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 method used for data compression. The basic idea is to assign shorter codes to more frequent characters and longer codes to less frequent characters, making the overall size of the data smaller when encoded. It works by creating a binary tree where each leaf node represents a character, and the path from the root to the leaf gives the code for that character.

The process starts by counting the frequency of each character in the data. Then, two characters with the lowest frequencies are merged into a new node, and this process repeats until only one node remains. This final tree is used to generate the Huffman codes, which are then used to encode the original data. The result is a more efficient representation of the data.

Algorithm of Huffman Codde:

1. **Count frequencies**:
    - List the frequency of each character in the text.
2. **Build a priority queue**:
    - Create a priority queue (or min-heap) where each node represents a character and its frequency.
3. **Build the tree**:
    - While there’s more than one node in the queue:
        - Remove the two nodes with the lowest frequency.
        - Create a new node with these two nodes as children, and the frequency as the sum of their frequencies.
        - Insert this new node back into the queue.
4. **Generate codes**:
    - Traverse the tree from root to each leaf node, assigning '0' for left branches and '1' for right branches. The path from the root to a leaf forms the Huffman code for the character.
5. **Encode the text**:
    - Replace each character in the text with its corresponding Huffman code.
6. **Decode the text** (optional):
    - Starting from the root of the Huffman tree, follow the path according to the bits in the encoded text until a leaf is reached, then output the character.

Example:

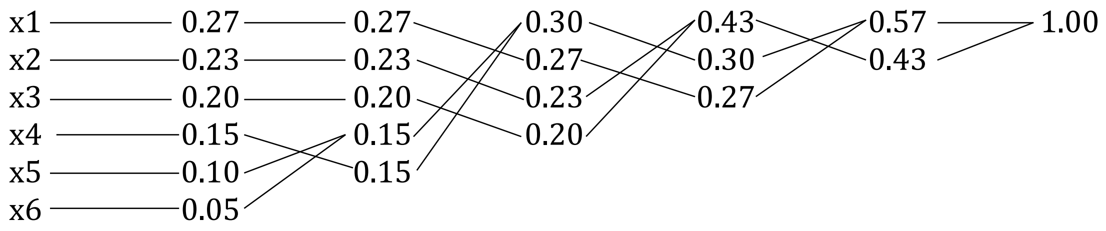
X	X1	X2	X3	X4	X5	X6
P(x)	0.05	0.10	0.15	0.20	0.23	0.27

We start with the given 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 sort the symbols by their probabilities in ascending order.

Next, we combine the two least probable symbols (X1 and X2, with probabilities 0.05 and 0.10) to form a new symbol with a combined probability of 0.15. This step is repeated by combining the next least probable symbols until we have a single symbol, which represents the entire set of symbols. After each combination, we sort the symbols again.

The process continues as we merge symbols based on their probabilities: X1+X2 and X3 (with probability 0.30), then X4 and X5 (with probability 0.43), followed by X6 and X1+X2+X3 (with probability 0.57), and finally combining the last two groups X4+X5 and X6+X1+X2+X3 to form a complete binary tree.

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 a technique used for error correction in digital communication. Unlike other codes that work with blocks of data, convolutional codes process data in a continuous stream, which makes them ideal for situations where data arrives in real-time, like audio or video signals.

The process involves using a shift register that stores a sequence of input bits, which then passes through logic gates (called generators). Each generator creates an output bit based on the current input bit and the previous bits stored in the shift register. This means each bit of input data influences multiple output bits, creating redundancy that helps detect and correct errors.

Decoding convolutional codes is usually done using an algorithm called the Viterbi algorithm, which traces back through the most likely paths in the received data to recover the original message. Convolutional codes are valued for their ability to enhance data reliability over noisy communication channels, especially when real-time processing is important.

Algorithm of Convolutional Codde:

1. **Input message bits**: Start with a sequence of input bits you want to encode.
2. **Initialize shift registers**: Use shift registers to hold a set of bits (a sliding window) that will be used to generate output bits.
3. **Shift bits**: For each input bit, shift it into the register, pushing older bits one position to the right.
4. **Apply generator functions**: Each generator function connects to specific positions in the shift register, performing a logical XOR operation on the selected bits.
5. **Generate output bits**: For each input bit, the generator functions produce a set of encoded bits as output.
6. **Repeat for each bit**: Continue shifting and encoding until all input bits are processed.

