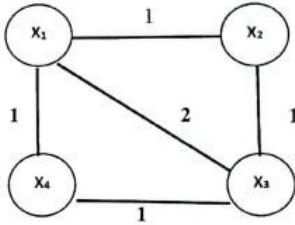


INDEX

SL	Experiment Name	Page																																				
1.	Write a program to implement Huffman code using symbols with their corresponding probabilities.	2																																				
2.	Write a program to simulate convolutional coding based on their encoder structure.	5																																				
3.	Write a program to implement Lempel-Ziv code.	9																																				
4.	Write a program to implement Hamming code.	11																																				
5.	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.	15																																				
6.	Write a program to check the optimality of Huffman code.	17																																				
7.	Write a program to find conditional entropy and join entropy and mutual information based on the following matrix. <table><tr><td></td><td>X</td><td></td><td></td><td></td><td></td></tr><tr><td>Y</td><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td></td><td>$\frac{1}{8}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{32}$</td><td>$\frac{1}{32}$</td></tr><tr><td>2</td><td></td><td>$\frac{1}{16}$</td><td>$\frac{1}{8}$</td><td>$\frac{1}{32}$</td><td>$\frac{1}{32}$</td></tr><tr><td>3</td><td></td><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td><td>$\frac{1}{16}$</td></tr><tr><td>4</td><td></td><td>$\frac{1}{4}$</td><td>0</td><td>0</td><td>0</td></tr></table>		X					Y		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	18
	X																																					
Y		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																																	
8.	Write a code to find the entropy rate of a random walk on the following weighted graph 	20																																				

Experiment no: 01

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

Theory:

Huffman coding: An important class of prefix code known as Huffman codes. The basic idea behind Huffman coding is to assign to each symbol of an alphabet a sequence of bits roughly equal in length to the amount of information conveyed by the symbol in equation. Huffman codes are compact codes. That is, the Huffman algorithm produces a code with an average length, L , which is the smallest possible to achieve for the given number of source symbols, code alphabet and source statistics.

Binary Huffman coding algorithm: For the design of binary Huffman coding the Huffman coding algorithm is as follows:

1. The source symbol is listed in order of decreasing probability. The two symbols of lowest probability are assigned a_0 and a_1 . This part of the step is referred to as a splitting stage.
2. These two source symbols are regarded as being combined into a new source symbol with probability equal to the sum of two original probabilities. The probability of the new symbol is placed in the accordance with its value.
3. The procedure is repeated until we are left with a final list of source statistics of only two for which a_0 and a_1 are assigned.

The code for each source symbol is found by working backward tracing the sequence of 0s and 1s assigned to that symbol as well as its successors.

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.

