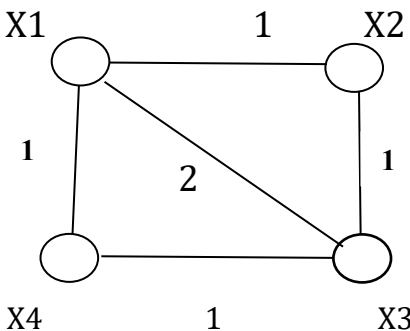


Index

SL NO	Experiment Name	Page NO																									
1	Write a program to implement Huffman code using symbols with their corresponding probabilities.	2-4																									
2	Write a program to simulate convolutional coding based on their encoder structure.	5-8																									
3	Write a program to implement Lempel-Ziv code.	9-10																									
4	Write a program to implement Hamming code.	11-13																									
5	A binary symmetric channel has the following noise matrix with probability, $P(x) = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}$ Now find the Channel Capacity C.	14-15																									
6	Write a program to check the optimality of Huffman code.	16-18																									
7	Explain entropy rate of a random walk on a weighted graph 	19-21																									
8	Write a program to find conditional entropy and joint entropy and mutual information based on the following matrix. <table><tr><th>X \ Y</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>	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	22-24
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																							

Experiment No:01

Experiment Name: Write a program to implement Huffman code using symbols with their corresponding probabilities.

Theory:

Huffman Coding is an algorithm used for data compression that assigns variable-length binary codes to symbols based on their frequencies or probabilities of occurrence. This technique was introduced by David A. Huffman in 1952 as a way to efficiently compress data by minimizing the number of bits needed to represent it.

Applications of Huffman Coding:

- File Compression (ZIP files, GZIP).
- Multimedia Encoding (JPEG, MP3).
- Transmission of Data over bandwidth-limited communication channels.

An optimal prefix free code is a prefix free code that minimizes the expected code word length.

$$L = \sum p(x_i)l_i$$

Over all prefix free codes

$$L = P_1L_1 + P_2L_2 + P_3L_3 + \dots + P_NL_N$$

Algorithm for Huffman Coding:

Step 01: The source symbols are arranged in the order of decreasing probability.

Step 02: Then two symbols of lowest probability are assigned '0' and '1'.

Step 03: Then combine last two symbols and move the combined symbols as high as possible.

Step 04: Repeat the above step until we are left with the final list of source symbols of only two for which '0' and '1' are assigned.

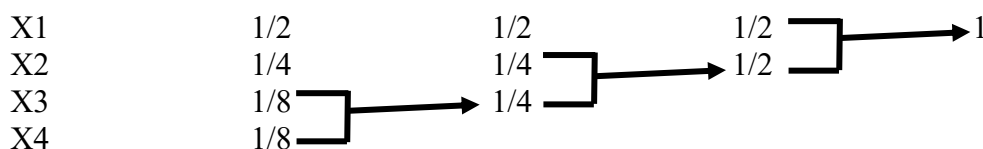
Example:

The random variable X is drawn from $X = \{x_1, x_2, x_3, x_4\}$ with the probabilities.

X	X1	X2	X3	X4
P(x)	1/2	1/4	1/8	1/8

Find the average path length, codeword, codeword length and construct the tree ?

Solve:



Symbol	Probabilities	Code word
x1	1/2	0
X2	1/4	10
X3	1/8	110
X4	1/8	111

$$\begin{aligned} \text{Average length } L &= \sum p(x_i)l_i \\ &= \frac{1}{2} * 1 + \frac{1}{2} + 2 * \frac{1}{8} * 3 \\ &= 1 + \frac{3}{4} \end{aligned}$$

$$= 1.75$$

So compare we derive the entropy

$$H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) = 1.75$$

The efficiency is 100%.