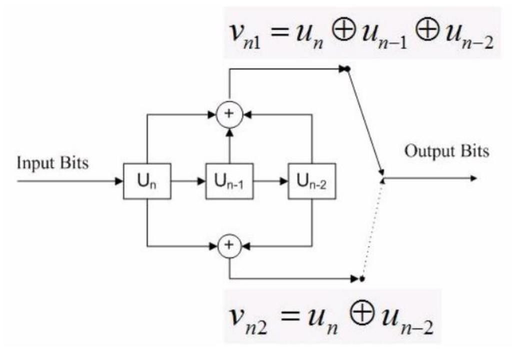

$v_{n1} = u_n \oplus u_{n-1} \oplus u_{n-2}$
Input Bits
U_n
U_{n-1}
U_{n-2}
Output Bits
$v_{n2} = u_n \oplus u_{n-2}$

Figure 1: Convolutional Code.

Example:

In this convolutional encoding example, the input bits are processed by a shift register and generator polynomials to generate encoded output bits. Let's walk through the example where the input data is 1011.

Initially, the shift register is empty, represented by [0, 0, 0]. The generator polynomials are G1 = 111 and G2 = 101, which means that the output bits are derived by XORing the current input bit with the last two bits in the register.

For the first input bit, 1, the register is updated by shifting in the new bit, resulting in the state [1, 0, 0]. The output bits are then calculated by XORing the bits in the register:

- The first output bit (y1) is the XOR of all three bits in the register: 1 XOR 0 XOR 0 = 1.
- The second output bit (y2) is the XOR of the first and third bits in the register: 1 XOR 0 = 1.

So, the encoded output for the first bit is 1 1.

Next, the input bit 0 shifts the register to [0, 1, 0]. The output is calculated again:

- The first output bit (y1) is 0 XOR 1 XOR 0 = 1.
- The second output bit (y2) is 0 XOR 0 = 0.

Thus, the encoded output for the second bit is 1 0.

For the third input bit, 1, the register is shifted to [1, 0, 1]. The output is:

- The first output bit (y1) is 1 XOR 0 XOR 1 = 0.
- The second output bit (y2) is 1 XOR 1 = 0.

So, the encoded output for the third bit is 0 0.

Finally, the input bit 1 shifts the register to [1, 1, 0]. The output is:

- 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 the fourth bit is 0 1.

Therefore, the final encoded data for the input 1011 is 1111011001. This encoding process ensures that each input bit is transformed into two output bits, allowing for redundancy and error 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 algorithm that replaces repeated occurrences of data with references to a single copy of that data. The idea is to find repeated sequences in the data and replace them with shorter representations, which reduces the size of the data.

The algorithm works by scanning the data and building a dictionary of sequences that have already appeared. As the algorithm processes each new piece of data, it checks if it exists in the dictionary. If it does, the algorithm outputs a reference to the previous occurrence instead of the sequence itself. If it doesn't, the algorithm adds the new sequence to the dictionary and continues.

Lempel-Ziv coding is particularly effective for compressing text or other data with repeated patterns. It forms the basis for popular compression algorithms like LZ77 and LZW, which are used in formats like GIF and ZIP.

Algorithm:

1. **Start with the input text**: Begin with the sequence of characters or symbols you want to compress.
2. **Search for longest matching substring**: Look for the longest substring in the input that matches a string that has already appeared in the processed part of the text.
3. **Encode the match**: If a match is found, encode it as a pair (position, length):
    - **Position**: The starting position of the match in the earlier part of the text.
    - **Length**: The length of the matching substring.
4. **Add the next unmatched character**: After the match, add the next unmatched character to the output.
5. **Repeat**: Continue searching for matches and adding unmatched characters until all the input is processed.
6. **Output**: The compressed output consists of pairs (position, length) and the remaining unmatched characters.

Example: Let's assume the input bit sequence is "010011010". To apply the Lempel-Ziv encoding process to this sequence, we will break it down into subsequences, assign a number representation to each subsequence, and then encode it as a binary block. We start with the first bit in the sequence, which is "0". Since this hasn't been seen before, we represent it as 0 and encode it as "0000". Next, we move to the second bit, which is "1". This hasn't been encountered either, so it is represented as 1 and encoded as "0001". The next subsequence is "00". This has not appeared before, so it is represented by 10 and encoded as "0010". After that, the next subsequence is "01". Since this also hasn't been seen yet, we represent it by 11 and encode it as "0011".