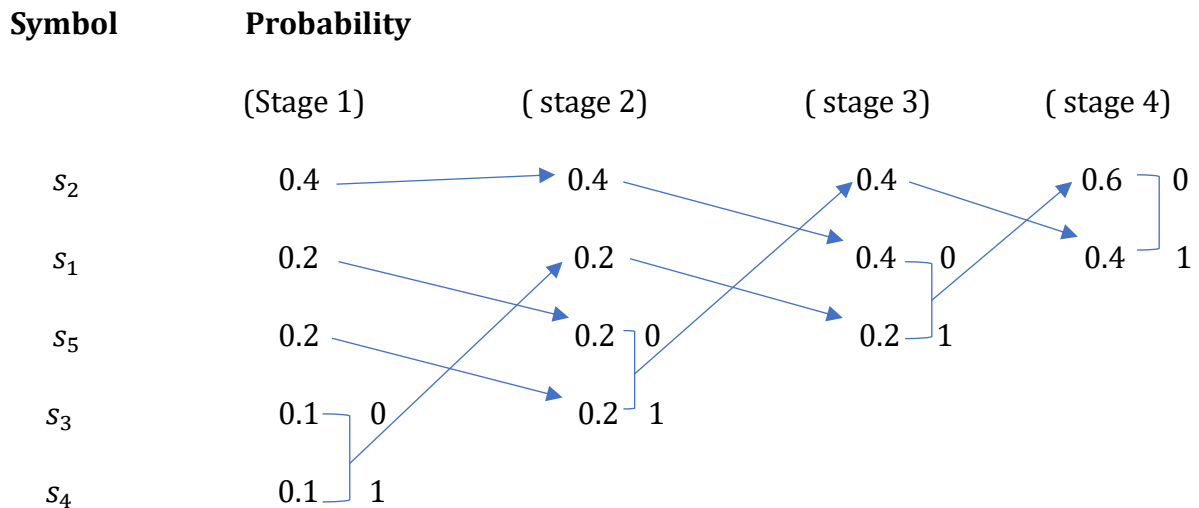


Figure-1: Binary Huffman coding table.

Table-01: Binary Huffman Code Table.

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

Source Code:

```
import huffman
from collections import Counter

# Calculate frequencies and probabilities
def calculate_probabilities(message):
    message = message.upper().replace(" ", "")
    freq = Counter(message)
    P = [fre / len(message) for fre in freq.values()]
    return P, freq

# Input message
message = "aaabbbbbccccccdddee"
P, freq = calculate_probabilities(message)
print("Probabilities:", P)

# Generate Huffman codes using huffman library
```

```
huffman_code = huffman.codebook(freq.items())
```

```
# Display Huffman codes
```

```
print("Huffman Codes:")
```

```
for char, code in huffman_code.items():
```

```
    print(f"{char} -> {code}")
```

Input:

```
message = "aaabbbbccccccddddee"
```

Output:

```
Probabilities: [0.15, 0.25, 0.3, 0.2, 0.1]
```

```
Huffman Codes:
```

```
A -> 101
```

```
B -> 01
```

```
C -> 11
```

```
D -> 00
```

```
E -> 100
```

Experiment No: 02

Experiment Name: Explain and Implementation of Convolution Coding.

Theory:

Convolution Coding: Convolution codes or Trellis Code introduce memory into the coding process to improve the error-correcting capabilities of the codes. The coding and decoding processes that are applied to error-correcting block codes are memoryless. The encoding and decoding of a block depends only on that block and is independent of any other block. They do this by making the parity checking bits dependent on the bits values in several consecutive blocks.

Say we have a message source that generates a sequence of information digits (Uks). We will assume that the information digits are binary, i.e., information bits. These information bits are bit fed into a convolutional encoder. As an example, consider the encoder shown below. This encoder is a finite state machine that has (a finite) memory. Its current output depends on the current input and on a certain number of past input.

In the example, its memory is 2 bits, i.e., it contains a shift register that keeps stored the values of the last two information bits. Moreover, the encoder has several modulo-2 adders. The output of the encoder is the codeword bits that will be then transmitted over the channel. In our example, for every information bit, two codeword bits are generated. Hence, the encoder rate is:

$$R_t = 1/2 \text{ bits.}$$

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

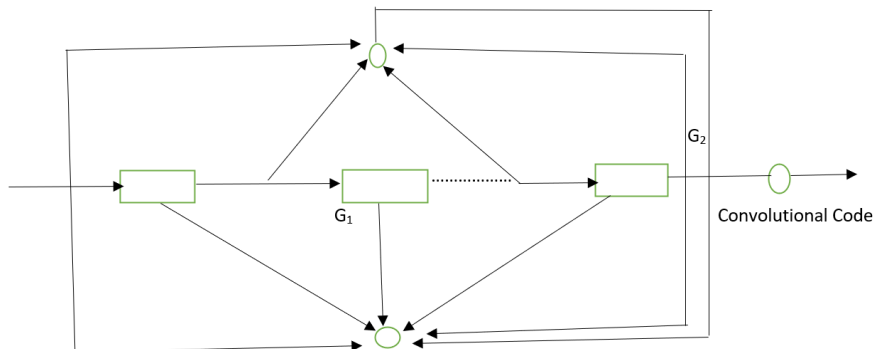


Figure-01: Convolutional Code

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_t 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)

