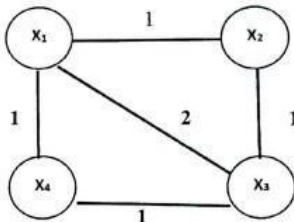


INDEX

SL. NO	Experiment Name	Page																									
01	Write a program to implement Huffman code using symbols with their corresponding probabilities.	02																									
02	Write a program to simulate convolutional coding based on their encoder structure.	10																									
03	Write a program to implement Lempel-Ziv code.	13																									
04	Write a program to implement Hamming code.	17																									
05	A binary symmetric channel has the following noise matrix with probability, $P(Y/X) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$ Now find the Channel Capacity C.	17																									
06	Write a program to check the optimality of Huffman code.	19																									
07	Write a code to find the entropy rate of a random walk on the following weighted graph 	21																									
08	Write a program to find conditional entropy and joint entropy and mutual information based on the following matrix. <table><tr><th>$Y \backslash X$</th><th>1</th><th>2</th><th>3</th><th>4</th></tr><tr><th>1</th><td>$\frac{1}{8}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{32}$</td><td>$\frac{1}{32}$</td></tr><tr><th>2</th><td>$\frac{1}{16}$</td><td>$\frac{1}{8}$</td><td>$\frac{1}{32}$</td><td>$\frac{1}{32}$</td></tr><tr><th>3</th><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td></tr><tr><th>4</th><td>$\frac{1}{4}$</td><td>0</td><td>0</td><td>0</td></tr></table>	$Y \backslash X$	1	2	3	4	1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$	2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$	3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	4	$\frac{1}{4}$	0	0	0	25
$Y \backslash X$	1	2	3	4																							
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$																							
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$																							
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$																							
4	$\frac{1}{4}$	0	0	0																							

Experiment no: 01

Experiment name: Explain and implementation of Huff-man code.

Theory: Huffman coding is a widely used algorithm for compressing data by assigning variable-length codes to input symbols, where shorter codes are assigned to more frequent symbols, and longer codes are assigned to less frequent ones. This technique is known as a prefix code, meaning no code is a prefix of any other code, ensuring the encoded data can be uniquely decoded. Huffman coding provides an optimal solution for minimizing the average length of encoded messages, making it a compact code.

Huffman coding produces a compact code by ensuring that the average length of the code (denoted as L) is the smallest possible for a given set of symbols and their associated probabilities. This allows efficient data compression, especially when symbol frequencies are unequal.

Binary Huffman coding algorithm: The steps for creating a binary Huffman code can be described as follows:

1. **Initialization:**
 - Begin by listing all the symbols (or characters) along with their associated probabilities or frequencies of occurrence in descending order.
2. **Splitting Stage:**
 - Identify the two symbols with the lowest probabilities. Assign a binary value (e.g., '0' and '1') to these two symbols, and combine them into a single symbol with a probability equal to the sum of their original probabilities.
3. **Iteration:**
 - Repeat the process by considering the new combined symbol along with the other symbols, reordering the list by probabilities.
 - Continue combining the two least probable symbols in each step until you are left with only two symbols.
4. **Code Assignment:**
 - Once you have the final two symbols, assign '0' and '1' to them.
 - Work backward through the sequence of combinations to assign codes to each of the original symbols by tracing the 0s and 1s assigned in each merging step.

Example: Consider a 5 symbol source with the following probability assignments:

$$P(s_1) = 0.2, \quad P(s_2) = 0.4, \quad P(s_3) = 0.1, \quad P(s_4) = 0.1, \quad P(s_5) = 0.2$$

Re ordering in decreasing order of symbol probability produces $\{s_2, s_1, s_5, s_3, s_4\}$. The re ordered source is then reduced to the source, s_3 with only two symbols as shown in figure- 1, where the arrow heads point to the combined symbol created in s_j by s_{j-1} starting with the trivial compact code of $\{0,1\}$ for s_3 and working back to s_1 a compact code is designed for each reduced source s_j and shown in figure 1. In each s_j the code word for the last two symbols is produced by taking the code word of the symbol pointed to by the arrow head and appending 0 and 1 to form two new code words. The Huffman code itself is the bit sequence generated by the path from the root to the corresponding leaf node.