Now, we encounter the subsequence "011". This subsequence has not been seen before, so we represent it by 41 and encode it as "1001". Moving forward, we find the subsequence "10". This is a new subsequence as well, so it is represented by 20 and encoded as "0100". Finally, the subsequence "010" is encountered. This hasn't been seen before, so it is represented by 40 and encoded as "1000".

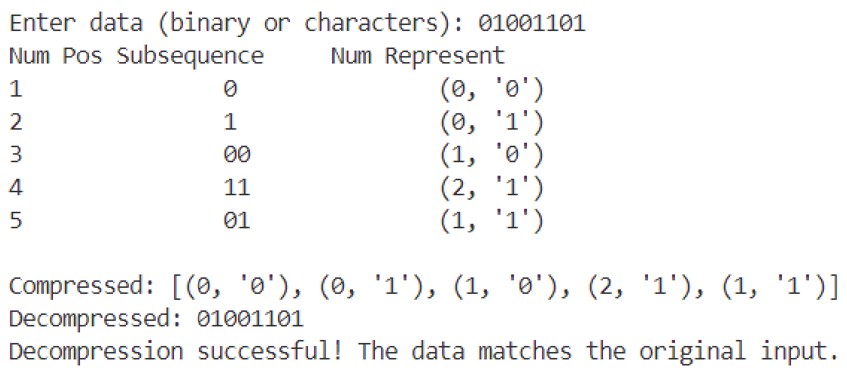
Table: Lempel-Ziv encoding table.

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): 01001101
Num Pos Subsequence     Num Represent
1               0               (0, '0')
2               1               (0, '1')
3               00              (1, '0')
4               11              (2, '1')
5               01              (1, '1')

Compressed: [(0, '0'), (0, '1'), (1, '0'), (2, '1'), (1, '1')]
Decompressed: 01001101
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 code used to ensure data integrity during transmission. It works by adding extra parity bits to the original data bits to create a code that can detect and correct single-bit errors.

The key idea behind Hamming code is to add parity bits at specific positions (powers of 2) in the data. These parity bits check the values of certain bits in the data, ensuring that the total number of 1s in those positions is even or odd, depending on the parity chosen (even or odd parity).

When the data is received, the receiver checks the parity of the data bits. If there's an error, the parity check will fail, and the receiver can detect the position of the error using the parity bits. With this information, the receiver can correct the single-bit error and recover the original data.

Hamming code is widely used in systems where reliable data transmission is essential, like computer memory and digital communication systems.

Algorithm:

1. **Input data bits**: Start with the original data bits you want to encode.
2. **Determine the number of parity bits**: Let the number of data bits be k. To determine the number of parity bits r, find the smallest r such that:

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

3. **Set the positions of parity bits**: Place the parity bits in positions that are powers of 2 (1, 2, 4, 8, etc.).
4. **Insert data bits**: Place the original data bits in the remaining positions (not at powers of 2).
5. **Calculate parity bits**: For each parity bit, calculate its value such that the total number of 1's in the positions covered by it (including the parity bit itself) is even. Use the following rule for each parity bit:
    - Parity bit at position 2^i checks all bits where the i^{th} bit of the position number is 1.
6. **Output encoded data**: The final encoded word consists of the data bits and the 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 bit position with an error can be identified and corrected.