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

And, then convolution sum for (1)

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

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

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:

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

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:

$$x_1^{(1)} = g_0^1 m_1 + g_1^1 m_0 = 1 \times 0 + 1 \times 1 = 0 + 1 = 1 \% 2 = 1$$

$$x_2^{(1)} = g_0^1 m_2 + g_1^1 m_1 = 1 \times 0 + 1 \times 0 + 1 \times 1 = 0 + 1 = 1 \% 2 = 1$$

$$x_3^{(1)} = g_0^1 m_3 + g_1^1 m_0 + g_2^1 m_1 = 1 \times 1 + 1 \times 0 + 1 \times 0 = 1 + 0 + 0 = 1 \% 2 = 1$$

$$x_4^{(1)} = g_0^1 m_4 + g_1^1 m_3 + g_2^1 m_2 = 1 \times 1 + 1 \times 1 + 1 \times 0 = 1 + 1 + 0 = 2 \% 2 = 0$$

$$x_5^{(1)} = g_1^1 m_4 + g_2^1 m_1 = 1 \times 1 + 1 \times 1 = 1 + 1 = 2 \% 2 = 0$$

$$x_6^{(1)} = g_2^1 m_4 = 1 \times 1 = 1 \% 2 = 1$$

$$X_i^1 = 1111001$$

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:

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

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

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

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

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

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

$$X_i^2 = 1011111$$

$$X_i = 11101111010111$$

Source Code:

```
import numpy as np

def encode(msg, K, n, generators):
    g, v = [], []
    for i in range(n):
        sub_g = generators[i]
        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
```

```
# Example values
message = [1, 0, 1, 0, 1] # Message to encode
K = 4 # Constraint length
n = 2 # Number of generators
generators = [
    [1, 1, 1, 1], # Generator polynomial g0
    [1, 1, 0, 1], # Generator polynomial g1
]

# Encoding the message
encoded_message = encode(message, K, n, generators)
print("Encoded Message:", encoded_message)
```

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 is a universal lossless data compression algorithm created by Abraham Lempel, Jacob ziv. It was the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format.

Lampel algorithm is accomplished by parsing the source data stream into segment that are the shortest subsequence not encountered previously. To illustrate, let is consider an input binary sequence as follows:

000101110010100101...

Let us assumed 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 subsequence 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.

Source Code:

```
# Input message
message = "000101110010100101"
dictionary = {"0": 1, "1": 2}
tmp, i = "", 3 # Starting index at 3
Sequence_output = []
Numerical_rep = []
binary_blocks = []

# LZ78 Encoding Algorithm with modified binary encoding
for x in message:
    tmp += x
    # If the current subsequence is not in the dictionary, add it
    if tmp not in dictionary:
        # Update dictionary, output, and binary blocks
        dictionary[tmp] = i

        prefix = tmp[:-1] # N-1 sequence
        prefix_position = dictionary[prefix]
        suffix = tmp[-1] # N-th bit
        suffix_position = dictionary[suffix]
        Numerical_rep.append(f"{prefix_position}{suffix_position}")
        # Combine binary representation of both positions
        binary_block = format(prefix_position, "03b") + suffix

        Sequence_output.append(tmp)
        binary_blocks.append(binary_block)
        tmp = "" # Reset for the next sequence
        i += 1

# Display results
print("Subsequences:      ", Sequence_output)
print("Numerical Representation:", Numerical_rep)
print("Binary Encoded Blocks: ", " ".join(binary_blocks))
```

Output:

Subsequences: ['00', '01', '011', '10', '010', '100', '101']
Numerical Representation: ['11', '12', '42', '21', '41', '61', '62']
Binary Encoded Blocks: 0010 0011 1001 0100 1000 1100 1101

Experiment No: 04

Experiment Name: Write a program to implement Hamming code.