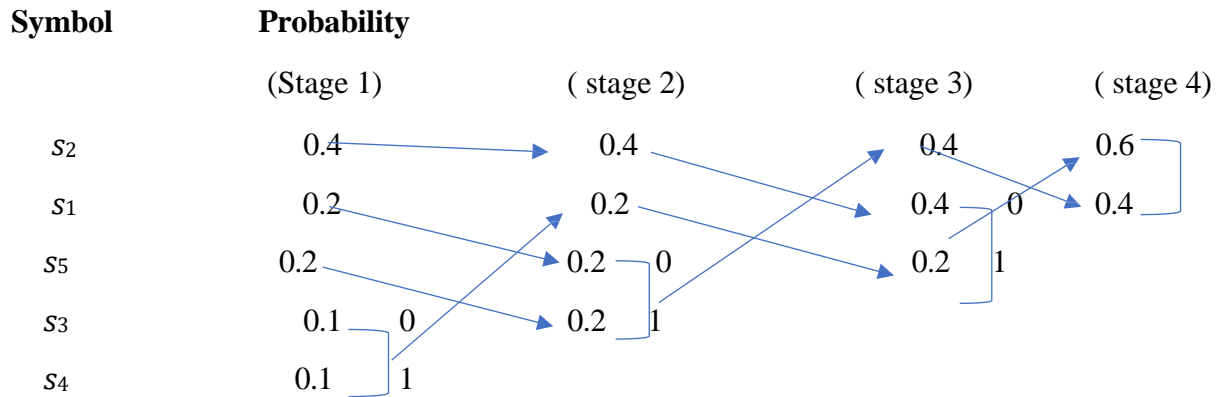


Figure-1: Binary Huffman coding table.

The binary Huffman code is:

Symbol	$P(s_1)$	Huffman code
S ₁	0.2	10
S ₂	0.4	00
S ₃	0.1	010
S ₄	0.1	011
S ₅	0.2	11

Figure: Huffman Code Table.

Source Code:

```
import heapq
from collections import defaultdict, Counter

def calculate_frequency(my_text):
    my_text = my_text.upper().replace(' ', '')
    frequency = dict(Counter(my_text))
    return frequency
```

```

def build_heap(freq):
    heap = [[weight, [char, ""]] for char, weight in freq.items()]
    heapq.heapify(heap)
    return heap

def build_tree(heap):
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return heap[0]

freq = calculate_frequency("aaabbbbccccccdddee")
heap = build_heap(freq)
tree = build_tree(heap)
for pair in tree[1:]:
    print(pair[0], '->', pair[1])

```

Input:

Data: “ aaabbbbccccccdddee ”

Output:

D -> 00
 B -> 01
 E -> 100
 A -> 101
 C -> 11

Experiment No: 02

Experiment Name: Explain and Implementation of Convolution Coding

Theory:

Convolution Coding:

Convolutional coding or Trellis coding is a type of error-correcting code that introduces memory into the coding process. Unlike block codes, where the encoding of each block is independent, convolutional codes use memory elements to make the current output depend not only on the current input but also on a certain number of previous inputs. This memory effect significantly improves the error-correcting capability of the code, making it more resilient to errors caused by noisy communication channels.

The rate of a convolutional encoder, denoted by R_t , is defined as the ratio of input information bits (n_i) to output codeword bits (n_c). For example, if one input bit generates two output bits, the rate is $R_t=1/2$. In general, $R_t=n_i/n_c$.

Consider a convolutional encoder where the input sequence consists of binary information bits ($m_0, m_1, m_2, \dots, m_n$). These bits are fed into a finite-state machine that has a memory, typically implemented using shift registers. The encoder also contains several modulo-2 adders, which perform binary addition (XOR operation) to generate the codeword bits.

For each input bit, the encoder produces multiple output bits. In the example given, for every input bit, two output bits are generated, resulting in a rate of $R_t=1/2$. The output bits are generated based on both the current input bit and the values stored in the shift registers from previous inputs.

