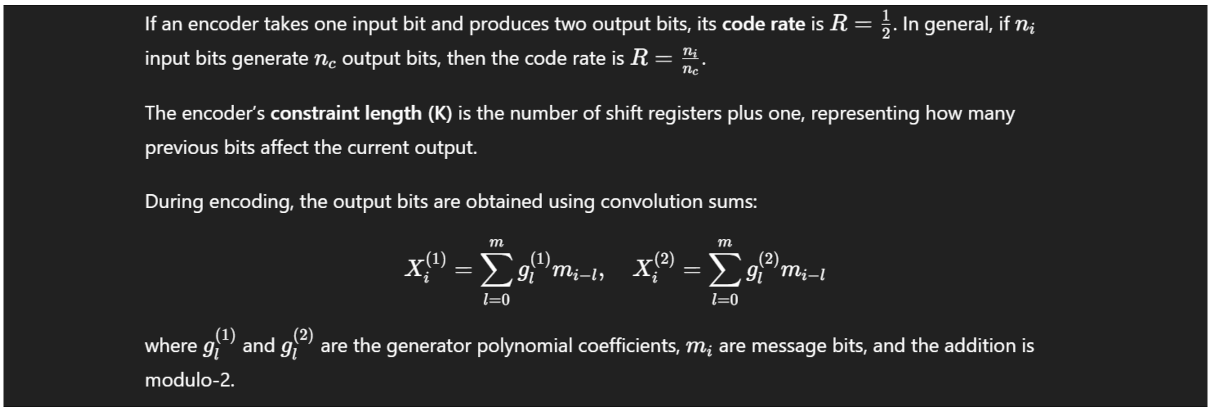
# <u>**Lab-01(convolutional Coding):**</u>

Theory

Convolution Coding is an error-correcting technique that introduces memory into the encoding process to improve data reliability during transmission. Unlike block codes, which encode data blocks independently, convolutional codes generate output bits based on the current input bit and several previous input bits. This dependency allows better detection and correction of transmission errors.

A convolutional encoder is a type of finite state machine that consists of:

- Shift registers – to store previous input bits (memory),
- Modulo-2 adders – to perform bitwise addition (XOR operation),
- Generator polynomials – to define how input bits are combined to form output bits.



If an encoder takes one input bit and produces two output bits, its **code rate** is $R = \frac{1}{2}$. In general, if $n_i$ input bits generate $n_c$ output bits, then the code rate is $R = \frac{n_i}{n_c}$.

The encoder's **constraint length (K)** is the number of shift registers plus one, representing how many previous bits affect the current output.

During encoding, the output bits are obtained using convolution sums:

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

where $g_l^{(1)}$ and $g_l^{(2)}$ are the generator polynomial coefficients, $m_i$ are message bits, and the addition is modulo-2.