Theory:

Hamming code is an error-correcting code used to ensure data accuracy during transmission or storage. Hamming code detects and corrects the errors that can occur when the data is moved or stored from the sender to the receiver. This simple and effective method helps improve the reliability of communication systems and digital storage. It adds extra bits to the original data, allowing the system to detect and correct single-bit errors. It is a technique developed by Richard Hamming in the 1950s.

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:

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

This 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.

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

Python Code:

```
def isPowerOfTwo(n):
    if n == 0:
        return False
    return n & (n - 1) == 0

message = "1011"
hamming = ""
length = len(message)
print("Message: ", message)

# Calculate redundant bits
r = 0
for i in range(length):
    if 2**i == length + i + 1:
        r = i
        break

# Hamming code with parity positioned
k = 0
for i in range(1, length + r + 1):
    if isPowerOfTwo(i):
        hamming += "p"
    else:
        hamming += message[k]
        k += 1

print("Position generate parity bit :", hamming)

# Calculate parity bits
res = 0
for i in range(len(hamming)):
    if hamming[i] == "1":
        res ^= i + 1

P = bin(res)[2:].zfill(r)[-1]

# Hamming code generate
k = 0
hamming_list = list(hamming)
for i in range(len(hamming)):
    if hamming_list[i] == "p":
        hamming_list[i] = P[k]
        k += 1
```

```
hamming = "".join(hamming_list)
print("Hamming code:", hamming)

# Error detection
rcv = "0111011"
print("Let received bit: ", rcv)
res = 0
for i in range(len(rcv)):
    if rcv[i] == "1":
        res ^= i + 1

if res == 0:
    print("No Error")
else:
    print("Error at position:", res)
```

Output:

Message: 1011
Position generate parity bit : pp1p011
Hamming code: 0110011
Let received bit: 0111011
Error at position: 4

Experiment No: 05

Experiment Name: A binary symmetric channel has the following noise matrix with probability,

$$P(Y/X) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix} \therefore \text{Now find the Channel Capacity } C.$$

Theory:

The channel capacity C for a Binary Symmetric Channel (BSC), we need the channel transition probability matrix, which typically provides the probabilities of bit flips (errors) in a communication system.

For a Binary Symmetric Channel, the transition probability matrix is often represented as:

$$P(x) = \begin{pmatrix} 1-p & p \\ p & 1-p \end{pmatrix}$$

Here:

- p is the probability of a bit flip (i.e., the probability of an error).
- $1-p$ is the probability that the transmitted bit is received correctly.

The channel capacity C for a Binary Symmetric Channel is given by the formula:

$$C = 1 - H(p)$$

Where $H(p)$ is the binary entropy function, defined as:

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

To find the channel capacity, follow these steps:

1. Identify the noise probability p from the noise matrix.
2. Calculate the binary entropy function $H(p)$.
3. Substitute $H(p)$ into the channel capacity formula $C = 1 - H(p)$

Example

$$P(x) = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}$$

1. Identify the noise probability p from the noise matrix.

$$p = \frac{1}{3}$$

2. Calculate the binary entropy function $H(p)$.

$$\begin{aligned} H(p) &= -p \log_2 p - (1-p) \log_2 (1-p) \\ &= -\frac{1}{3} \log_2 \left(\frac{1}{3}\right) - \frac{2}{3} \log_2 \left(\frac{2}{3}\right) = 0.918 \text{ bits/msg symbol} \end{aligned}$$

3. Substitute $H(p)$ into the channel capacity formula C

$$C = 1 - H(p) = 1 - 0.918 = 0.082 \text{ bits/msg symbol}$$

Python Code:

```
import math

matrix = [[2 / 3, 1 / 3], [1 / 3, 2 / 3]]
for i in range(0, 2):
    for j in range(0, 2):
        print(f"{matrix[i][j]:.2f} ", end=" ")
    print()

# Calculate H(Y/X) using formula (1-p)log(1/(1-p))+plog(1/p)
Hp = matrix[0][0] * math.log2(1.0 / matrix[0][0]) + matrix[0][1] * math.log2(
    1.0 / matrix[0][1]
)
print(f"Conditional probability H(Y|X) = {Hp:.2f} bits/msg symbol")

# Now calculate channel capacity using formula C = 1- H(Y/X)
print(f"Channel Capacity = {(1-Hp):.2f} bits/msg symbol")
```