In general, the encoder can take n_i information bits to generate n_c codeword bits yielding an encoder rate of $R_t = n_i/n_c$ bit

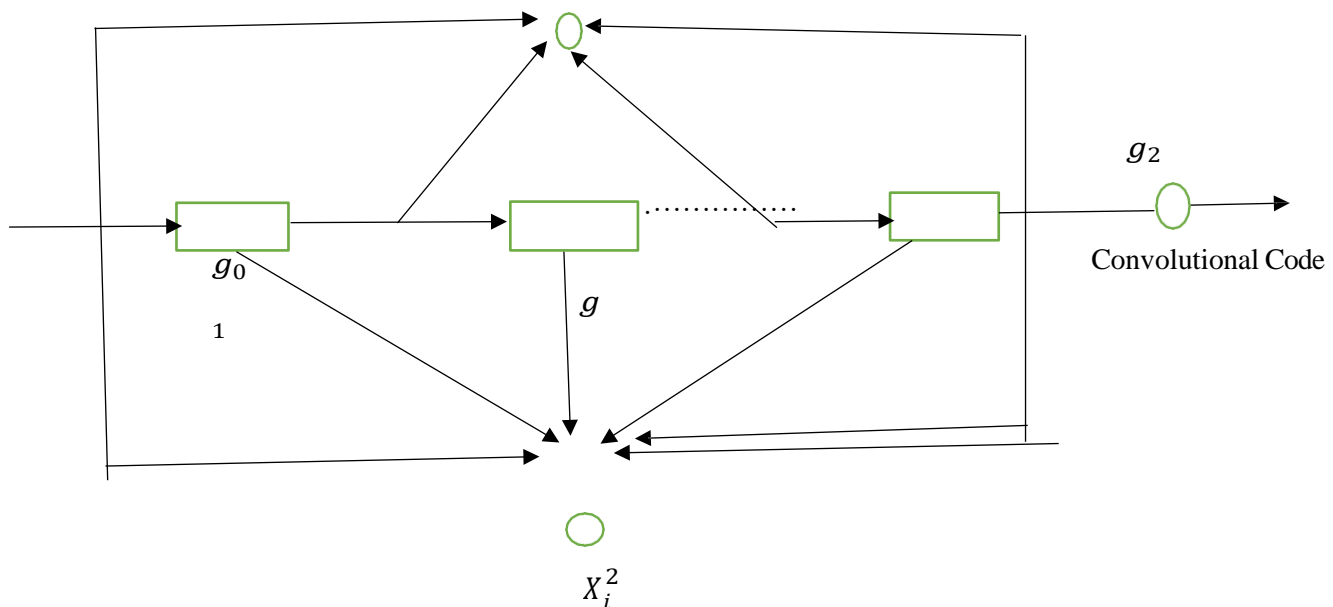


Figure: Convolutional Code

To make sure that the outcome of the encoder is a deterministic function of the sequence of input bits, we ask the memory cells of the encoder to contain zeros at the beginning of the encoding process.

Moreover, once L information bits have been encoded, we stop the information bit sequence and will feed T dummy zero-bits as inputs instead, where T is chosen to be equal to the memory size of the encoder. These dummy bits will make sure that the state of the memory cells are turned back to zero. Here in the above diagram,

L = the message length,

m = no. of shift registers,

n = no. of modulo-2 adders.

Output = $n(m+L)$ bit and code rate, $r = L/n(m+L)$; $L \gg m$

$$= L/(Ln)$$

$$= 1/n$$

Constraint length, $K = m+1$. If $g_0^{(1)}, g_1^{(1)}, g_2^{(1)}, \dots, g_m^{(1)}$ are the state of shift registers. Then The input-top adder output path is given by:

$$g_0^{(1)}, g_1^{(1)}, g_2^{(1)}, \dots, g_m^{(1)}$$

and the input-bottom adder output path is given by:

$$g_0^{(2)}, g_1^{(2)}, g_2^{(2)}, \dots, g_m^{(2)}$$

Let the message sequence be $m_0, m_1, m_2, \dots, m_n$.

