Experiment No : 01

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

Theory: Huffman coding is a method used in data compression to encode data efficiently, particularly for lossless data compression. It was developed by David A. Huffman in 1952 while he was a student at MIT. The key idea behind Huffman coding is to assign shorter codes to more frequent symbols and longer codes to less frequent symbols, resulting in overall compression of the data.
Symbol Frequencies:

Huffman coding starts with a frequency analysis of the symbols in the data to be encoded. For example, if we are encoding text, each symbol might be a character (e.g., 'a', 'b', 'c', etc.), and we calculate the frequency of occurrence of each symbol.
Constructing the Huffman Tree:

Once the frequencies of symbols are determined, Huffman coding constructs a binary tree called the Huffman tree. This tree is constructed in a way that ensures that more frequent symbols have shorter binary codes, reducing the average length of the encoded message.
The process starts by creating a leaf node for each symbol, each with a weight equal to its frequency. These leaf nodes are then repeatedly merged into a single parent node until there is only one node left, which becomes the root of the Huffman tree.
The merging process involves taking the two nodes with the lowest frequencies and creating a new parent node with a frequency equal to the sum of the frequencies of the two nodes. This parent node has the two nodes as its children.
This process continues until all nodes are merged into the root node, resulting in a binary tree where symbols are represented by the paths from the root to the leaf nodes.
Assigning Huffman Codes:

Once the Huffman tree is constructed, Huffman codes are assigned to each symbol based on the path from the root to the leaf node representing that symbol.
The Huffman code for a symbol is determined by traversing the tree from the root to the leaf node corresponding to the symbol. Each time a left branch is taken, a '0' is added to the code, and each time a right branch is taken, a '1' is added.
Since shorter codes are assigned to more frequent symbols, the most common symbols will have the shortest codes, leading to compression.
Encoding and Decoding:

To encode a message, each symbol in the message is replaced by its corresponding Huffman code.
To decode a Huffman-encoded message, the Huffman tree is traversed starting from the root, and at each step, the next bit in the encoded message is examined to determine whether to move left or right in the tree until a leaf node (symbol) is reached.

Source Code :

```
import heapq
from collections import defaultdict, Counter


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


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


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


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

Output:

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

Experiment No : 02

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

Theory: Encoder Structure:

The convolutional encoder consists of shift registers and modulo-2 adders (XOR gates).
It takes a stream of input bits and produces a stream of output bits that are a linear combination of the input bits and the previous states of the shift registers.
The structure of the encoder is defined by the generator polynomials, which determine the connections between the shift register stages and the XOR operations.
Shift Registers:

Shift registers are memory elements that store a fixed number of bits.
In the context of convolutional coding, each shift register represents a state in the encoder.
The input bits are shifted into the shift registers, and the output bits are generated based on the current states of the shift registers.
Generator Polynomials:

Generator polynomials define the connections between the shift register stages and the XOR operations.
Each generator polynomial corresponds to a specific output bit of the encoder.
The coefficients of the polynomial indicate which shift register stages are involved in generating the output bit.
Encoding Process:

During encoding, input bits are fed into the encoder one at a time.
At each clock cycle, the input bits are shifted into the shift registers.
The output bits are generated based on the current states of the shift registers and the generator polynomials.
The output sequence is the convolutional code, which has redundancy added to facilitate error detection and correction.

Source Code:

```
import numpy as np


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

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


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

Output:

```
Enter message: 1 0 1 0 1
Constraints: 4
Number of output(generator): 2
Enter bits for generator 0: 1 1 1 1
Enter bits for generator 1: 1 1 0 1
Encoded Message [1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1]
PS C:\Users\Sha Alam\Downloads\Compressed\Information-Theory-and-Coding-Sessional-main>
```

Experiment No : 03

Experiment Name : Write a program to implement Lempel-Ziv code

Theory: Lempel-Ziv (LZ) coding is a family of lossless data compression algorithms that are widely used in various applications. The primary idea behind LZ coding is to replace repeated occurrences of data with references to a previously occurring instance of that data. This approach exploits redundancy within the data to achieve compression.
Dictionary-based Compression:

LZ coding operates on the principle of dictionary-based compression, where a dictionary is maintained to store previously encountered sequences of data.
As the input data is processed, sequences of characters are replaced with references to entries in the dictionary.
The dictionary starts empty, and entries are added as new sequences are encountered.
Compression Process:

The compression process involves scanning the input data from left to right, identifying repeated sequences, and replacing them with references to entries in the dictionary.
When a new sequence is encountered that is not already in the dictionary, it is added to the dictionary, and its reference is output.
When a repeated sequence is found, its reference to the corresponding entry in the dictionary is output instead of the repeated sequence itself.
Dictionary Maintenance:

The dictionary can be implemented using various data structures such as hash tables, trees, or arrays.
Entries in the dictionary typically consist of a sequence of characters along with a reference or index to that sequence.
The dictionary may have a maximum size, and strategies such as dictionary replacement or resizing may be employed to manage its size.
Decompression Process:

The decompression process involves reconstructing the original data from the compressed representation.
As references to entries in the dictionary are encountered in the compressed data, the corresponding sequences are retrieved from the dictionary and output.
If a reference is encountered that points to a sequence not yet present in the dictionary (e.g., due to dictionary replacement), it may be reconstructed based on previously decoded sequences.

Source Code:

```
message = 'AABABBBABAABABBBABBABB'
dictionary = {}
tmp, i, last = '', 1, 0
Flag = True
for x in message:
    tmp += x
    Flag = False
```

```
    if tmp not in dictionary.keys():
        dictionary[tmp] = i
        tmp = ''
        i += 1
        Flag = True
if not Flag:
    last = dictionary[tmp]
res = ['1']
for char, idx in list(dictionary.items())[1:]:
    tmp, s = '', ''
    for x, j in zip(char[:-1], range(len(char))):
        tmp += x
        if tmp in dictionary.keys():
            take = dictionary[tmp]
            s = str(take) + char[j + 1:]
    if len(char) == 1:
        s = char
    res.append(s)
if last:
    res.append(str(last))

mark = {
    'A': 0,
    'B': 1
}
final_res = []
for x in res:
    tmp = ""
    for char in x:
        if char.isalpha():
            tmp += bin(mark[char])[2:]
        else:
            tmp += bin(int(char))[2:]
    final_res.append(tmp.zfill(4))

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

Output:

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

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

Experiment No : 04

Experiment Name : Write a program to implement Hamming code.

Theory: Hamming codes are a family of error-correcting codes that can detect and correct single-bit errors in transmitted data. They add redundant bits to the original data in such a way that errors can be detected and corrected at the receiver's end. Hamming codes are widely used in digital communication systems to ensure data integrity.
Basic Concept:

Hamming codes are linear error-correcting codes that operate on blocks of data bits.
The key idea is to add redundant parity bits to the original data bits, allowing the detection and correction of errors that occur during transmission.
Parity Check Matrix:

Hamming codes are defined by a parity check matrix (H matrix) that determines the relationships between the data bits and the parity bits.
Each row of the parity check matrix represents a parity check equation, specifying which data bits are involved in the calculation of each parity bit.
The number of parity bits in the Hamming code is determined by the number of rows in the parity check matrix.
Encoding Process:

During encoding, the original data bits are arranged into a matrix, and the parity bits are calculated based on the parity check equations defined by the parity check matrix.
The calculated parity bits are appended to the original data bits to form the encoded Hamming code.
Error Detection and Correction:

At the receiver's end, the received Hamming code is checked against the parity check equations to detect errors.
If an error is detected, the receiver uses the parity bits to identify and correct the erroneous bit.
Hamming codes can detect and correct single-bit errors, and they can also detect some multiple-bit errors.
Syndrome Calculation:

The syndrome of a received code word is calculated by multiplying the received code word by the transpose of the parity check matrix.
The syndrome provides information about the location of the error within the code word, enabling error correction.

Source Code:

```
def calculate_parity_bits(data):
    n = len(data)
    m = 0
    while 2 ** m < n + m + 1:
        m += 1
    parity_bits = [0] * m
```