Python Code:

```
import heapq
# Define a class for the nodes of the Huffman Tree
class Node:
    def __init__(self, symbol, probability):
        self.symbol = symbol
        self.probability = probability
        self.left = None
        self.right = None
    # Overload less than operator for priority queue
    def __lt__(self, other):
        return self.probability < other.probability
# Function to build the Huffman Tree
def build_huffman_tree(symbols_with_probs):
    heap = [Node(symbol, prob) for symbol, prob in symbols_with_probs]
    heapq.heapify(heap)
    while len(heap) > 1:
        # Pop two nodes with lowest probabilities
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        # Merge nodes and create new node
        merged = Node(None, left.probability + right.probability)
        merged.left = left
        merged.right = right
        # Add the merged node back to the heap
        heapq.heappush(heap, merged)
    # The remaining node is the root of the Huffman Tree
    return heap[0]
# Function to generate Huffman Codes for each symbol
def generate_codes(node, prefix="", codebook={}):
    if node is not None:
        # Leaf node, assign code to symbol
        if node.symbol is not None:
            codebook[node.symbol] = prefix
        else:
            # Traverse left with '0' and right with '1'
            generate_codes(node.left, prefix + "0", codebook)
            generate_codes(node.right, prefix + "1", codebook)
    return codebook

# Example symbols with their probabilities
```

```

symbols_with_probs = [
    ("A", 0.4),
    ("B", 0.2),
    ("C", 0.2),
    ("D", 0.1),
    ("E", 0.1)
]
# Build the Huffman Tree
root = build_huffman_tree(symbols_with_probs)
# Generate Huffman Codes
huffman_codes = generate_codes(root)
# Print the generated codes
print("Symbol\tProbability\tHuffman Code")
for symbol, prob in symbols_with_probs:
    print(f"{symbol}\t{prob}\t{huffman_codes[symbol]}")

```

Output:

Symbol	Probability	Huffman Code
A	0.4	11
B	0.2	01
C	0.2	00
D	0.1	101
E	0.1	100

Experiment No:02

Experiment Name: Write a program to simulate convolutional coding based on their encoder structure.

Theory:

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 (U_k s). 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.

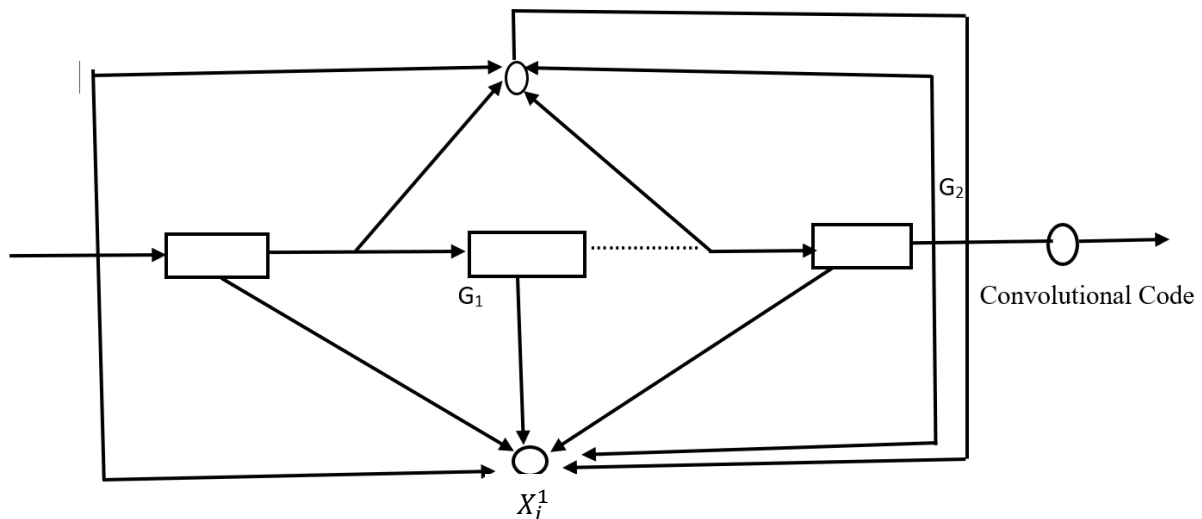


Figure 1: Convolutional Code

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

Mathematical problem.

