Experiment Number: 01

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

Theory: Huffman coding is a technique used to compress data by reducing its size. The concept involves assigning shorter codes to characters that appear more frequently and longer codes to those that appear less often. This approach minimizes the total length of the encoded data. Huffman coding accomplishes this by constructing a binary tree in which each leaf node corresponds to a character, and the path from the root to each leaf defines that character's code.

The method begins by calculating the frequency of each character within the data. The two characters with the lowest frequencies are then combined to form a new node, and this process is repeated until only a single node, the root of the tree, remains. This final tree structure is used to create the Huffman codes, which are then applied to encode the original data, leading to a more compact representation.

Algorithm of Huffman Codde:

1. **Count frequencies:** Start by determining how often each character appears in the text.
2. **Build a priority queue:** Create a priority queue (or min-heap) where each entry represents a character and its frequency.
3. **Build the tree:** While the queue has more than one node:
    - Remove the two nodes with the lowest frequencies.
    - Create a new node with these two nodes as children, setting its frequency to the sum of their frequencies.
    - Insert this new combined node back into the queue.
4. **Generate codes:** Traverse the tree from the root to each leaf node, assigning '0' for left branches and '1' for right branches. The path from the root to each leaf gives the Huffman code for that character.
5. **Encode the text:** Replace each character in the text with its Huffman code.
6. **Decode the text (optional):** Starting at the root of the Huffman tree, follow the path dictated by each bit in the encoded text until reaching a leaf, then output the corresponding character.

Example:

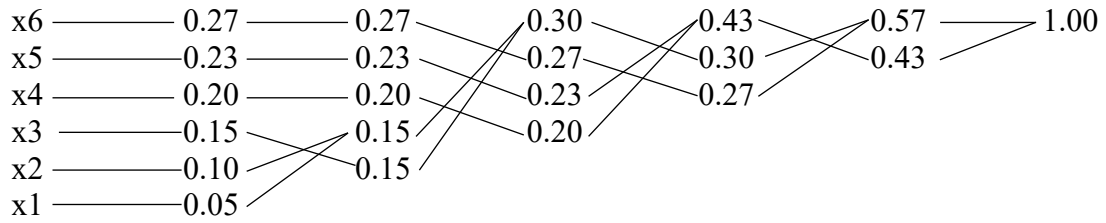
X	X1	X2	X3	X4	X5	X6
P(x)	0.05	0.10	0.15	0.20	0.23	0.27

We start with symbols and their probabilities: X1 (0.05), X2 (0.10), X3 (0.15), X4 (0.20), X5 (0.23), and X6 (0.27). First, we arrange the symbols in ascending order of their probabilities.

Next, we combine the two symbols with the lowest probabilities (X1 and X2, with probabilities 0.05 and 0.10) to create a new symbol with a combined probability of 0.15. This process is repeated, merging the next smallest probabilities until only one symbol remains, representing the entire set. After each combination, we reorder the symbols.

The steps continue as follows: we merge X1+X2 and X3 to form a symbol with probability 0.30, then combine X4 and X5 to create a symbol with probability 0.43. Next, X6 is merged with X1+X2+X3 for a combined probability of 0.57. Finally, we combine the last two groups, X4+X5 and X6+X1+X2+X3, to complete a binary tree structure.

This gives us the optimal Huffman codes for each symbol.



Python code:

```
import heapq

class HuffmanNode:

    def __init__(self, symbol, freq):

        self.symbol = symbol

        self.freq = freq

        self.left = None

        self.right = None

    def __lt__(self, other):

        return self.freq < other.freq

def build_huffman_tree(symbols):

    heap = [HuffmanNode(symbol, freq) for symbol, freq in symbols]

    heapq.heapify(heap)

    while len(heap) > 1:

        left = heapq.heappop(heap)

        right = heapq.heappop(heap)

        merged = HuffmanNode(None, left.freq + right.freq)

        merged.left = left

        merged.right = right

        heapq.heappush(heap, merged)

    return heap[0]
```

```
def generate_huffman_codes(node, prefix="", codebook={}):
    if node.symbol is not None:
        codebook[node.symbol] = prefix
    else:
        generate_huffman_codes(node.left, prefix + "0", codebook)
        generate_huffman_codes(node.right, prefix + "1", codebook)
    return codebook
symbols = [("A", 0.05), ("B", 0.1), ("C", 0.15), ("D", 0.2),("E", 0.23),("F", 0.27)]
root = build_huffman_tree(symbols)
huffman_codes = generate_huffman_codes(root)
print("Huffman Codes for each symbol:")
for symbol, code in huffman_codes.items():
    print(f"{symbol}: {code}")
```

Output:

```
Huffman Codes for each symbol:
D: 00
E: 01
F: 10
C: 110
A: 1110
B: 1111
```

Experiment Number: 02

Name of the Experiment: To Write a program to implement Convolutional Code.