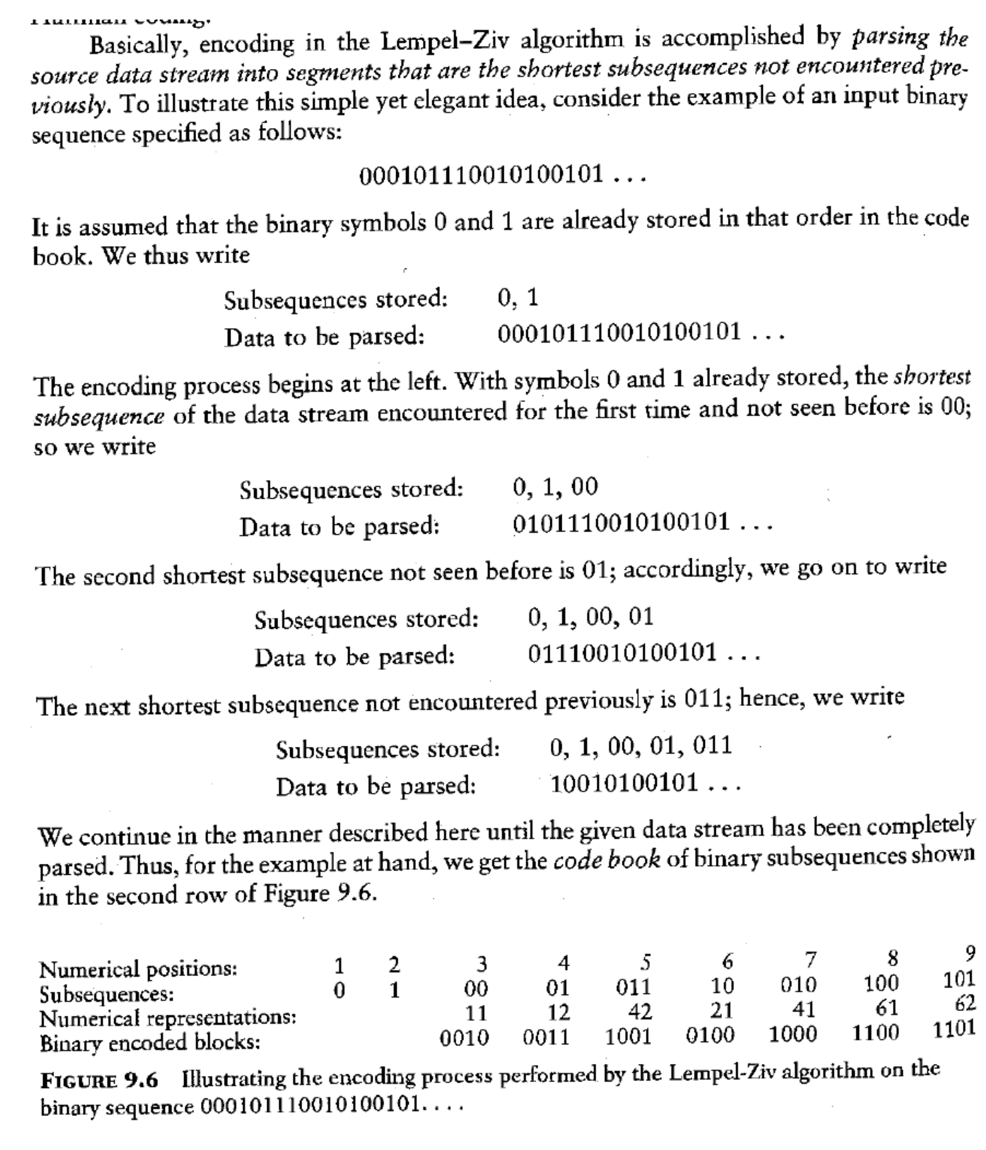
Finally, the output code sequence is a combination of all encoded bit streams. The memory elements of the encoder are initialized and terminated with zeros to ensure deterministic behavior.

Convolutional codes are widely used in digital communication systems such as satellite links, mobile networks, and deep-space communication, due to their strong error correction capability and efficiency in noisy channels.

**<u>Lab-02(lampel-ziv (LZ) code):</u>**

Lempel–Ziv (LZ) coding is a lossless data compression algorithm invented by *Abraham Lempel* and *Jacob Ziv*. It is one of the earliest and most widely used compression techniques — applied in formats like GIF and tools such as UNIX Compress.

The key idea of the Lempel–Ziv algorithm is to remove repeated patterns from data by building a dictionary of sequences that have appeared before.
Instead of storing the same pattern multiple times, the algorithm stores a short *reference* (called an *index*) that points to the previously seen sequence. This makes the overall data shorter but perfectly reversible (no information is lost).



Huffman coding.

Basically, encoding in the Lempel–Ziv algorithm is accomplished by *parsing the source data stream into segments that are the shortest subsequences not encountered previously*. To illustrate this simple yet elegant idea, consider the example of an input binary sequence specified as follows:

000101110010100101 . . .

It is assumed that the binary symbols 0 and 1 are already stored in that order in the code book. We thus write

| | |
|---|---|
| Subsequences stored: | 0, 1 |
| Data to be parsed: | 000101110010100101 . . . |

The encoding process begins at the left. With symbols 0 and 1 already stored, the *shortest subsequence* of the data stream encountered for the first time and not seen before is 00; so we write

| | |
|---|---|
| Subsequences stored: | 0, 1, 00 |
| Data to be parsed: | 0101110010100101 . . . |

The second shortest subsequence not seen before is 01; accordingly, we go on to write

| | |
|---|---|
| Subsequences stored: | 0, 1, 00, 01 |
| Data to be parsed: | 01110010100101 . . . |

The next shortest subsequence not encountered previously is 011; hence, we write

| | |
|---|---|
| Subsequences stored: | 0, 1, 00, 01, 011 |
| Data to be parsed: | 10010100101 . . . |

We continue in the manner described here until the given data stream has been completely parsed. Thus, for the example at hand, we get the *code book* of binary subsequences shown in the second row of Figure 9.6.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Numerical positions: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Subsequences: | 0 | 1 | 00 | 01 | 011 | 10 | 010 | 100 | 101 |
| Numerical representations: | | | 11 | 12 | 42 | 21 | 41 | 61 | 62 |
| Binary encoded blocks: | | | 0010 | 0011 | 1001 | 0100 | 1000 | 1100 | 1101 |

FIGURE 9.6 Illustrating the encoding process performed by the Lempel-Ziv algorithm on the binary sequence 000101110010100101. . . .