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,

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

Then for successive i, we get:

$$x_1^{(2)} = g_0^2 m_1 + g_1^2 m_0 = 1x0 + 0x1 = 0+0 = 0$$

$$x_2^{(2)} = g_0^2 m_2 + g_1^2 m_1 + g_2^2 m_0 = 1x0 + 0x0 + 1x1 = 0+1 = 1\%2 = 1$$

$$x_3^{(2)} = g_0^2 m_3 + g_1^2 m_2 + g_2^2 m_1 = 1x1 + 0x0 + 1x0 = 1+0+0 = 1\%2 = 1$$

$$x_4^{(2)} = g_0^2 m_4 + g_1^2 m_3 + g_2^2 m_2 = 1x1 + 0x1 + 1x0 = 1+0+0 = 1\%2 = 1$$

$$x_5^{(2)} = g_1^2 m_4 + g_2^2 m_3 = 0x1 + 1x1 = 1+0 = 1\%2 = 1$$

$$x_6^{(2)} = g_2^2 m_3 = 1x1 = 1\%2 = 1$$

$$X_i^{(2)} = 1011111$$

$$X_i = 11101111010111$$

Python Code:

```
import numpy as np

# Define generator polynomials and message
g0 = [1, 1, 1, 1]
g1 = [1, 1, 0, 1]
message = [1, 0, 1, 0, 1]
K = 4
n = 2

def encode(msg, generators):
    # Initialize an empty list to hold encoded results for each generator
    v = []

    # Loop over each generator polynomial
    for g in generators:
        # Perform polynomial multiplication (convolution) and convert to list
        res = list(np.polyld(g) * np.polyld(msg))
        # Keep results modulo 2
        res = [int(x) % 2 for x in res]
        v.append(res)

    # Find the length of the longest encoded result
    listMax = max(len(l) for l in v)

    # Zero-pad all encoded results to the length of the longest result
    for i in range(n):
        if len(v[i]) < listMax:
            v[i] = [0] * (listMax - len(v[i])) + v[i]

    # Interleave bits from each encoded result to form the final encoded message
    res = []
```

```

    for i in range(listMax):
        for j in range(n):
            res.append(v[j][i])

    return res

# Define the list of generator polynomials
generators = [g0, g1]
# Encode the message
encoded_message = encode(message, 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: Write a program to implement Lempel-Ziv code.

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:

AABABBBABAABABBBABBABB

Let us assumed A=0 & B=1

Lets make a position for this sequence.

Position	1	2	3	4	5	6	7	8	9
Sequence	A	AB	ABB	B	ABA	ABAB	BB	ABBA	BB
Numerical representation	A	1B	2B	B	2A	5B	4B	3A	7
Binary coded block.	0	11	101	1	101	1011	1001	110	111

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(f'N.representatin: {res}')
print("Encoded: ", final_res)

```

Output:

```

N. representatin: ['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 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 \rightarrow P1 = 0 (since 1+1+0 is even).
 - Parity bit at position 2 (P2): Covers bits 2, 3, 6, 7: 0, 1, 1 \rightarrow P2 = 0 (since 0+1+1 is even).
 - Parity bit at position 4 (P4): Covers bits 4, 5, 6, 7: 0, 1, 1, 1 \rightarrow 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 \rightarrow Sum = 0+1+0+1=2 (even), so Syndrome P1 = 0.

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

Syndrome for P4 (positions 4, 5, 6, 7): Bits: 0, 0, 1, 1 \rightarrow 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 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}')
# 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))}')

```

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

```

Experiment No:05

Experiment Name: A binary symmetric channel has the following noise matrix with probability, $P(x) = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}$ 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
```

```
# given
```

```
matrix = [[2 / 3, 1 / 3], [1 / 3, 2 / 3]]
```

```

print("Symmetric matrix is:")
for i in range(0, 2):
    for j in range(0, 2):
        print("%.2f" % matrix[i][j], 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("Conditional probability H(Y/X) is = %.3f" % Hp, "bits/msg symbol")

# Now calculate channel capacity using formula C = 1 - H(Y/X)
C = 1 - Hp
print("Channel Capacity is = %.3f" % C, "bits/msg symbol")