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 1
Corrected Data Bits: [1, 0, 1, 1]
Error Position (1-7, 0 means no error): 0
No error detected in the received codeword.
```

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.

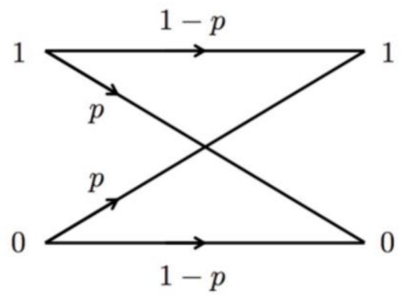
$$P(Y|X) = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

Theory: A binary symmetric channel (BSC) is a basic model in information theory used to represent a communication channel with binary data transmission. In this model, each bit sent (either 0 or 1) has a fixed probability of being received incorrectly due to noise in the channel. This probability of error is the same for both 0 and 1, making it "symmetric."

In a BSC, if a bit is sent, it has:

- A probability p of flipping (being received as the opposite bit, like 0 changing to 1 or 1 changing to 0).
- A probability 1−p of being received correctly.

The binary symmetric channel is used to analyze error probabilities and develop error correction codes, as it gives a simple yet effective way to understand how noise can affect binary data transmission.



Mathematical Representation:

- The probabilities of different input-output combinations are given as:
  - Pr(Y=0|X=0) = 1-p (Probability of receiving 0 when 0 is sent)
  - Pr(Y=0|X=1) = p (Probability of receiving 0 when 1 is sent)
  - Pr(Y=1|X=0) = p (Probability of receiving 1 when 0 is sent)
  - Pr(Y=1|X=1) = 1-p (Probability of receiving 1 when 1 is sent)

Assumptions: The crossover probability "p" is assumed to be between 0 and 1/2 (0 ≤ p ≤ 1/2).

Equivalent Channel: The receiver can swap the output interpretation (0 becomes 1 and vice versa) to obtain an equivalent channel with an error probability of (1-p).

Why BSC is Important:

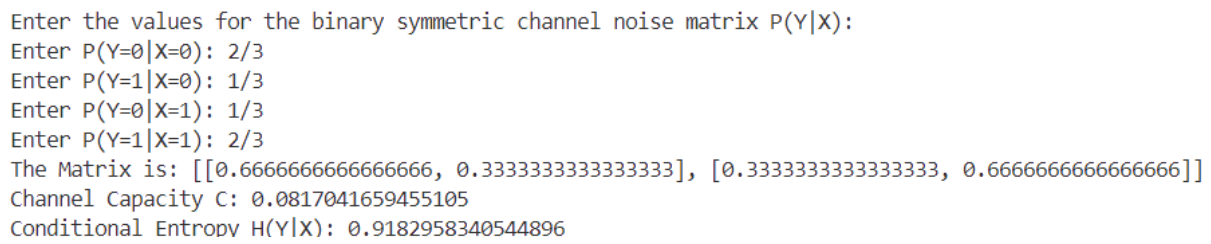
- It's one of the simplest noisy channel models to analyze.
- Many communication theory problems can be reduced to studying BSCs.
- Being able to transmit effectively over a BSC can lead to solutions for more complex channels.

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=1|X=0): 1/3
Enter P(Y=0|X=1): 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 optimal for single-symbol encoding because it meets three key criteria: Kraft's inequality, the prefix-free property, and minimized average code length.

Firstly, Huffman codes satisfy Kraft's inequality, which ensures that the code is uniquely decodable. Kraft's inequality 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. This condition guarantees that a unique decoding is possible, which Huffman codes naturally meet by construction. Mathematically, for a set of code lengths $l_1, l_2, \dots., l_n$:

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

Secondly, Huffman coding achieves the prefix-free property. This means that no code word is a prefix of another code word, which is crucial for unambiguous decoding. Huffman codes achieve this property by arranging symbols in a binary tree where each leaf represents a unique symbol, ensuring that every path in the tree ends distinctly. This is why Huffman codes can be decoded without confusion, making them highly efficient.

Lastly, Huffman coding minimizes the average code length. The algorithm assigns shorter codes to symbols with higher frequencies and longer codes to less frequent symbols, resulting in the lowest possible average code length for the given distribution of symbols. This approach is essential for efficient data compression, as it creates the most compact representation based on the symbol frequencies.

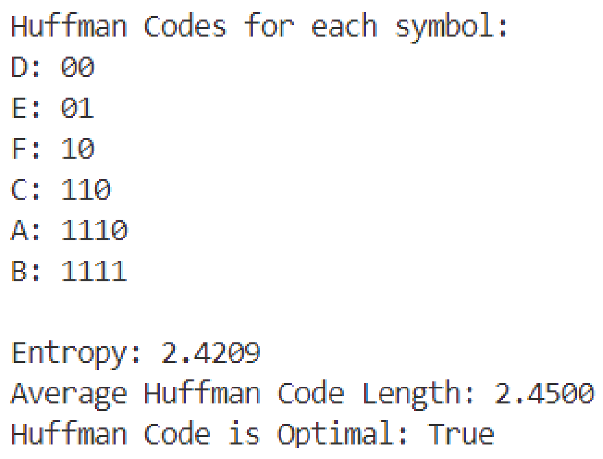
In summary, Huffman coding is optimal for symbol-by-symbol compression because it meets Kraft's inequality, maintains a prefix-free structure, and minimizes average code length, making it the most efficient encoding for single-symbol cases.

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
```

```python
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)
```