Then convolution sum for (1)

$$X_i^1 = \sum_{l=0}^m g_l^1 m_{i-l} \quad ; \quad i = 0, 1, 2, \dots, n$$

And, then convolution sum for (1)

$$X_i^1 = \sum_{l=0}^m g_l^2 m_{i-l} \quad ; \quad i = 0, 1, 2, \dots, n$$

So output, $X_i = \{ x_0^{(1)} x_0^{(2)} x_1^{(1)} x_1^{(2)} x_2^{(1)} x_2^{(2)} \dots \}$

Math:

Input:

* Top output path: $(g_0^{(1)}, g_1^{(1)}, g_2^{(1)}) = (1, 1, 1)$

* Bottom output path: $(g_0^{(2)}, g_1^{(2)}, g_2^{(2)}) = (1, 0, 1)$

* Message bit sequence = $(m_0, m_1, m_2, m_3, m_4) = (1, 0, 0, 1, 0)$

Solution:

We know that:

$$X_i^1 = \sum_{l=0}^m g^2 m_{i-l}$$

When $j=1$ & $i=0$ then,

$$x_0^{(1)} = g_0^{(1)} m_0 = 1 \times 1 = 1 \% 2 = 1$$

Then for successive i , we get:

$$x_1^{(1)} = g_{0m_1}^1 + g_{1m_0}^1 = 1 \times 0 + 1 \times 1 = 0 + 1 = 1 \% 2 = 1$$

$$x_2^{(1)} = g_{0m_2}^1 + g_{1m_1}^1 = 1 \times 0 + 1 \times 0 + 1 \times 1 = 0 + 1 = 1 \% 2 = 1$$

$$x_3^{(1)} = g_{0m_3}^1 + g_{1m_0}^1 + g_{2m_1}^1 = 1 \times 1 + 1 \times 0 + 1 \times 0 = 1 + 0 + 0 = 1 \% 2 = 1$$

$$x_4^{(1)} = g_{0m_4}^1 + g_{1m_3}^1 + g_{2m_2}^1 = 1 \times 1 + 1 \times 1 + 1 \times 0 = 1 + 1 + 0 = 2 \% 2 = 0$$

$$x_5^{(1)} = g_{1m_4}^1 + g_{2m_1}^1 = 1 \times 1 + 1 \times 1 = 1 + 1 = 2 \% 2 = 0$$

$$x_6^{(1)} = g_{2m_4}^1 = 1 \times 1 = 1 \% 2 = 1$$

$$X_i^1 = 1111001$$

When $j = 2$ & $I = 0$ then,

$$x_0^{(2)} = g_{0m_0}^2 = 1 \times 1 = 1 \% 2 = 1$$

Then for successive i , we get:

$$x_1^{(2)} = g_{0m_1}^2 + g_{1m_0}^2 = 1 \times 0 + 0 \times 1 = 0 + 0 = 0$$

$$x_2^{(2)} = g_{0m_2}^2 + g_{1m_1}^2 + g_{2m_0}^2 = 1 \times 0 + 0 \times 0 + 1 \times 1 = 0 + 1 = 1 \% 2 = 1$$

$$x_3^{(2)} = g_{0m_3}^2 + g_{1m_2}^2 + g_{2m_1}^2 = 1 \times 1 + 0 \times 0 + 1 \times 0 = 1 + 0 + 0 = 1 \% 2 = 1$$

$$x_4^{(2)} = g_{0m_4}^2 + g_{1m_3}^2 + g_{2m_2}^2 = 1 \times 1 + 0 \times 1 + 1 \times 0 = 1 + 0 + 0 = 1 \% 2 = 1$$

$$x_5^{(2)} = g_{1m_4}^2 + g_{2m_3}^2 = 0 \times 1 + 1 \times 1 = 1 + 0 = 1 \% 2 = 1$$