```

Output:

```

Symmetric matrix is:
0.67 0.33
0.33 0.67
Conditional probability H(Y/X) is = 0.918 bits/msg symbol
Channel Capacity is = 0.082 bits/msg symbol

```

Experiment No:06

Experiment Name: Write a program to check the optimality of Huffman code.

Theory:

Huffman Coding is an algorithm used for data compression that assigns variable-length binary codes to symbols based on their frequencies or probabilities of occurrence. This technique was introduced by David A. Huffman in 1952 as a way to efficiently compress data by minimizing the number of bits needed to represent it.

Applications of Huffman Coding:

- File Compression (ZIP files, GZIP).
- Multimedia Encoding (JPEG, MP3).
- Transmission of Data over bandwidth-limited communication channels.

An optimal prefix free code is a prefix free code that minimizes the expected code word length.

$$L = \sum p(x_i)l_i$$

Over all prefix free codes

$$L = P_1L_1 + P_2L_2 + P_3L_3 + \dots + P_NL_N$$

Compare Expected Length with Entropy:

- For an optimal Huffman code, the expected length L should satisfy:

$$H(X) \leq L < H(X) + 1$$

If this condition is met, the Huffman code is optimal.

Algorithm for Huffman Coding:

Step 01: The source symbols are arranged in the order of decreasing probability.

Step 02: Then two symbols of lowest probability are assigned '0' and '1'.

Step 03: Then combine last two symbols and move the combined symbols as high as possible.

Step 04: Repeat the above step until we are left with the final list of source symbols of only two for which '0' and '1' are assigned.

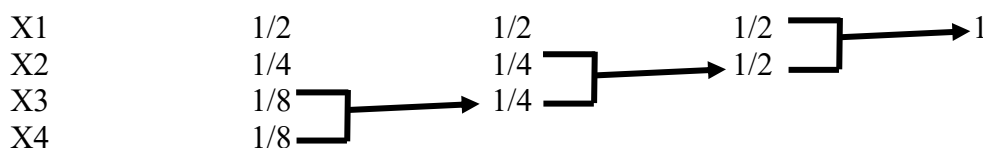
Example:

The random variable X is drawn from $X = \{x_1, x_2, x_3, x_4\}$ with the probabilities.

X	X1	X2	X3	X4
P(x)	1/2	1/4	1/8	1/8

Find the average path length, codeword, codeword length and construct the tree ?

Solve:



Symbol	Probabilities	Code word
x1	1/2	1
X2	1/4	01
X3	1/8	110
X4	1/8	111

$$\begin{aligned}
 \text{Average length } L &= \sum p(x_i)l_i \\
 &= \frac{1}{2} * 1 + \frac{1}{2} + 2 * \frac{1}{8} * 3 \\
 &= 1 + \frac{3}{4} \\
 &= 1.75
 \end{aligned}$$

So compare we derive the entropy

$$H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) = 1.75$$

Compare Expected Length with Entropy:

For an optimal Huffman code, the expected length L should satisfy:

$$H(X) \leq L < H(X) + 1$$

If this condition is satisfied, the Huffman code is optimal.

