecoverable". ***(Note that by "Irrecoverable", I do not mean 2 bit errors. All 1 bit errors can be detected and corrected while for 2 bit errors, sometimes they may slip under our nose and may pose as if they were 1 bit error. This happens due to the incompetence of Hamming(7,4) to correct 2 bit errors. In cases where they don't have an equivalent 1 bit error, we're referring to them as "irrecoverable" otherwise "recoverable")*** and we had to replace the character 'I' (from **I**IT) with '@'.
As far '�' is concerned, you may refer https://codescracker.com/python/program/python-program-print-ascii-values.htm (specially snapshot 4th onward on the webpage) for more info. This is shortcoming of python. The character there had ASCII value 160! You do not need to worry about the symbol shown on your machine, since your code and our solution code will be run on the same machine and produce the same displayed character.

Example 2:
Original text:
#/$.+-Az425c? Is the, bEzT pswd! :)

Encoded Text (sent) :
010101010000110**1**0101011**1**1111**0**101010100110**0**0101010001011**1**0010101**0**0110011010
10101010101100110**0**11010010001111110110**10**10000111001100100001101010**1**0100001
10100101110011010000111000011111111101010100000000010011000011001000111110
00011010101000000000000011111001100110011**0**11100001100110010010101010101001111
0001010100000000011001100101010100110001001**0**10001**1**111011010010010110011000
10101000000000000011110**0**00000000111110000110001111 0**0**01111110011010011000101
010110100101010100000000**00**100001110110100101010**0**0011001

Received Text (**Sample Input 2**):
010101010000110**0**0101011**0**1111**1**101010100110**1**010101010001010**0**0010101**1**0110011010
1010101010110011 0**1**1101001000111110110**01**10000111001100100001101010**0**0100001
1010010111001101000011100001111111110101010000000010011000011001000111110
0001101010100000000000011111001100110011**1**1110000110011001001010101010 01111
0001010100000000011001100101010100110001001**1**10001**0**111011010010010110011000
1010100000000000011110**1**00000000111110000110001110**1**011111110011010011000101
0101101001010101000000**11**10000111011010010101**1**0011001

Sample output 2:
#/$.+-Ay425c? Is the, bEzT pswd!#:)

Explanation:
In this example, there are mostly one bit errors only. So, almost all have been recovered. Only two characters are decoded wrong (‘z’ as ‘y’ and ‘ ’ as ‘#’). Absence of ‘@’ shows that all errors were recoverable.
‘z’ has ASCII value 122 and encoded as 0111 1010 in binary which in turn is encoded as 0001111 1011010 in Hamming(7,4) code but was received as 0001111 1011001. Moreover, ‘y’ is encoded as 0001111 0011001 in Hamming(7,4) code, which has a hamming distance of 1 with z’s corrupted code that we received, that is why it failed and misinterpreted ‘z’ as ‘y’.
The same reasoning can be given for second changed character as well.

Grading Policy
If the program correctly handles the two given test cases (mentioned in this file) (20 points)
If the program correctly handles test cases of the hidden input file (30 points)

Additional instructions
- All programs submitted will be checked by a plagiarism detection software. Honor code policy will be strictly enforced. Write the code by yourself and protect your code.
- Some sample testing files will be provided to you for your convenience. But thorough testing will be done while grading the assignment, so you should test it accordingly before the final submission and not just for the files given.
- For coding in Python, no special libraries required, just the conversions among different data types, etc.