Theory: Convolutional coding is an error correction method used in digital communication, particularly effective for real-time data streams like audio and video signals. Unlike block codes, which handle data in fixed-size blocks, convolutional codes operate on a continuous data stream, making them suitable for applications where data is transmitted in real-time.

This method uses a shift register that holds a sequence of input bits, which then pass through logic gates, known as generators. Each generator produces an output bit based on the current input bit and previous bits stored in the shift register, meaning that each input bit impacts multiple output bits. This redundancy enables the detection and correction of errors.

To decode convolutional codes, the Viterbi algorithm is often used. This algorithm examines the most likely paths in the received data to reconstruct the original message. Convolutional coding is highly valued for its ability to improve data reliability in noisy communication environments, particularly in scenarios where continuous, real-time processing is essential.

Algorithm of Convolutional Code:

1. **Input message bits:** Begin with the sequence of bits that need encoding.
2. **Initialize shift registers:** Set up shift registers to store a series of bits (like a sliding window) that will be used to produce output bits.
3. **Shift bits:** For each new input bit, add it to the register, shifting older bits one position to the right.
4. **Apply generator functions:** Each generator function taps into specific positions in the shift register and applies an XOR operation to the selected bits.
5. **Generate output bits:** For each input bit, the generator functions produce a series of encoded bits as output.
6. **Repeat for each bit:** Continue shifting the register and generating encoded bits until all input bits are processed.

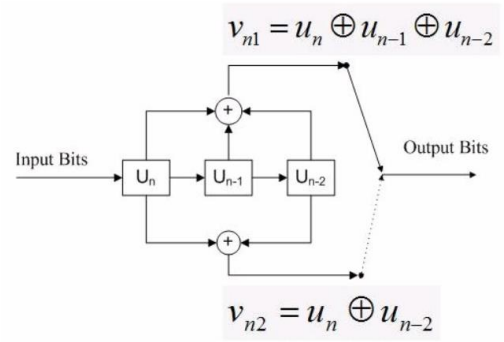


Figure 1: Convolutional Code.

Example:

In this convolutional encoding example, the input bits are processed using a shift register and generator polynomials to create encoded output bits. Let's walk through the encoding process for the input data 1011.

Initially, the shift register starts empty, represented as [0, 0, 0]. The generator polynomials are G1 = 111 and G2 = 101, meaning the output bits are produced by XORing the current input bit with the previous bits in the register.

1. For the first input bit, **1**:
    - The register shifts to [1, 0, 0].
    - The first output bit (y1) is the XOR of all three bits: 1 XOR 0 XOR 0 = 1.
    - The second output bit (y2) is the XOR of the first and third bits: 1 XOR 0 = 1.
    - Thus, the encoded output for this bit is **1 1**.
2. For the second input bit, **0**:
    - The register shifts to [0, 1, 0].
    - The first output bit (y1) is 0 XOR 1 XOR 0 = 1.
    - The second output bit (y2) is 0 XOR 0 = 0.
    - The encoded output for this bit is **1 0**.
3. For the third input bit, **1**:
    - The register shifts to [1, 0, 1].
    - The first output bit (y1) is 1 XOR 0 XOR 1 = 0.
    - The second output bit (y2) is 1 XOR 1 = 0.
    - The encoded output for this bit is **0 0**.
4. For the fourth input bit, **1**:
    - The register shifts to [1, 1, 0].
    - The first output bit (y1) is 1 XOR 1 XOR 0 = 0.
    - The second output bit (y2) is 1 XOR 0 = 1.
    - The encoded output for this bit is **0 1**.

Therefore, the final encoded data for the input 1011 is **1111011001**. This encoding method ensures each input bit is transformed into two output bits, adding redundancy for error detection and correction during transmission.

Code:

```
# Define generator polynomials (rate 1/2 code)
G1 = [1, 1, 1]  # Polynomial: 111
G2 = [1, 0, 1]  # Polynomial: 101
# Encoding function
def encode_data(data):
    state = [0, 0, 0]  # Initialize shift register state with three bits
    encoded_data = []
    for bit in data:
        # Shift the register and add the new bit at the start
        state = [bit] + state[:-1]
        # Calculate output bits by XORing the state with the generator polynomials
        out1 = state[0] ^ state[1] ^ state[2]  # XOR for G1
        out2 = state[0] ^ state[2]             # XOR for G2

        # Append the output bits to the encoded data
        encoded_data.append(out1)
        encoded_data.append(out2)
    return encoded_data
# Main program
input_data = list(map(int, input("Enter binary data (e.g., 1011): ")))
# Encode the data
encoded_data = encode_data(input_data)
print("Encoded Data:", encoded_data)
```

Output:

Encoded Data: [1, 1, 1, 0, 0, 0, 0, 1]

Experiment Number: 03

Name of the Experiment: To Write a program to explain and implement Lampel-Ziv Code.