$$x_6^{(2)} = g_{2m_3}^2 = 1 \times 1 = 1 \% 2 = 1$$

So,

$$X_i^2 = 1011111$$

$$X_i = 11101111010111$$

Source Code:

```
import numpy as np

def encode(msg, K, n):
    g, v = [], []
    for i in range(n):
        sub_g = list(map(int, input(f'Enter bits for generator {i}: ').split()))
        if len(sub_g) != K:
            raise ValueError(f'You entered {len(sub_g)} bits.\n need to enter {K} bits')
        g.append(sub_g)
    for i in range(n):
        res = list(np.poly1d(g[i]) * np.poly1d(msg))
        v.append(res)

    listMax = max(len(l) for l in v)
    for i in range(n):
        if len(v[i]) != listMax:
            tmp = [0] * (listMax - len(v[i]))
            v[i] = tmp + v[i]
    res = []
    for i in range(listMax):
        res += [v[j][i] % 2 for j in range(n)]
    return res

message = list(map(int, input('Enter message: ').split()))
K = int(input('Constraints: '))
n = int(input('Number of output(generator): '))
print('Encoded Message', encode(message, K, n))

# message= 1 0 1 0 1
# K=4
# n=2
# g0= 1 1 1 1
# g1= 1 1 0 1
# Encoded Message: 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1 1
```


Input:

Enter message: 1 0 1 0 1

Constraints: 4

Number of output(generator): 2

Enter bits for generator 0: 1 1 1 1

Enter bits for generator 1: 1 1 0 1

Output:

Encoded Message: [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1]

Experiment No: 03

Experiment Name: Explain and implementation of Lempel – ziv code using Python.

Theory:

Lempel–Ziv coding is a universal lossless data compression algorithm developed by Abraham Lempel and Jacob Ziv. It serves as the basis for various compression methods, including the widely used Unix file compression utility compress and the GIF image format. The Lempel–Ziv algorithm operates by building a dictionary of previously seen substrings during the encoding process, allowing for efficient compression by identifying and encoding repeated patterns in the data.

To illustrate, let us consider an input binary sequence as follows:

000101110010100101...

Let us assume 0 & 1 are already stored so, subsequence stored: 0, 1

Data to be parsed : 000101110010100101...

The encoding process begins at left. As 0 & 1 are already stored. The shortest sub-sequence of data stream are written as,

Subsequence stored : 0, 1

Data to be parsed : 000101110010100101....

The second and the sequences are:

Subsequence stored: 0, 1, 00, 01,

Data to be parsed: 01110010100101...

We continue this until the given data stream is completely parsed. Now the binary code blocks of the sequences are,

<u>Numerical Positions:</u>	1	2	3	4	5	6	7	8	9
<u>Subsequence:</u>	0	1	00	01	011	10	010	100	101
<u>Numerical representations:</u>	11	12	42	21	41	61	62		
<u>Binary encoded blocks:</u>	0010	0011	1001	0100	1000	1100	1101		

Figure: Illustrating the encoding process performed by the Lempel – Ziv algorithm.

From the figure, the first now show the numerical position of individual subsequence in the code. A sequence of data stream 010 consist of the concatenation of the sub- sequence 01 in position 4 and symbol 0 in position 1; hence the numerical representation is 41. Similarly others are.

The last row in figure is the binary subsequence is Binary encoded blocks. The last symbol of each sequence in the code book is an innovation symbol. Which is so called in recognition of the fact that can distinguishes it from all previous subsequence stored in the code book. The last bit of each uniform blocks of bits represents the innovation symbol and the remaining bits provide the equivalent binary representation of the “printer” to the root subsequence that matches the one in question expect for the innovation symbol.

The decoder is Just simple as the encoder. Use the pointer to identify the root subsequence and appends the innovation symbol. Such as the binary blocks encoded blocks 1101 in position 9. The last bits is the innovation symbol. Here 110 point to the root subsequence 10 in position 6. Here, the blocks 1101 is decoded into 101, which is correct.

In contrast of Huffman coding, the Lempel – Zip algorithm uses fixed length codes to represent a variable numbers of source symbols. That makes Lempel – Ziv coding suitable for synchronous transmission.

