Github link for Q1:
https://github.com/AaliyahSalia/CE305_HW2_Week4/blob/main/Week4_HW2_Q1.py

1. Cyclic Redundancy Check (CRC) is one of the popular coding and decoding techniques in the data transmitted over the network for error detection and correction. Given $x^5 + x^2 + 1$ as a CRC generation polynomial from International Telegraph and Telephone Consultative Committee (CCITT), write the encoding and decoding *def* functions in Python for the only 4-bits original binary data. The examples and testcases of the encoding and decoding processes are shown as follows for your programming. After that, discuss how many bits errors CRC can detect.

```
def encoding(msg, poly):
  """
    org_sig1 = '1010'          # original binary data
    poly = '100101'            # x^5 + x^2 + 1 => b5b4b3b2b1b0 = 100101
    encoding (org_sig1, poly) # find the reminder from 1010 00000 % 100101 = 00111
     '1010 00111'              # encoded output

    org_sig2 = '1100'          # original binary data
    poly = '100101'
    encoding (org_sig2, poly) # find the reminder from 1100 00000 % 100101 = 11001
     '1100 11001'              # encoded output
  """


def decoding(rcv, poly):
  """
   received_sig1 = '1010 00111' # if receiving the data without error
   poly = '100101'              # x^5 + x^2 + 1 => b5b4b3b2b1b0 = 100101
   decoding (received_sig1, poly) # 1010 00111 % 100101 = 00000 (reminder is zero)
   'No error'

   received_sig2 = '1010 01111' # if receiving the data with 1-bit error
   poly = '100101'
   decoding (received_sig2, poly) # 1010 01111 % 100101 = 01000 (reminder is NOT zero)
   'Error'

    received_sig3 = '1100 11001' # if receiving the data without error
    poly = '100101'
    decoding (received_sig3, poly) # 1100 11001 % 100101 = 00000 (reminder is zero)
    'No error'

    received_sig4 = '1100 11111' # if receiving the data with 2-bits error
    poly = '100101'
    decoding (received_sig4, poly) # 1100 11111% 100101 = 00110 (reminder is NOT zero)
    'Error'

  """
```

CODE

```
def binary_long_division(dividend, divisor):
n = len(divisor)
#Append zeroes to the divident
dividend += '0' * (n-1)
for i in range(len(dividend)-n+1):
if dividend[i] == '1':
for j in range(n):
#XOR operation
dividend = dividend[:i+j] + str(int(dividend[i+j]) ^ int(divisor[j])) + dividend[i+j+1:]
return dividend[-(n-1):]

def encoding(msg, poly):
remainder = binary_long_division(msg, poly)
return msg + " " + remainder

def decoding(rcv, poly):
remainder = binary_long_division(rcv.replace(" ", ""), poly)
if '1' in remainder:
return 'Error'
else:
return 'No Error'

#Test Cases
org_sig1 = '1010'
poly = '100101'
print(encoding(org_sig1, poly))

org_sig2 = '1100'
poly = '100101'
print(encoding(org_sig2, poly))
```

```
received_sig1 = '1010 00111'
print(decoding(received_sig1, poly))

received_sig2 = '1010 01111'
print(decoding(received_sig2, poly))

received_sig3 = '1100 11001'
print(decoding(received_sig3, poly))

received_sig4 = '1100 11111'
print(decoding(received_sig4, poly))
```

OUTPUT

```
aaliyahsalia - 2023-10-17 00:23:34 $ /opt/homebrew/bin/python3 "/Us
rs/aaliyahsalia/Desktop/SFBU/6th Trimester/CE305/HW2/Q1.py"
1010 00111
1100 11001
No Error
Error
No Error
Error
aaliyahsalia - 2023-10-17 00:24:27 $
```

Regarding the error detection capability of CRC:

CRC's error-detecting capability depends on the chosen polynomial. CRC can effectively detect:

- All single-bit errors.
- All two-bit errors (provided the polynomial used has at least three terms).
- Errors that affect an odd number of bits.
- Any burst error of length less than the degree of the polynomial.
- Most larger burst errors.

However while CRC is good at detecting errors, it cannot correct them.