Theory: Lempel-Ziv coding is a lossless data compression technique that reduces data size by replacing repeated sequences with references to their first occurrence. The core idea is to identify patterns within the data and substitute each repeated instance with a shorter representation, thereby compressing the data.

The algorithm scans the data while constructing a dictionary of sequences it has already encountered. For each new data segment, it checks if the sequence exists in the dictionary. If found, the algorithm outputs a reference to the previous occurrence instead of the full sequence. If the sequence is new, it adds it to the dictionary and continues scanning.

Lempel-Ziv coding is especially useful for compressing text or data containing repetitive patterns. It underpins well-known compression methods like LZ77 and LZW, which are widely used in formats such as GIF and ZIP files.

Algorithm:

1. **Start with the input text:** Begin with the sequence of characters or symbols that needs compression.
2. **Search for the longest matching substring:** Find the longest substring in the input that matches a previously seen sequence in the processed part of the text.
3. **Encode the match:** When a match is found, represent it as a pair (position, length):
    - **Position:** The starting position of the matching substring in the earlier text.
    - **Length:** The length of this matched substring.
4. **Add the next unmatched character:** After encoding the match, append the next unmatched character to the output.
5. **Repeat:** Continue looking for matches and adding unmatched characters until the entire input is processed.
6. **Output:** The compressed data consists of encoded pairs (position, length) along with any remaining unmatched characters.

Example: To apply Lempel-Ziv encoding to the input bit sequence "010011010," we'll break down the sequence, assign each new subsequence a unique representation, and encode it as a binary block. Here's the step-by-step process:

1. **First bit:** "0"; This is the first occurrence, so it's represented as 0 and encoded as "0000."
2. **Second bit:** "1"; This bit hasn't been encountered before, so it's represented as 1 and encoded as "0001."
3. **Next subsequence:** "00"; This is a new subsequence, so it's represented by 10 and encoded as "0010."

4. **Next subsequence:** "01"; This is also new, so it's represented by 11 and encoded as "0011."
5. **Next subsequence:** "011"; This has not been seen before, so it's represented by 41 and encoded as "1001."
6. **Next subsequence:** "10"; Another new subsequence, so it's represented by 20 and encoded as "0100."
7. **Final subsequence:** "010"; This hasn't appeared before, so it's represented by 40 and encoded as "1000."

The Lempel-Ziv encoded output for the sequence "010011010" is therefore a series of binary blocks representing each unique subsequence in the order they appeared: **0000, 0001, 0010, 0011, 1001, 0100, 1000**

Table: Lempel-Ziv encoding table.

Number Position	1	2	3	4	5	6	7
Subsequence	0	1	00	01	011	10	010
Number Representation	0	1	10	11	41	20	40
Binary Encoded Block	0000	0001	0010	0011	1001	0100	1000

Code:

```
def lz78_compress(data):
    dictionary = {}
    compressed_data = []
    current_string = ""
    dict_size = 1  # Start dictionary indexing from 1
    print("Num Pos\tSubsequence\tNum Represent")
    for i, char in enumerate(data):
        current_string += char
        if current_string not in dictionary:
            # Add the current string to the dictionary
            dictionary[current_string] = dict_size
            previous_index = dictionary.get(current_string[:-1], 0)
            dict_size += 1
            # Record the compressed pair (index of prefix, next character)
            compressed_data.append((previous_index, char))
            # Print the details
                   print(f"{dict_size-1}\t\t{current_string}\t\t({previous_index},
'{char}')")
```

```
            # Reset current_string for the next sequence
            current_string = ""
    return compressed_data
def lz78_decompress(compressed_data):
    dictionary = {0: ""}  # Initialize dictionary with an empty prefix at index 0
    decompressed_data = []
    dict_size = 1
    for index, char in compressed_data:
        entry = dictionary[index] + char
        decompressed_data.append(entry)
        # Add new entry to the dictionary
        dictionary[dict_size] = entry
        dict_size += 1
    return ''.join(decompressed_data)
# Main program with user input
input_data = input("Enter data (binary or characters): ")
# Compress
compressed = lz78_compress(input_data)
print("\nCompressed:", compressed)
# Decompress
decompressed = lz78_decompress(compressed)
print("Decompressed:", decompressed)
# Verify correctness
if decompressed == input_data:
    print("Decompression successful! The data matches the original input.")
else:
    print("Decompression failed! The data does not match the original input.")
```

Output:

```
Enter data (binary or characters): 01100111110

Num Pos Subsequence     Num Represent

1               0               (0, '0')

2               1               (0, '1')

3               10              (2, '0')

4               01              (1, '1')

5               11              (2, '1')

6               110             (5, '0')


Compressed: [(0, '0'), (0, '1'), (2, '0'), (1, '1'), (2, '1'), (5, '0')]

Decompressed: 01100111110

Decompression successful! The data matches the original input.
```

Experiment Number: 04

Name of the Experiment: To Write a program to explain and implement Hamming Code.