Python Code:

```
message = 'AABABBBABAABABBBABBABB'
dictionary = { }
tmp, i, last = "", 1, 0
Flag = True
for x in message:
    tmp += x
    Flag = False
    if tmp not in dictionary.keys():
        dictionary[tmp] = i
        tmp = ""
        i += 1
        Flag = True

if not Flag:
    last = dictionary[tmp]

res = ['1']
for char, idx in list(dictionary.items())[1:]:
    tmp, s = "", ""
    for x, j in zip(char[:-1], range(len(char))):
        tmp += x
        if tmp in dictionary.keys():
            take = dictionary[tmp]
```

```
s = str(take) + char[j + 1:]
if len(char) == 1:
    s = char
res.append(s)
if last:
    res.append(str(last))

mark = {
    'A': 0,
    'B': 1
}

final_res = []
for x in res:
    tmp = ""
    for char in x:
        if char.isalpha():
            tmp += bin(mark[char])[2:]
        else:
            tmp += bin(int(char))[2:]
    final_res.append(tmp.zfill(4))

print(res)
print("Encoded: ", final_res)
```

Input:

message = 'AABABBBABAABABBBABBABB '

Output:

['1', '1B', '2B', 'B', '2A', '5B', '4B', '3A', '7']

Encoded: ['0001', '0011', '0101', '0001', '0100', '1011', '1001', '0110', '0111']

Experiment No: 04

Experiment Name: Write a program to implement Hamming code

Theory:

Hamming code is a widely used error-detecting and error-correcting code that ensures data integrity during transmission or storage. It was developed by Richard Hamming in the 1950s to solve the problem of single-bit errors that occur when data is transferred between systems. Hamming code not only detects single-bit errors but also corrects them, thus improving the reliability of communication systems and digital storage devices. This makes it a robust and efficient method for ensuring data accuracy.

Steps to Create Hamming Code:

1. Determine the number of parity bits required: For a data block of size 'm', the number of parity bits 'r' must satisfy the inequality:

$$2^r \geq m + r + 1$$

his inequality ensures that the parity bits can cover all data bits and themselves.

2. Place the parity bits in the data: The positions of the parity bits are powers of two (1, 2, 4, 8, ...).
3. Calculate the values of the parity bits: Each parity bit checks a different combination of data bits. For example, the parity bit at position 1 checks bits 1, 3, 5, 7, etc. The value of the parity bit is set such that the total number of 1s in its combination is even (even parity).
4. Transmit the data along with the parity bits.
- 5.

Steps for Error Detection and Correction with Hamming Code:

1. Step 1: Receiver Gets the Data
2. Step 2: Calculate Parity Check Bits (Syndrome)
 - The receiver recalculates the parity bits (P1, P2, and P4) for the received data, just like we did during encoding.
 - Each parity bit checks a specific set of bits and ensures that their sum is even (for even parity). If the parity calculation doesn't match, the receiver knows there's an error.
 - For a 7-bit Hamming code, the receiver calculates the syndrome using the parity check bits. If the syndrome is 0 then there is no error. If the syndrome is non-zero, then an error has occurred.
3. Step 3: Syndrome Calculation The syndrome is a 3-bit binary number calculated by checking each of the parity bits (P1, P2, and P4):
 - a. **Syndrome Calculation for P1:** P1 checks positions 1, 3, 5, and 7.
 - i. Sum the bits at these positions and see if the sum is even.
 - b. **Syndrome Calculation for P2:** P2 checks positions 2, 3, 6, and 7.
 - i. Sum the bits at these positions and see if the sum is even.
 - c. **Syndrome Calculation for P4:** P4 checks positions 4, 5, 6, and 7.
 - i. Sum the bits at these positions and see if the sum is even.

Each of these checks results in either a 0 or 1 (even or odd sum), forming a 3-bit syndrome that indicates which bit (if any) has been corrupted.

4. Step 4: Error Detection
 - a. If the syndrome is 000. there is no error (the data is correct).
 - b. If the syndrome is a non-zero value (e.g. 001, 010, 100 etc.), it means there is an error at the bit position indicated by the syndrome.
5. Step 5: Error Correction

Once the receiver knows the position of the error (based on the syndrome), it can flip the erroneous bit (from 0 to 1 or from 1 to 0) to correct the message.
6. Step 6: Output the Corrected Data

After detecting and correcting any error, the receiver now has the correct data and can extract the original data bits.

Example:

Suppose we want to transmit 4 bits of data: 1011

1. **Determine the number of parity bits:** Since we have 4 data bits, we need 3 parity bits to satisfy

$$2^r \geq m + r + 1 \text{ i.e., } 2^3 \geq 4 + 3 + 1$$

2. **Position the parity bits:** Place the parity bits at positions 1, 2, and 4:

_ _ 1 _ 0 1 1

Here, _ represents the positions of parity bits.