Github link for Q2:
https://github.com/AaliyahSalia/CE305_HW2_Week4/blob/main/Week4_HW2_Q2.py

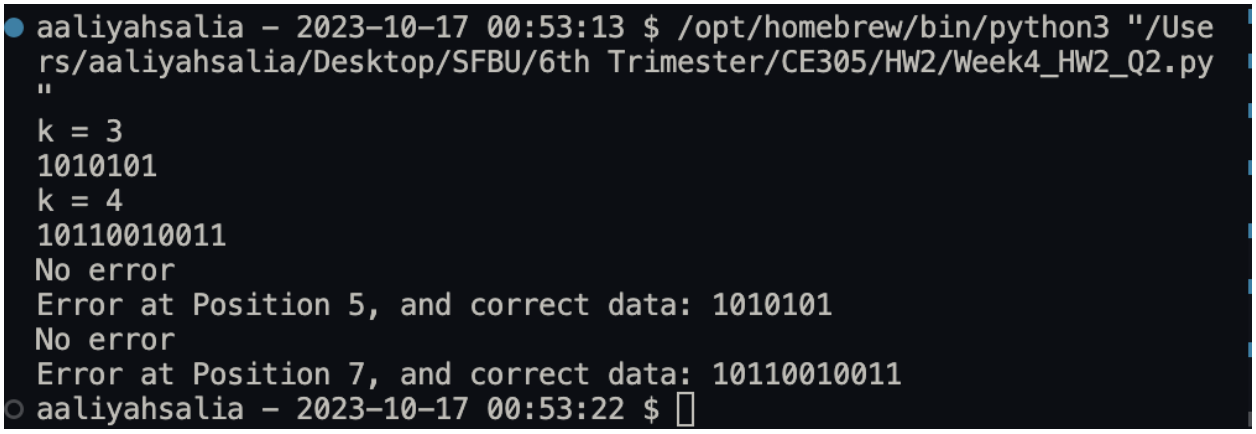
CODE

```
def HamEncoding(msg):
m = len(msg)
k = 0
while (2 ** k) < m + k + 1:
k += 1
encoded = ['0'] * (m + k)
# Placing message bits at appropriate positions
j = 0
for i in range(1, m + k + 1):
if i == 2 ** j:
j += 1
else:
encoded[i-1] = msg[i-j-1]

# Setting parity bits
for i in range(k):
pos = 2**i
parity_bit_val = 0
for j in range(1, len(encoded) + 1):
if j & pos:
parity_bit_val ^= int(encoded[j-1])
encoded[pos-1] = str(parity_bit_val)

print("k =", k)
return ''.join(encoded)
```

```
def HamDecoding(rcv, k):
n = len(rcv)
error_pos = 0

# Calculating error position
for i in range(k):
pos = 2**i
parity_bit_val = 0
for j in range(1, n + 1):
if j & pos:
parity_bit_val ^= int(rcv[j-1])
error_pos += parity_bit_val * pos

if error_pos == 0:
print("No error")
else:
# Correcting the error
rcv_list = list(rcv)
rcv_list[error_pos - 1] = '1' if rcv[error_pos-1] == '0' else '0'
print(f"Error at Position {error_pos}, and correct data: {''.join(rcv_list)}")

# Testing
org_sig1 = '1101'
print(HamEncoding(org_sig1))

org_sig2 = '1001011'
print(HamEncoding(org_sig2))

received_sig1 = '1010101'
HamDecoding(received_sig1, 3)
```

```
received_sig2 = '1010001'
HamDecoding(received_sig2, 3)

received_sig3 = '10110010011'
HamDecoding(received_sig3, 4)

received_sig4 = '10110000011'
HamDecoding(received_sig4, 4)
```

OUTPUT


aaliyahsalia – 2023-10-17 00:53:13 $ /opt/homebrew/bin/python3 "/Users/aaliyahsalia/Desktop/SFBU/6th Trimester/CE305/HW2/Week4_HW2_Q2.py"
k = 3
1010101
k = 4
10110010011
No error
Error at Position 5, and correct data: 1010101
No error
Error at Position 7, and correct data: 10110010011
aaliyahsalia – 2023-10-17 00:53:22 $