Theory: Hamming code is an error detection and correction method used to maintain data integrity during transmission. It works by adding extra parity bits to the original data, forming a code that can detect and correct single-bit errors.The core concept of Hamming code is to place parity bits at specific positions within the data, typically at powers of 2 (like positions 1, 2, 4, 8, etc.). These parity bits are responsible for checking certain bits in the data to ensure that the total number of 1s in those checked positions is either even or odd, depending on whether even or odd parity is used.

Upon receiving the data, the receiver checks the parity of the bits. If there is an error, the parity check will fail, and the receiver can pinpoint the exact position of the error using the information provided by the parity bits. This allows the receiver to correct a single-bit error and restore the original data.

Hamming code is widely used in applications where reliable data transmission is crucial, such as in computer memory systems and digital communication technologies.

Algorithm:

1. **Input data bits:** Begin with the original data bits that need to be encoded.
2. **Determine the number of parity bits:** Let the number of data bits be **k**. To find the number of required parity bits **r**, solve for the smallest **r** such that the following condition holds:

$$2^p \geq p + k + 1$$

   This ensures that there are enough parity bits to cover all data bits.

3. **Set the positions of parity bits:** Place the parity bits at positions that are powers of 2 (such as positions 1, 2, 4, 8, etc.).
4. **Insert data bits:** Place the original data bits in the remaining positions (those that are not powers of 2).
5. **Calculate parity bits:** For each parity bit, calculate its value so that the total number of 1's in the positions it checks (including the parity bit itself) is even. Use the following rule for each parity bit:
   - The parity bit at position 2^i checks all bit positions where the i^{th}bit of the position number is 1. This ensures that the parity covers the correct bits.
6. **Output encoded data:** The final encoded word consists of both the original data bits and the calculated parity bits.
7. **Error detection (optional):** To detect errors, compare the received encoded word with the calculated parity bits. If the parity check fails, the error can be located and corrected by identifying the bit position with an error. This can be done using the combination of parity checks that failed, which corresponds to the binary representation of the error position.

Example:

Let's go through this example to understand the Hamming code encoding and decoding process, including error detection and correction.

Encoding the Data Bits

1. **Input Data Bits**: We start with four data bits: 1 0 1 1.
2. **Calculate Parity Bits**:
    - **p1** is calculated to ensure even parity for positions 1, 3, 5, and 7.
    - **p2** is calculated to ensure even parity for positions 2, 3, 6, and 7.
    - **p3** is calculated to ensure even parity for positions 4, 5, 6, and 7.

For our data bits 1 0 1 1, the calculated parity bits are p1 = 0, p2 = 1, and p3 = 0.

3. **Construct the Codeword**: The Hamming codeword is formed by arranging the parity and data bits in the order [p1, p2, d1, p3, d2, d3, d4], which results in the 7-bit codeword [0, 1, 1, 0, 0, 1, 1].

Transmitting and Receiving the Codeword

4. **Transmit Codeword**: The codeword [0, 1, 1, 0, 0, 1, 1] is sent.
5. **Received Codeword**: The received codeword, in this case, is exactly the same as the transmitted codeword: [0, 1, 1, 0, 0, 1, 1].

Decoding and Error Detection

6. **Error Checking**:
    - The program recalculates the parity bits for the received codeword to check for any discrepancies.
    - It uses the recalculated parity bits to determine an error position. In this case, all calculated parity checks match, meaning no error was detected.
7. **Extracting Original Data Bits**: Since no error is detected, the original data bits [1, 0, 1, 1] are extracted directly from the received codeword.
8. **Output**: The program outputs:
    - **Corrected Data Bits**: [1, 0, 1, 1], which matches the original data.
    - **Error Position**: 0, indicating no error was detected in the received codeword.