```
    j = 0
    for i in range(1, n + m + 1):
        if i == 2 ** j:
            parity_bits[j] = calculate_parity(data, i)
            j += 1
    return parity_bits

def calculate_parity(data, parity_index):
    count = 0
    for i in range(len(data)):
        if data[i] == '1' and is_set(i + 1, parity_index):
            count += 1
    return count % 2

def is_set(bit, position):
    return bit & (1 << (position - 1))

def encode_data(data):
    n = len(data)
    m = 0
    while 2 ** m < n + m + 1:
        m += 1

    j = 0
    encoded_data = []
    for i in range(1, n + m + 1):
        if i == 2 ** j:
            encoded_data.append(0)
            j += 1
        else:
            if j < n:
                encoded_data.append(int(data[j]))
                j += 1
            else:
                encoded_data.append(0)

    parity_bits = calculate_parity_bits(encoded_data)
    for i in range(m):
        encoded_data[2 ** i - 1] = parity_bits[i]

    return ''.join(map(str, encoded_data))
```

```
def flip_bit(bit):
    return '0' if bit == '1' else '1'

def detect_error(encoded_data):
    m = 0
    while 2 ** m < len(encoded_data):
        m += 1

    error_position = 0
    for i in range(m):
        parity_index = 2 ** i
        calculated_parity = calculate_parity(encoded_data,
parity_index)
        if calculated_parity != int(encoded_data[parity_index - 1]):
            error_position += parity_index

    return error_position

def correct_error(encoded_data):
    error_position = detect_error(encoded_data)
    if error_position != 0:
        encoded_data = list(encoded_data)
        encoded_data[error_position - 1] =
flip_bit(encoded_data[error_position - 1])
        return ''.join(encoded_data)
    else:
        return encoded_data

def decode_data(encoded_data):
    m = 0
    while 2 ** m < len(encoded_data):
        m += 1

    decoded_data = []
    j = 0
    for i in range(1, len(encoded_data) + 1):
        if i == 2 ** j:
            j += 1
        else:
            decoded_data.append(encoded_data[i - 1])

    return ''.join(decoded_data)
```

```
# Example usage:
data = "1101"  # Input data
encoded_data = encode_data(data)
print("Encoded data:", encoded_data)

# Simulate an error by flipping one bit
error_position = 3
encoded_data = list(encoded_data)
encoded_data[error_position - 1] = flip_bit(encoded_data[error_position
- 1])
encoded_data = ''.join(encoded_data)
print("Data with error:", encoded_data)
# Correct the error
corrected_data = correct_error(encoded_data)
print("Corrected data:", corrected_data)

# Decode the corrected data
decoded_data = decode_data(corrected_data)
print("Decoded data:", decoded_data)
```

Output:
Encoded data: 0000000
Data with error: 0010000
Corrected data: 0000000
Decoded data: 0000

Experiment No : 05

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

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

Now find the Channel Capacity C.

Theory: The channel capacity C of a binary symmetric channel (BSC) with a given noise probability p can be calculated using the Shannon-Hartley theorem. This theorem states that the channel capacity of a communication channel is determined by the bandwidth, signal power, and noise power. In the case of a binary symmetric channel:

Shannon-Hartley Theorem:

$$C = B \times \log_2(1 + \frac{S}{N})$$

Binary Symmetric Channel:

In a binary symmetric channel, the bandwidth
B can be considered as 1 (since it's a binary channel).
The signal power
S can be considered as 1 (since we're dealing with binary symbols).
The noise power
N is equal to the probability of error
p.

Channel Capacity Calculation:

$$C = \log_2(1 + \frac{1}{p})$$

This formula allows us to calculate the channel capacity
C of a binary symmetric channel based on the given noise probability
p. The channel capacity represents the maximum rate at which information can be reliably transmitted over the channel, measured in bits per channel use.

Source 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:

Huffman coding is a popular algorithm used for lossless data compression. It assigns variable-length prefix codes to input symbols based on their frequencies.
The core idea of Huffman coding is to assign shorter codes to more frequent symbols and longer codes to less frequent symbols, ensuring that no code is a prefix of another.
Optimality of Huffman Codes:

Huffman codes are optimal in terms of minimizing the average codeword length for a given set of symbol probabilities.
This optimality is proven by the Kraft-McMillan inequality, which states that for any prefix code, the sum of the codeword lengths must satisfy:

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$

where

Where li is the length of the ith codeword.
Huffman codes achieve the equality in the Kraft-McMillan inequality, meaning they produce the minimum average codeword length for a given set of symbol probabilities.
Checking Optimality:

To check the optimality of Huffman codes, we compare the average codeword length produced by Huffman coding with that of any other prefix-free code.
If the average codeword length produced by Huffman coding is shorter than or equal to the average codeword length produced by any other prefix-free code, then Huffman codes are considered optimal.

Source 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 = "aaabbbbbcccccddddee"
```