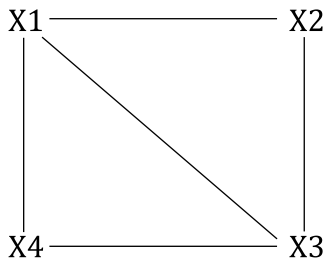
Output:



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

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: Computing Information-Theoretic Quantities from a Joint Probability Matrix

Given a joint probability matrix P(X,Y), we can compute various information-theoretic quantities to understand the relationship between the random variables X and Y.

1. Marginal Distributions:

- P(X=x): Sum the probabilities in the row corresponding to X=x.
- P(Y=y): Sum the probabilities in the column corresponding to Y=y.

2. Entropy:

- H(X): Measures the uncertainty of X.
- H(X) = - Σ P(X=x) * log2(P(X=x))
- H(Y): Measures the uncertainty of Y.
- H(Y) = - Σ P(Y=y) * log2(P(Y=y))

3. Joint Entropy:

- H(X,Y): Measures the joint uncertainty of X and Y.
- H(X,Y) = - Σ Σ P(X=x, Y=y) * log2(P(X=x, Y=y))

4. Conditional Entropy:

- H(X|Y): Measures the uncertainty of X given Y.
- H(X|Y) = Σ P(Y=y) * H(X|Y=y)
- H(Y|X): Measures the uncertainty of Y given X.
- H(Y|X) = Σ P(X=x) * H(Y|X=x)

5. Mutual Information:

- I(X;Y): Measures the amount of information shared between X and Y.
- I(X;Y) = H(X) + H(Y) - H(X,Y)

Alternatively,

I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)

Steps to Compute:

1. Calculate Marginal Probabilities: Sum rows and columns of the joint probability matrix.
2. Calculate Entropy: Use the formula for entropy, summing over all possible values of the random variable.
3. Calculate Joint Entropy: Use the formula for joint entropy, summing over all possible pairs of values.
4. Calculate Conditional Entropy: Calculate conditional probabilities using Bayes' theorem, then use the entropy formula.
5. Calculate Mutual Information: Use either of the formulas for mutual information, utilizing the previously calculated values.

By following these steps and using the provided formulas, you can effectively compute information-theoretic quantities from a joint probability matrix and gain insights into the relationships between random variables.

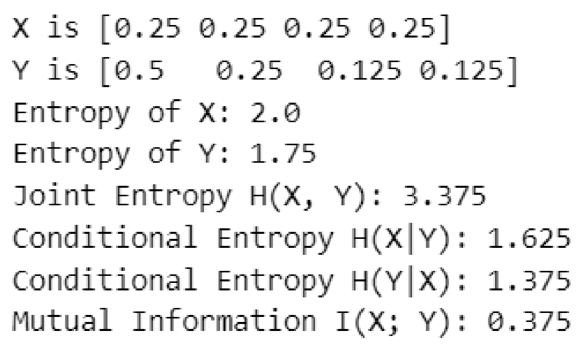
Python Code:

```
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=1)  # Sum across rows for P(X)

py = np.sum(matrix, axis=0)  # Sum across columns for P(Y)

print("X is",px)

print("Y is",py)

def entropy(p):

    """Calculate the entropy of a probability distribution."""

    return -np.sum(p * np.log2(p, where=(p > 0)))
```

```
def joint_entropy(matrix):
    """Calculate the joint entropy H(X, Y)."""
    return -np.sum(matrix * np.log2(matrix, where=(matrix > 0)))

def conditional_entropy_X_given_Y(matrix, py):
    """Calculate the conditional entropy H(X|Y)."""
    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):
    """Calculate the conditional entropy H(Y|X)."""
    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):
    """Calculate the mutual information I(X; Y)."""
    px_py = np.outer(px, py)  # 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:



```
X is [0.25 0.25 0.25 0.25]
Y is [0.5   0.25  0.125 0.125]
Entropy of X: 2.0
Entropy of Y: 1.75
Joint Entropy H(X, Y): 3.375
Conditional Entropy H(X|Y): 1.625
Conditional Entropy H(Y|X): 1.375
Mutual Information I(X; Y): 0.375
```