Python Code:

```
def calculate_parity_bits(data_bits):
    """Calculate parity bits for 4 data bits to form a 7-bit Hamming code."""
    p1 = data_bits[0] ^ data_bits[1] ^ data_bits[3]  # Parity bit for positions 1, 3, 5, 7
    p2 = data_bits[0] ^ data_bits[2] ^ data_bits[3]  # Parity bit for positions 2, 3, 6, 7
    p3 = data_bits[1] ^ data_bits[2] ^ data_bits[3]  # Parity bit for positions 4, 5, 6, 7
    return [p1, p2, p3]
def encode_hamming(data_bits):
    p1, p2, p3 = calculate_parity_bits(data_bits)
    codeword = [p1, p2] + [data_bits[0]] + [p3] + data_bits[1:]
    return codeword
def decode_hamming(received_codeword):
    p1, p2, d1, p3, d2, d3, d4 = received_codeword
    c1 = p1 ^ d1 ^ d2 ^ d4  # Checks parity for positions 1, 3, 5, 7
    c2 = p2 ^ d1 ^ d3 ^ d4  # Checks parity for positions 2, 3, 6, 7
    c3 = p3 ^ d2 ^ d3 ^ d4  # Checks parity for positions 4, 5, 6, 7
    error_position = (c3 << 2) | (c2 << 1) | c1
    corrected_codeword = received_codeword.copy()
    if error_position != 0:
        corrected_codeword[error_position - 1] ^= 1  # Flip the erroneous bit
        corrected_data_bits = [corrected_codeword[2], corrected_codeword[4], corrected_codeword[5], corrected_codeword[6]]
    return corrected_data_bits, error_position
data_bits = list(map(int, input("Enter 4 data bits separated by spaces: ").split()))
encoded_codeword = encode_hamming(data_bits)
print("Encoded Hamming Codeword to Transmit:", encoded_codeword)
received_codeword = list(map(int, input("Enter received 7-bit codeword separated by spaces: ").split()))
corrected_data, error_position = decode_hamming(received_codeword)
print("Corrected Data Bits:", corrected_data)
print("Error Position (1-7, 0 means no error):", error_position)
if error_position == 0:
    print("No error detected in the received codeword.")
```

```
else:
    print(f"Error detected and corrected at position {error_position}.")
```

Output:

```
Enter 4 data bits separated by spaces: 1 0 1 1
Encoded Hamming Codeword to Transmit: [0, 1, 1, 0, 0, 1, 1]
Enter received 7-bit codeword separated by spaces: 0 1 1 0 0 1 0
Corrected Data Bits: [1, 0, 1, 1]
Error Position (1-7, 0 means no error): 7
Error detected and corrected at position 7.
```

Experiment Number: 05

Name of the Experiment: To Write a program to find the Channel Capacity and the conditional entropy of a binary symmetric channel having the following noise matrix with probability.

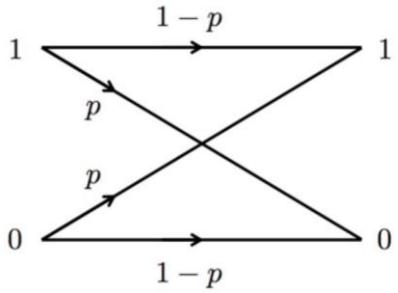
$$P(Y|X) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

Theory: A Binary Symmetric Channel (BSC) is a simple model used in communication theory to represent how binary data (0s and 1s) is transmitted through a noisy channel. In this model, every bit that is sent has a chance of being flipped due to noise.

Here's how it works:

- When a bit is sent, it has:
    - **Probability p** of flipping (changing from 0 to 1 or from 1 to 0).
    - **Probability 1 - p** of being received correctly (staying the same, either 0 or 1).

The BSC is called "symmetric" because the error probability (p) is the same for both 0 and 1. This model helps in understanding how noise impacts the transmission of binary data and is essential for creating error detection and correction methods.



Mathematical Representation of a Binary Symmetric Channel (BSC)

The BSC can be described mathematically by the following probabilities for the input-output combinations:

- **Pr (Y = 0 | X = 0) = 1 – p** (The probability of receiving 0 when 0 is sent.)
- **Pr (Y = 0 | X = 1) = p** (The probability of receiving 0 when 1 is sent.)
- **Pr (Y = 1 | X = 0) = p** (The probability of receiving 1 when 0 is sent.)
- **Pr (Y = 1 | X = 1) = 1 – p** (The probability of receiving 1 when 1 is sent.)

Here, **p** is the probability of a bit flipping (error), and it's assumed to be between **0** and **1/2** ($0 \leq p \leq 1/2$).

Equivalent Channel: If needed, the receiver can "invert" the interpretation of the output (i.e., treat 0 as 1 and 1 as 0) to create an equivalent channel with an error probability of **(1 - p)**. This is useful for analyzing the channel in different ways.

Why BSC is Important

- **Simplicity**: It's one of the simplest models for a noisy communication channel, making it easy to analyze.
- **Key to Many Problems**: Many problems in communication theory can be reduced to studying the BSC.
- **Building Blocks**: Understanding how to transmit effectively over a BSC provides insights that can help solve problems for more complex channels.