Output:

0.67 0.33

0.33 0.67

Conditional probability H(Y|X) = 0.92 bits/msg symbol

Channel Capacity = 0.08 bits/msg symbol

Experiment No: 06

Experiment Name: Check Optimality of Huffman Code.

Theory:

To check the optimality of a Huffman code, you can use the Kraft inequality, which states that the sum of the code word lengths (in bits) raised to the power of -1 must be less than or equal to 1. If a code satisfies the Kraft inequality, it is considered optimal.

Another way to check the optimality of Huffman code is to verify that the code is prefix-free, meaning no code word is a prefix of any other code word. A prefix-free code is always optimal.

A third way to check the optimality of Huffman code is by comparing the average length of the code to the entropy of the source. In all cases, if the code is not optimal, you can use the standard algorithm for constructing a Huffman code to generate a new, optimal code.

In this source code, I am using Kraft inequality check for proving optimality of Huffman code.

Source Code

```
import math
from collections import Counter
import Huffman

# Input message
message = "aaabbbbccccdddee"
freq = dict(Counter(message))
length = len(message)

# Generate Huffman codes
huffman_code = Huffman.codebook(freq.items())
huffman_avg_length = 0
for char, code in huffman_code.items():
    huffman_avg_length += len(code) * (freq[char] / length)

# Calculate entropy
H = -sum((count / length) * math.log2(count / length) for count in freq.values())
print("Huffman Code:", huffman_code)
print(f"Average Huffman Code Length: {huffman_avg_length:.2f} bits")
print(f"Entropy: {H:.2f} bits")
print("Huffman code is optimal" if huffman_avg_length >= H else "Code is not optimal")
```

Output:

```
Huffman Code: {'a': '101', 'b': '01', 'c': '11', 'd': '00', 'e': '100'}
Average Huffman Code Length: 2.25 bits
Entropy: 2.23 bits
Huffman code is optimal
```

Experiment No: 07**Experiment Name:** Numerical Based on Conditional and Joint Entropy.

Example 2.2.1: Let (X, Y) have the following joint distribution:

Y \ 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

The marginal distribution of X is $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$,and the marginal distribution of Y is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, and hence:

$$H(X) = \frac{7}{4} \text{ bits}$$

$$H(Y) = 2 \text{ bits}$$

Also,

$$\begin{aligned}
 H(X|Y) &= \sum_{i=1}^4 P(Y = i) H(X|Y = i) \\
 &= \frac{1}{4} H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4} H\left(\frac{1}{4}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4} H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) + \frac{1}{4} H(1, 0, 0, 0) \\
 &= \frac{1}{4} \times \frac{7}{4} + \frac{1}{4} \times \frac{7}{4} + \frac{1}{4} \times 2 + \frac{1}{4} \times 0 \\
 &= \frac{11}{8} \text{ bits}
 \end{aligned}$$

Similarly, $H(X|Y) = \frac{13}{8}$ bits and $H(X,Y) = \frac{27}{8}$ bits.**Theory:****Definition:** The entropy $H(X)$ of a discrete random variable X is defined by:

$$H(X) = - \sum_{x \in X} P(x) \log P(x)$$

Definition: Given $(X,Y) \sim P(x,y)$, the conditional entropy $H(Y|X)$ is defined as:

$$\begin{aligned}
 H(X|Y) &= \sum_{x \in X} P(x) H(Y|X = x) \\
 &= - \sum_{x \in X} P(x) \sum_{y \in Y} P(y|x) \log P(y|x) \\
 &= - \sum_{x \in X} \sum_{y \in Y} P(x, y) \log P(y|x) \\
 &= -E \log P(Y|X)
 \end{aligned}$$