3. **Calculate the parity bits:**

- Parity bit at position 1 (P1): Covers bits 1, 3, 5, 7: 1, 1, 0 → P1 = 0 (since 1+1+0 is even).
- Parity bit at position 2 (P2): Covers bits 2, 3, 6, 7: 0, 1, 1 → P2 = 0 (since 0+1+1 is even).
- Parity bit at position 4 (P4): Covers bits 4, 5, 6, 7: 0, 1, 1, 1 → P4 = 1 (since 0+1+1+1 is odd).

Now the transmitted data becomes (even parity): 0110011.

Let's say the receiver gets the following **received data** (which has an error)

Received Data: 0110111

Now, let's calculate the syndrome:

Syndrome for P1 (positions 1, 3, 5, 7): Bits: 0, 1, 0, 1 → Sum = 0+1+0+1=2 (even), so Syndrome P1 = 0.

Syndrome for P2 (positions 2, 3, 6, 7): Bits: 1, 1, 1, 1 → Sum = 1+1+1+1=4 (even), so Syndrome P2 = 0.

Syndrome for P4 (positions 4, 5, 6, 7): Bits: 0, 0, 1, 1 → Sum = 0+0+1+1=2 (even), so Syndrome P4 = 0.

Now, let's check the syndrome:

Syndrome = 000, meaning there's **no error** in the received data.

But if we have **Syndrome = 001**, it means the error is at position 1, and the receiver will flip the bit at position 1 to correct the data.

Corrected Data: 0110011

Source Code:

```
def calculate_parity_bits(data_bits):
    # Place data bits in positions 3, 5, 6, and 7
    d3, d5, d6, d7 = map(int, data_bits)
    # Calculate parity bits for even parity
    # P1 covers positions 1, 3, 5, 7
    p1 = (d3 + d5 + d7) % 2
    # P2 covers positions 2, 3, 6, 7
    p2 = (d3 + d6 + d7) % 2
    # P4 covers positions 4, 5, 6, 7
    p4 = (d5 + d6 + d7) % 2

    # Return the encoded message
    hamming_code = f"{p1}{p2}{d3}{p4}{d5}{d6}{d7}"
    return hamming_code

def detect_and_correct_error(received_data):
    # Convert the received data to a list of integers for easier manipulation
    received_bits = list(map(int, received_data))

    # Check parity for P1, P2, and P4
    p1_check = (received_bits[0] + received_bits[2] + received_bits[4] + received_bits[6]) % 2
    p2_check = (received_bits[1] + received_bits[2] + received_bits[5] + received_bits[6]) % 2
    p4_check = (received_bits[3] + received_bits[4] + received_bits[5] + received_bits[6]) % 2

    # Calculate the syndrome (error position)
    error_position = p1_check * 1 + p2_check * 2 + p4_check * 4

    if error_position == 0:
        return "No error detected", received_data # No error
    else:
        # Correct the error at the error_position
        print(f"Error detected at position {error_position}. Correcting it.")
        received_bits[error_position - 1] = 1 - received_bits[error_position - 1] # Flip the
        erroneous bit
        return "Error detected and corrected", received_bits

# Example usage
data_bits = "1011" # Data bits to encode
print(f>Data: {data_bits}<

# Calculate the encoded message (Hamming code)
encoded_data = calculate_parity_bits(data_bits)
print(f>Encoded Data: {encoded_data}<
```

<pre># Simulate a transmission with an error (let's say bit 6 has an error) received_data_with_error = "0110111" # This is the received data with a single-bit error print(f'Received Data (with error): {received_data_with_error}') # Detect and correct errors status, corrected_data = detect_and_correct_error(received_data_with_error) print(status) print(f'Corrected Data: {"".join(map(str, corrected_data))}')</pre>	
---	--

Output:

```
Data: 1011
Encoded Data: 0110011
Received Data (with error): 0110111
Error detected at position 5. Correcting it.
Error detected and corrected
Corrected Data: 0110011
```