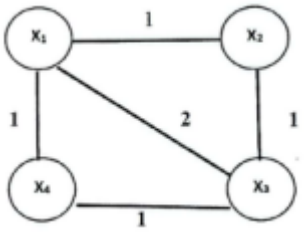
```
freq = calculate_frequency(message)
heap = build_heap(freq)
tree = build_tree(heap)
# tree=[20, ['D', '0'], ['B', '01'], ['E', '100'], ['A', '101'], ['C',
'11']] not optimal
huffman_avg_length = compute_huffman_avg_length(freq, tree,
len(message))
H = entropy(freq, len(message))
print("Huffman : %.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 : Write a code to find the entropy rate of a random walk on the following weighted graph



Theory: Entropy Rate:

Entropy rate is a measure of the uncertainty or randomness in a stochastic process. It quantifies the average rate of information produced by the process per unit time or step.
In the context of a random walk on a graph, the entropy rate measures the average amount of information generated by the random walk per step.
Random Walk on a Weighted Graph:

A random walk is a stochastic process where a "walker" moves from one vertex to another in a graph based on a set of probabilistic rules.
In a weighted graph, each edge has an associated weight or probability, indicating the likelihood of transitioning from one vertex to another.
The random walk on the graph is characterized by a transition probability matrix, where each entry represents the probability of transitioning from one vertex to another.
Entropy Rate Calculation:

To calculate the entropy rate of a random walk on a weighted graph, we first need to determine the transition probabilities between vertices.
We can construct the transition probability matrix based on the weights of the edges in the graph.
Once we have the transition probability matrix, we can compute the entropy rate using information theory concepts such as entropy and conditional entropy.

Source Code:

```
import math
from collections import defaultdict

# given
g = defaultdict(list)
xij = [[1, 1, 2], [1, 1], [1, 2, 1], [1, 1]]


def makeGraph(li):
    for node in range(len(li)):
        for x in li[node]:
            g[node].append(x)


def entropy(li):
    H = 0
    for x in li:
        if x == 0:
            continue
        H += -(x * math.log2(x))
    return H


# make graph
makeGraph(xij)
wi = []
for node in range(len(g)):
    wi.append(sum(g[node]))

# 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.

Y \ X	1	2	3	4
1	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$
2	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{32}$	$\frac{1}{32}$
3	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
4	$\frac{1}{4}$	0	0	0

Theory: Conditional Entropy:
Conditional entropy measures the uncertainty or randomness of a random variable given the value of another random variable.
It quantifies the amount of uncertainty remaining about one random variable once the value of another random variable is known.
For example, if we have two random variables X and Y, the conditional entropy (|)
H(X | Y) represents the average uncertainty in X given the value of Y.
Joint Entropy:

Joint entropy measures the uncertainty or randomness of a joint distribution of multiple random variables.
It quantifies the total amount of uncertainty present in a system characterized by multiple random variables.
For example, for two random variables X and Y, the joint entropy
(,)
H(X,Y) represents the average uncertainty in the combined system of X and Y.
Mutual Information:

Mutual information measures the amount of information that one random variable provides about another random variable.
It quantifies the degree of dependency between two random variables.
Mutual information can be seen as a measure of the reduction in uncertainty about one random variable when the value of another random variable is known.
It is calculated as the difference between the joint entropy of the two variables and the conditional entropy of one variable given the other.
High mutual information indicates a strong relationship or dependency between the variables, while low mutual information indicates independence.

Source Code:

```
# given
import math

matrix = [
    [1 / 8, 1 / 16, 1 / 32, 1 / 32],
    [1 / 16, 1 / 8, 1 / 32, 1 / 32],
    [1 / 16, 1 / 16, 1 / 16, 1 / 16],
    [1 / 4, 0, 0, 0]
]

# 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