Source Code:

```
# given
import math
matrix = [
    [1 / 8, 1 / 16, 1 / 32, 1 / 32],
```

```

[1 / 16, 1 / 8, 1 / 32, 1 / 32],
[1 / 16, 1 / 16, 1 / 16, 1 / 16],
[1 / 4, 0, 0, 0],
]
marginal_x = []
for i in range(len(matrix[0])):
    marginal_x.append(sum(matrix[j][i] for j in range(len(matrix))))
marginal_y = []
for i in range(len(matrix)):
    marginal_y.append(sum(matrix[i][j] for j in range(len(matrix[0]))))

# H(x)
def entropy(marginal_var):
    H = 0
    for x in marginal_var:
        if x == 0:
            continue
        H += -(x * math.log2(x))
    return H
H_x = entropy(marginal_x)
H_y = entropy(marginal_y)

# H(x/y)
H_xy = 0
for i in range(len(matrix)):
    tmp = [(1 / marginal_y[i]) * matrix[i][j] for j in range(len(matrix[0]))]
    H_xy += entropy(tmp) * marginal_y[i]

# H(y/x)
H_yx = 0
for i in range(len(matrix[0])):
    tmp = [(1 / marginal_x[i]) * matrix[j][i] for j in range(len(matrix))]
    H_yx += entropy(tmp) * marginal_x[i]

print("Conditional Entropy H(x|y): ", H_xy)
print("Conditional Entropy H(y|x): ", H_yx)
# H(x,y)
H_of_xy = H_x + H_yx
print("Joint Entropy H(x,y): ", H_of_xy)
# I(x,y)
I_of_xy = H_y - H_yx
print("Mutual Information: ", I_of_xy)

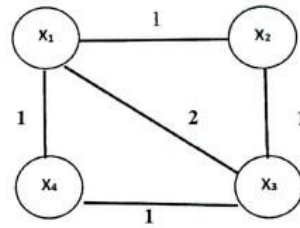
```

Output:

```

Conditional Entropy H(x|y): 1.375
Conditional Entropy H(y|x): 1.625
Joint Entropy H(x,y): 3.375
Mutual Information: 0.375

```

Experiment No: 08**Experiment Name:** Explain entropy rate of a random walk on a weighted graph.**Theory:**

The entropy of a stochastic process $\{X_i\}$ is defined by,

$$H(X) = \lim_{n \rightarrow \infty} 1/n H(X_1, X_2, \dots, X_n)$$

when the limit exists. We now consider some simple examples of stochastic processes and their corresponding entropy rates.

1. Typewriter.

Consider the case of a typewriter that has m equally likely output letters. The typewriter can produce m^n sequence of length n , all of them equally likely. Hence

$$H(X_1, X_2, \dots, X_n) = \log \frac{m^n}{1} \text{ and the entropy rate is}$$

$$H(X) = \log m \text{ bits per symbol.}$$

2. X_1, X_2, \dots are iid random variables. Then

$$H(X) = \lim_{n \rightarrow \infty} H(X_1, X_2, \dots, X_n)/n = \lim_{n \rightarrow \infty} n H(X_1)/n = H(X_1)$$

Sequence of independent but not identically distributed random variables.

In this case,

$$H(X_1, X_2, \dots, X_n) = \sum H(X_i)$$

But the $H(X_i)$'s are all not equal. We can choose a sequence of distributions on X_1, X_2, \dots such that the limit of $1/n \sum H(X_i)$ does not exist.

Consider a graph with m nodes labeled $\{1, 2, \dots, m\}$ with weight $W_{ij} \geq 0$ on the edge joining node i to j . (The graph is assumed to be undirected, so $W_{ij} = W_{ji}$. We set $W_{ij} = 0$ if there is no edge joining nodes i and j .) A particle walks randomly from node to node in this graph. The random walk $\{X_n\}$, $X_n \in \{1, 2, \dots, m\}$, is a sequence of vertices of the graph. Given $X_n = i$, the next vertex j is chosen from among the nodes connected to node i with a probability proportional to the weight of the edge connecting i to j . Thus $P_{ij} = W_{ij} / \sum_k W_{ik}$.