Python Code:

```
import math

def channel_capacity(matrix):
    p = matrix[0][1]  # `p` is the off-diagonal element in the first row
    if p == 0 or p == 1:
        return 0  # Capacity is zero when there's no uncertainty
    return 1 + p * math.log2(p) + (1 - p) * math.log2(1 - p)

def conditional_entropy(matrix):
    p = matrix[0][1]
    if p == 0 or p == 1:
        return 0  # No entropy when probability is 0 or 1
    return - (p * math.log2(p) + (1 - p) * math.log2(1 - p))

def get_probability_input(prompt):
    while True:
        try:
            value = eval(input(prompt))
            if 0 <= value <= 1:
                return value
            else:
                print("Please enter a number between 0 and 1.")
        except (ValueError, SyntaxError, NameError):
             print("Invalid input. Please enter a valid number or fraction (e.g., 0.5 or 2/3).")

print("Enter the values for the binary symmetric channel noise matrix P(Y|X):")

One_one = get_probability_input("Enter P(Y=0|X=0): ")

One_two = get_probability_input("Enter P(Y=1|X=0): ")
```

```
two_one = get_probability_input("Enter P(Y=0|X=1): ")
two_two = get_probability_input("Enter P(Y=1|X=1): ")
matrix = [
    [One_one, One_two],
    [two_one, two_two]
]
print("The Matrix is:", matrix)
capacity = channel_capacity(matrix)
print("Channel Capacity C:", capacity)
cond_entropy = conditional_entropy(matrix)
print("Conditional Entropy H(Y|X):", cond_entropy)
```

Output:

Enter the values for the binary symmetric channel noise matrix P(Y|X)

Enter P(Y=0|X=0): 2/3

Enter P(Y=0|X=1): 1/3

Enter P(Y=1|X=0): 1/3

Enter P(Y=1|X=1): 2/3

The Matrix is: [[0.6666666666666666, 0.3333333333333333], [0.3333333333333333, 0.6666666666666666]]

Channel Capacity C: 0.0817041659455105

Conditional Entropy H(Y|X): 0.9182958340544896

Experiment Number: 06

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

Theory: Huffman coding is considered optimal for single-symbol encoding because it satisfies three essential criteria: **Kraft's inequality**, the **prefix-free property**, and **minimized average code length**.

Kraft's Inequality: This inequality ensures that the code is uniquely decodable. It states that for a set of code lengths, the sum of the reciprocals of 2 raised to the power of each code length must be less than or equal to 1. Huffman coding meets this condition by construction, ensuring that each code is decodable without ambiguity. Mathematically, for a set of code lengths $l_1, l_2, \ldots, l_n$

$$\sum_{i=1}^{n} \frac{1}{2^{li}} \leq 1$$

Prefix-Free Property: In Huffman coding, no code word is a prefix of another. This is critical for preventing decoding errors. Huffman achieves this by organizing symbols in a binary tree, where each leaf node represents a symbol. Each unique symbol's code is derived from the path from the root to the leaf, and no code is part of another, ensuring unambiguous decoding.

Minimized Average Code Length: Huffman coding assigns shorter codes to more frequent symbols and longer codes to less frequent symbols. This minimizes the average code length, making the encoding as efficient as possible for the given distribution of symbols. This is key to achieving effective data compression, as the final encoded data uses the least amount of space while preserving the original information.

In conclusion, Huffman coding is optimal for single-symbol compression because it adheres to these principles, ensuring it is both efficient and error-free.

Python Code:

```
import heapq

import math

class HuffmanNode:

    def __init__(self, symbol, freq):

        self.symbol = symbol     # Symbol (like 'A', 'B', etc.)

        self.freq = freq         # Frequency (or probability) of the symbol

        self.left = None         # Left child

        self.right = None        # Right child

    def __lt__(self, other):

        return self.freq < other.freq
```

```
def build_huffman_tree(symbols):
    heap = [HuffmanNode(symbol, freq) for symbol, freq in symbols]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = HuffmanNode(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def generate_huffman_codes(node, prefix="", codebook={}):
    if node.symbol is not None:
        codebook[node.symbol] = prefix
    else:
        generate_huffman_codes(node.left, prefix + "0", codebook)
        generate_huffman_codes(node.right, prefix + "1", codebook)
    return codebook
def calculate_entropy(symbols):
    return -sum(freq * math.log2(freq) for _, freq in symbols if freq > 0)
symbols = [("A", 0.05), ("B", 0.1), ("C", 0.15), ("D", 0.2),("E", 0.23),("F", 0.27)]
root = build_huffman_tree(symbols)
huffman_codes = generate_huffman_codes(root)
entropy = calculate_entropy(symbols)
avg_length = sum(freq * len(huffman_codes[symbol]) for symbol, freq in symbols)
print("Huffman Codes for each symbol:")
for symbol, code in huffman_codes.items():
    print(f"{symbol}: {code}")
print(f"\nEntropy: {entropy:.4f}")
print(f"Average Huffman Code Length: {avg_length:.4f}")
tolerance = 0.1  # Allow a small tolerance for practical overhead
print("Huffman Code is Optimal:", avg_length <= entropy + tolerance)
```

