1. Unreliable transmission

Input(bit string and probability) then for loop testing for 100 trial output as follow

```
count = 0
for i in range(0,100):

    flag = unreliable_transmission('10001', 0.05)
    if(flag == 1):
        count+=1
        #print('corrupted frame')
    else:
        #print('uncorrupted frame')
        pass

print(str(count) + '%')
```

```
25%
```

```
count = 0
for i in range(0,100):

    flag = unreliable_transmission('10001', 0.1)
    if(flag == 1):
        count+=1
        #print('corrupted frame')
    else:
        #print('uncorrupted frame')
        pass

print(str(count) + '%')
```

```
30%
```

```
count = 0
for i in range(0,100):

    flag = unreliable_transmission('10001', 0.2)
    if(flag == 1):
        count+=1
        #print('corrupted frame')
    else:
        #print('uncorrupted frame')
        pass

print(str(count) + '%')
```

```
61%
```

2. **Parity Bit**

Generator: Input the word and then the function will convert to a binary string automatically. The maximum size of bit string is corresponding to the biggest size of the bit string of the character ex. ‘1001 10’ will convert to ‘1001 0010’.

```
The word is: g]9)
parity type: one-dimensional-even
11001111 10111011 01110010 01010011
********************************************************************************
The word is: g]9)
parity type: one-dimensional-odd
11001110 10111010 01110011 01010010
********************************************************************************
The word is: g]9)
parity type: two-dimensional-even
11001111 10111011 01110010 01010011 01010101
********************************************************************************
The word is: g]9)
parity type: two-dimensional-odd
11001110 10111010 01110011 01010010 10101011
********************************************************************************
The word is: Hello World
parity type: one-dimensional-even
10010000 11001010 11011000 11011000 11011110 01000001 10101111 11011110 11100100 11011000 11001001
********************************************************************************
The word is: Hello world
parity type: one-dimensional-odd
10010001 11001011 11011001 11011001 11011111 01000000 11101111 11011111 11100101 11011001 11001000
********************************************************************************
The word is: Hello wolrd
parity type: two-dimensional-even
10010000 11001010 11011000 11011000 11011110 01000001 11101110 11011110 11011000 11100100 11001001 00000000
********************************************************************************
The word is: Hello world
parity type: two-dimensional-odd
10010001 11001011 11011001 11011001 11011111 01000000 11101111 11011111 11100101 11011001 11001000 11111110
********************************************************************************
The word is: MUICT NO1 THAILAND
parity type: two-dimensional-even
10011010 10101010 10010011 10000111 10101001 01000001 10011100 10011111 01100011 01000001 10101001 10010000 10000010 10010
011 10011001 10000010 10011100 10001000 11001010
********************************************************************************
The word is: MUICT NO1 THAILAND
parity type: two-dimensional-odd
10011011 10101011 10010010 10000110 10101000 01000000 10011101 10011110 01100010 01000000 10101000 10010001 10000011 10010
010 10011000 10000011 10011101 10001001 00110100
********************************************************************************
```

Checker: Input the bit string of any size the program will automatically increase each bit string ex. If input is ‘101 11 1101’ the programming will convert to ‘0101 0011 1101’ then proceed to the checking method. The test case for PASS that I use is the bit string from the generator so that means our generator and checker are correct.

```
parity_type: one-dimensional-odd
11001110 10111010 01110011 01010010
PASS
********************************************************************************
parity_type: one-dimensional-even
11001111 10111011 01110010 01010011
PASS
********************************************************************************
parity_type: two-dimensional-even
11001111 10111011 01110010 01010011 01010101
PASS
********************************************************************************
parity_type: two-dimensional-odd
11001110 10111010 01110011 01010010 10101011
PASS
********************************************************************************
parity_type: one-dimensional-odd
10010001 11001011 11011001 11011001 11011111 01000000 11101111 11011111 11100101 11011001 11001000
PASS
********************************************************************************
parity_type: one-dimensional-even
10010000 11001010 11011000 11011000 11011110 01000001 10101111 11011110 11100100 11011000 11001001
PASS
********************************************************************************
parity_type: two-dimensional-even
10010000 11001010 11011000 11011000 11011110 01000001 11101110 11011110 11011000 11100100 11001001 00000000
PASS
********************************************************************************
parity_type: two-dimensional-odd
10010001 11001011 11011001 11011001 11011111 01000000 11101111 11011111 11100101 11011001 11001000 11111110
PASS
********************************************************************************
parity_type: two-dimensional-even
10010000 11001010 11011000 11011000 11011110 01000001 11101110 11011110 11011000 11100100 11001001 11100000
FAIL
********************************************************************************
parity_type: two-dimensional-odd
10010001 11001011 11011001 11011001 11011111 01000000 11101111 11011111 11100101 11011001 11001000 10111110
FAIL
********************************************************************************
```

3.

4. **Checksum**

Generator

Input the only word. The program will generate a binary of the checksum by adding all numbers of each alphabet together. After that it will 1'complement our checksum.

```
Codeword: Parin
[80, 97, 114, 105, 110]
code
0111111010
Tmp1:
1111010
Tmp2:
011
result
sum
1111101
Before 1 complement:
1111101
After 1 complement:
Check sum is 0000010
```

Checking

After we get that number, the program will add that check sum number with the alphabet again. If the checksum after 1'complement is all 0's. It means that our data word is correct.

```
**************************
Checking method
sum
111111100
Tmp1:
1111100
Tmp2:
11
result
sum
1111111
Before 1 complement:
1111111
After 1 complement:
Check sum is 0000000
Your data is correct
```

5. **Hamming Code**

Generator: Input the bit string if the size is less than 7 it will automatically increase to size 7 so it can proceed to the generating method. The checker is in fixed position then its

```
Hamming_gen('101001')
Hamming_gen('1001101')
Hamming_gen('10001')
Hamming_gen('0000101')
Hamming_gen('100011')
Hamming_gen('0000101')
Hamming_gen('100111')
Hamming_gen('1001')
Hamming_gen('00101')
Hamming_gen('1001')
```

```
result: 01011001110
**********************************************
result: 10011100101
**********************************************
result: 00110000110
**********************************************
result: 00000101101
**********************************************
result: 01010011100
**********************************************
result: 00000101101
**********************************************
result: 01010110110
**********************************************
result: 00001001100
**********************************************
result: 00000101101
**********************************************
result: 00001001100
**********************************************
```

Checker: Input the bit string of size 11 if not the program will increase the size automatically. The program will return -1 if there is no error (In the return -1 test case, I use the bit string that generates from the generator so it also means that our generator and checker are correct), if error returns the position of the error bit.

```
Hamming_check('01011001110')
Hamming_check('10011100101')
Hamming_check('00110000110')
Hamming_check('01010011100')
Hamming_check('00001001100')
Hamming_check('00001101100')
Hamming_check('01000101101')
Hamming_check('00010101101')
Hamming_check('01000101101')
Hamming_check('00000101100')
```

```
-1
-1
-1
-1
-1
6
10
8
10
1
```