In this case, the stationary distribution has a surprisingly simple form, which we will guess and verify. The stationary distribution for this Markov chain assigns probability to node i proportional to the total weight of the edges emanating from node i .

Let,

$$W_i = \sum_j W_{ij}$$

be the total weight of edges emanating from node i , and let

$$W = \sum_{i,j} W_{ij}$$

be the sum of the weights of all the edges. Then $\sum_i W_i = 2W$. We now guess that the stationary distribution is

$$\mu_i = W_i / 2W$$

We verify that this is the stationary distribution by checking that $\mu_p = \mu$. Here,

$$\begin{aligned} \sum_i \mu_i P_{ij} &= \sum_i \frac{W_i W_{ij}}{2W W_i} \\ &= \sum_i \frac{1}{2W} W_{ij} \\ &= \frac{W_j}{2W} \\ &= \mu_j \end{aligned}$$

Thus, the stationary probability of state i is proportional to the weight of edges emanating from node i . This stationary distribution has an interesting property of locality: it depends only on the total weight and the weight of edges connected to the node and hence does not change. If the weights in some other part of the graph are changed while keeping the total weight constant, we can now calculate the entropy rate as

$$\begin{aligned} H(X) &= H(X_2 | X_1) \\ &= - \sum_i \frac{W_i}{2W} \sum_j \frac{W_{ij}}{W_i} \log \frac{W_{ij}}{W_i} \\ &= - \sum_i \sum_j \frac{W_i}{2W} \log \frac{W_{ij}}{W_i} \\ &= - \sum_i \sum_j \frac{W_{ij}}{2W} \log \frac{W_{ij}}{W_i} + \sum_i \sum_j \frac{W_{ij}}{2W} \log \frac{W_{ij}}{2W} \\ &= H(\dots \frac{W_{ij}}{2W} \dots) - H(\dots \frac{W_i}{2W}) \end{aligned}$$

If all the edges have equal weight, the stationary distribution puts weight $E_i / 2E$ on node i , where E_i is the number of edges emanating from node i and E is the total number of edges in the graph. In this case, the entropy rate of the random walk is

$$H(X) = \log 2E - H(E_1/2E, E_2/2E, \dots, E_m/2E)$$

This answer for the entropy rate is so simple that it is almost misleading. Apparently, the entropy rate, which is the average transition entropy, depends only on the entropy of the stationary distribution and the total number of edges.

Source Code:

```
import math
from collections import defaultdict
# given
g = defaultdict(list)
xij = [[1, 1, 2], [1, 1], [1, 2, 1], [1, 1]]

def makeGraph(li):
    for node in range(len(li)):
```

```

    for x in li[node]:
        g[node].append(x)

def entropy(li):
    H = 0
    for x in li:
        if x == 0:
            continue
        H += -(x * math.log2(x))
    return H
# make graph
makeGraph(xij)
wi = []
for node in range(len(g)):
    wi.append(sum(g[node]))

w = sum(wi) / 2
ui = [weight / (2 * w) for weight in wi]
H_wi_div_2w = entropy(ui)

#  $H(wij/2*w) = H(g[]/2*w)$ 
wij_div_2w_list = []
for i in range(len(g)):
    wij_div_2w_list += [weight / (2 * w) for weight in g[i]]

#  $H(wij/2*w) = H(wij\_div\_2w\_list)$ 
H_wij_div_2w = entropy(wij_div_2w_list)

# finally the entropy rate
#  $H(x) = H(wij/2w) - H(wi/2w)$ 
H_x = H_wij_div_2w - H_wi_div_2w
print('Entropy Rate: %.2f % H_x)'

```

Output:

Entropy Rate: 1.33