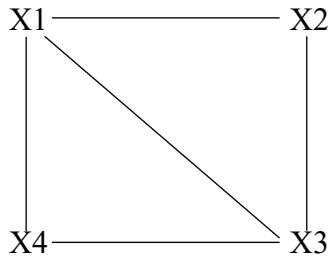
Output:

```
Huffman Codes for each symbol:
D: 11
E: 10
F: 01
C: 001
A: 0001
B: 0000

Entropy: 2.4209
Average Huffman Code Length: 2.4500
Huffman Code is Optimal: True
```

Experiment Number: 07

Name of the Experiment: To Write a program to find the entropy rate of a random walk on the following weighted graph.



Theory: The entropy rate of a random walk on a weighted graph quantifies the uncertainty in the transitions of a particle as it moves from node to node. In this scenario, we have a connected, undirected graph where each edge between node ***i*** and node ***j*** has a weight W_{ij}. If there is no edge between nodes ***i*** and ***j***, the weight is zero. The particle moves from one node to another with a probability proportional to the weight of the edge connecting them.

The random walk is represented as a sequence of vertices $\{X_t\}$where X_t is the node the particle is at time t. Given that the particle is at node i, the next node j is chosen randomly from the nodes connected to iii, with the transition probability proportional to the edge weight W_{ij}.

The stationary distribution of this random walk assigns a probability to each node based on the total weight of edges connected to it. The total weight for node i is:

$$W_{ij} = \sum_{i \neq j} W_{ij}$$

where W_{ij} is the weight of the edge between nodes i and j. The total weight of all edges in the graph is denoted as:

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

The stationary probability p_i for node i is then proportional to the total weight of edges emanating from node i:

$$p_i = \frac{W_i}{2W}$$

This shows that the stationary probability of being at node i depends only on the weight of the edges connected to iii, and not on other parts of the graph.

The entropy rate of the random walk measures the uncertainty in the transitions between nodes. It can be computed using the entropy of the stationary distribution and the transition probabilities. If all edges have equal weight, the stationary probability pip_ipi for each node iii is proportional to the number of edges E_i emanating from it. In this case, the entropy rate simplifies to:

$$H_{walk} = \log(E) - H(p)$$

where E is the total number of edges in the graph, and $H(p)$ is the entropy of the stationary distribution.

Thus, the entropy rate for a random walk on a weighted graph depends on the structure of the graph and the distribution of edge weights. If all edges have equal weight, the entropy rate is influenced primarily by the total number of edges and the entropy of the stationary distribution, making it relatively simple to compute.

Python Code:

```
import math

graph = {
    'A': {'B': 1, 'C': 2, 'D': 1},
    'B': {'A': 1, 'C': 1},
    'C': {'A': 2, 'B': 1, 'D': 1},
    'D': {'A': 1, 'C': 1}
}

def calculate_transition_probabilities(graph):
    transitions = {}
    for node, edges in graph.items():
        total_weight = sum(edges.values())
        transitions[node] = {neighbor: weight / total_weight for neighbor, weight
in edges.items()}
    return transitions

def entropy_rate(transitions):
    entropy = 0
    for node, edges in transitions.items():
        for prob in edges.values():
            if prob > 0:
                entropy += -prob * math.log2(prob)
    return entropy / len(transitions)  # Average over nodes

transitions = calculate_transition_probabilities(graph)

rate = entropy_rate(transitions)

print("Entropy Rate of Random Walk:", rate)
```

Output:

```
Entropy Rate of Random Walk: 1.25
```

Experiment Number: 08

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

$$\begin{bmatrix} 1/8 & 1/16 & 1/32 & 1/32 \\ 1/16 & 1/8 & 1/32 & 1/32 \\ 1/16 & 1/16 & 1/16 & 1/16 \\ 1/4 & 0 & 0 & 0 \end{bmatrix}$$

Theory: Information theory provides a mathematical framework for quantifying information, uncertainty, and dependence between random variables. When working with joint probability distributions, several key information-theoretic quantities can be calculated to gain insights into the underlying relationships.