Python Code:

```

import heapq
import math
from collections import 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]
def compute_huffman_avg_length(freq, tree, length):
    huffman_avg_length = 0
    for pair in tree[1:]:
        huffman_avg_length += (len(pair[1]) * (freq[pair[0]] / length))

```

```

    return huffman_avg_length
def entropy(freq, length):
    H = 0
    P = [fre / length for _, fre in freq.items()]
    for x in P:
        H += -(x * math.log2(x))
    return H
message = "aaabbbbbccccccdddee"
freq = calculate_frequency(message)
heap = build_heap(freq)
tree = build_tree(heap)
# tree=[20, ['D', '0'], ['B', '01'], ['E', '100'], ['A', '101'], ['C', '11']] not optimal
huffman_avg_length = compute_huffman_avg_length(freq, tree, len(message))
H = entropy(freq, len(message))
print("Huffman : %.2f bits" % huffman_avg_length)
print('Entropy : %.2f bits' % H)
if huffman_avg_length >= H:
    print("Huffman code is optimal")
else:
    print("Code is not optimal")

```

Output:

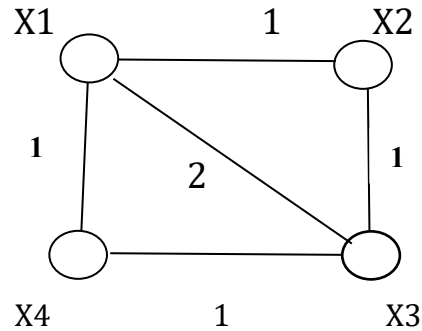
```

Huffman : 2.25 bits
Entropy : 2.23 bits
Huffman code is optimal

```

Experiment No:07

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.

Python 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]))
# we know
# summation(wi)=2w
w = sum(wi) / 2
# the stationary distribution is
# ui=(wi)/2w
ui = [weight / (2 * w) for weight in wi]

# H((wi)/2w)=H(ui)
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

Experiment No:08

Experiment Name: Write a program to find conditional entropy and join entropy and mutual information based on the following matrix.

Theory:

The entropy $H(X)$ of a discrete random variable X is defined by:

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

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

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

Mutual information: The mutual information $I(X; Y)$ is the relative entropy between the joint distribution and the product distribution $p(x)p(y)$:

$$\begin{aligned} I(X; Y) &= \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \\ &= D(p(x, y) \| p(x)p(y)) \\ &= E_{p(x, y)} \log \frac{p(x, y)}{p(x)p(y)} \end{aligned}$$

Example

Find conditional entropy and join entropy and mutual information based on the following matrix.

$\begin{matrix} X \\ Y \end{matrix}$	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 $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$.

Entropy, $H(X) = \frac{7}{4}$ bits and $H(Y) = 2$ bits.

Conditional Entropy

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

Similarly,

$$H(Y/X) = \sum_{i=1}^4 P(x = i) H(Y/X = i)$$

$$\begin{aligned}
&= \frac{1}{2} H\left(\frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{2}\right) + \frac{1}{4} H\left(\frac{1}{4}, \frac{1}{2}, \frac{1}{4}, 0\right) + \frac{1}{8} H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{2}, 0\right) + \frac{1}{8} H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{2}, 0\right) \\
&= \frac{1}{2} \times \frac{14}{8} + \frac{1}{4} \times \frac{6}{4} + \frac{1}{8} \times \frac{6}{4} + \frac{1}{8} \times \frac{6}{4} \\
&= \frac{13}{8} \text{ bits}
\end{aligned}$$

Joint Entropy

$$H(X, Y) = H(X) + H(Y/X) = \frac{7}{4} + \frac{13}{8} = \frac{27}{8} \text{ bits}$$

Mutual Information,

$$I(X; Y) = H(X) - H\left(\frac{X}{Y}\right) = \frac{7}{4} - \frac{11}{8} = \frac{3}{8} \text{ bits}$$

$$\text{Or, } I(X; Y) = H(Y) - H\left(\frac{Y}{X}\right) = 2 - \frac{13}{8} = \frac{3}{8} \text{ bits}$$

Python Code:

```

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]
]

# the marginal distribution of x
marginal_x = []
for i in range(len(matrix[0])):
    marginal_x.append(sum(matrix[j][i] for j in range(len(matrix))))

# the marginal distribution of y
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)

# conditional entropy
# 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)

# Joint entropy
# H(x,y)
H_of_xy = H_x + H_yx
print('Joint Entropy H(x,y): ', H_of_xy)

# Mutual Information
# 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

```