Key Concepts and Calculations

Marginal Probabilities

- Definition: The marginal probability of a random variable is the probability of that variable occurring without regard to the value of the other variable.
- Calculation:
  - $P(X=x) = \Sigma_y P(X=x, Y=y)$
  - $P(Y=y) = \Sigma_x P(X=x, Y=y)$

Entropy

- Definition: Entropy measures the uncertainty or randomness associated with a random variable.
- Calculation:
  - $H(X) = - \Sigma_x P(X=x) * \log2(P(X=x))$
  - $H(Y) = - \Sigma_y P(Y=y) * \log2(P(Y=y))$

Joint Entropy

- Definition: Joint entropy measures the uncertainty associated with a pair of random variables.
- Calculation:
  - $H(X,Y) = - \Sigma_x \Sigma_y P(X=x, Y=y) * \log2(P(X=x, Y=y))$

Conditional Entropy

- Definition: Conditional entropy measures the uncertainty of one random variable given the value of another.
- Calculation:
  - $H(X|Y) = \Sigma_y P(Y=y) * H(X|Y=y)$

- $H(Y|X) = \Sigma_x P(X=x) * H(Y|X=x)$

Mutual Information

- Definition: Mutual information measures the amount of information shared between two random variables.
- Calculation:
  - $I(X;Y) = H(X) + H(Y) - H(X,Y)$
  - Alternatively:
    - $I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$

Practical Application

To compute these quantities, follow these steps:

1. Calculate Marginal Probabilities: Sum the rows and columns of the joint probability matrix.
2. Calculate Entropy: Apply the entropy formula to the marginal probability distributions.
3. Calculate Joint Entropy: Apply the joint entropy formula to the joint probability matrix.
4. Calculate Conditional Entropy: Compute conditional probabilities and then apply the conditional entropy formula.
5. Calculate Mutual Information: Use any of the equivalent formulas to compute the mutual information.

By understanding and calculating these information-theoretic quantities, we can gain valuable insights into the relationships between random variables, the amount of information they share, and the uncertainty associated with them.

Python Code:

```
import numpy as np


# Joint probability matrix as shown in the image
matrix = np.array([
    [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]
])
# Normalize the matrix to ensure the total sum is 1
matrix = matrix / np.sum(matrix)
```

```
# Marginal distribution of X and Y
px = np.sum(matrix, axis=0)  # Sum across rows for P(X)
py = np.sum(matrix, axis=1)  # Sum across columns for P(Y)
print("The Marginal distribution of X is",px)
print("The Marginal distribution of Y is",py)

def entropy(p):
    return -np.sum(p * np.log2(p, where=(p > 0)))
def joint_entropy(matrix):
    return -np.sum(matrix * np.log2(matrix, where=(matrix > 0)))
def conditional_entropy_X_given_Y(matrix, py):
    conditional_entropy_value = 0
    for j in range(len(py)):
        if py[j] > 0:
            # Calculate conditional distribution P(X|Y=j)
            px_given_y = matrix[:, j] / py[j]
            # Weight by P(Y=j) and add entropy for this distribution
            conditional_entropy_value += py[j] * entropy(px_given_y)
    return conditional_entropy_value
def conditional_entropy_Y_given_X(matrix, px):
    conditional_entropy_value = 0
    for i in range(len(px)):
        if px[i] > 0:
            # Calculate conditional distribution P(Y|X=i)
            py_given_x = matrix[:, i] / px[i]
            # Weight by P(X=i) and add entropy for this distribution
            conditional_entropy_value += px[i] * entropy(py_given_x)
    return conditional_entropy_value
def mutual_information(matrix, px, py):
    px_py = np.outer(py, px)  # Outer product for P(X) * P(Y)
    # Ensure no division by zero in log2 computation
    return np.sum(matrix * np.log2(matrix / px_py, where=(matrix > 0) & (px_py > 0)))
# Calculate Entropies
```

```
H_x = entropy(px)  # Entropy of X

H_y = entropy(py)  # Entropy of Y

H_xy = joint_entropy(matrix)  # Joint Entropy H(X, Y)

H_x_given_y = conditional_entropy_X_given_Y(matrix, py)  # Conditional Entropy H(X|Y)

H_y_given_x = conditional_entropy_Y_given_X(matrix, px)  # Conditional Entropy H(Y|X)

I_xy = mutual_information(matrix, px, py)  # Mutual Information I(X; Y)



# Print results

print("Entropy of X:", H_x)

print("Entropy of Y:", H_y)

print("Joint Entropy H(X, Y):", H_xy)

print("Conditional Entropy H(X|Y):", H_x_given_y)

print("Conditional Entropy H(Y|X):", H_y_given_x)

print("Mutual Information I(X; Y):", I_xy)
```

Output:

The Marginal distribution of X is [0.5   0.25 0.125 0.125]

The Marginal distribution of Y is [0.25 0.25 0.25 0.25]

Entropy of X: 1.75

Entropy of Y: 2.0

Joint Entropy H(X, Y): 3.375

Conditional Entropy H(X|Y): 1.375

Conditional Entropy H(Y|X): 1.625

Mutual Information I(X; Y): 0.375