## Lab-03(Hamming Code)

Hamming codes are a family of linear error-correcting codes invented by Richard Hamming (1950). They add parity (redundant) bits to user data bits so that single-bit errors can be detected and corrected. The common classroom example is the Hamming(7,4) code, which maps 4 data bits into a 7-bit codeword by adding 3 parity bits.

Structure (Hamming(7,4))

Positions are numbered starting at 1:

- Parity bit positions: 1, 2, 4 (these are powers of two).
- Data bit positions: 3, 5, 6, 7.

So codeword bit order (pos 1..7): p1, p2, d1, p4, d2, d3, d4.

Each parity bit checks a subset of positions:

- p1 checks positions with LSB (bit 1) = 1: 1,3,5,7
- p2 checks positions with bit 2 = 1: 2,3,6,7
- p4 checks positions with bit 3 = 1: 4,5,6,7

Hamming codes typically use even parity: parity bit is set so that the XOR (parity) of the bits it covers (including itself) is 0.

Encoding algorithm (conceptual)

1. Place the 4 data bits into positions 3,5,6,7.
2. For each parity position (1,2,4), compute parity as XOR of the covered positions (excluding the parity itself) and set parity bit so that the total parity (including the parity bit) is 0 (even).
3. Output 7-bit codeword.

Decoding & Error Correction (syndrome method)

1. Receive 7-bit word. Compute the parity checks (syndrome bits):
   - s1 = XOR of received bits at positions 1,3,5,7
   - s2 = XOR of received bits at positions 2,3,6,7
   - s4 = XOR of received bits at positions 4,5,6,7
2. Combine syndrome bits into a binary number S = s4 s2 s1 (s4 is MSB).
   - If S = 0 → no single-bit error detected.
   - If S ≠ 0 → S gives the position (decimal) of the bit that is in error. Flip that bit to correct the single-bit error.
3. Extract data bits from positions 3,5,6,7 to recover original 4 data bits.

## Lab-04(BSC)

**Theory:**

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

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

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

Here:

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

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

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

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

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

To find the channel capacity, follow these steps:

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

Example

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

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

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

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

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

$$= -\frac{1}{3}\log_2\left(\frac{1}{3}\right) - \frac{2}{3}\log_2\left(\frac{2}{3}\right) = 0.918\ bits/msg\ symbol$$

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

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

## LAB-05:(Huffman Code):

Huffman coding is an important class of prefix codes.
The basic idea is to assign each symbol a binary sequence whose length is roughly proportional to the amount of information carried by that symbol. Symbols with higher probability get shorter codes, and symbols with lower probability get longer codes.

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

1. The sources symbol are listed in order of decreasing probability. The two listed in order of decreasing probability . The two sources symbols of lowest probability are assigned ao and a1. This part of the step is refffered to as a splitting stage .

2. These two source symbols are regarded as being combined into a new source symbol with probability equal to the sum two original probabilities. The probability of the news symbols is placed in the accordance with its value.

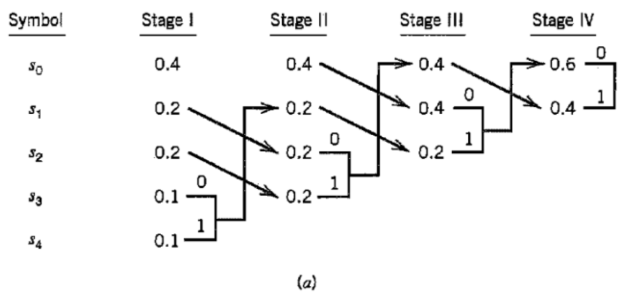
3. The procedure is repeated until we are left with a final list of source statistics of only two for which a 0 and a1 are assigned .

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

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

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

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



| Symbol | Probability | Code word |
|---|---|---|
| $s_0$ | 0.4 | 00 |
| $s_1$ | 0.2 | 10 |
| $s_2$ | 0.2 | 11 |
| $s_3$ | 0.1 | 010 |
| $s_4$ | 0.1 | 011 |

(b)

**FIGURE 9.5** (*a*) Example of the Huffman encoding algorithm. (*b*) Source code.

The average code-word length is therefore

$$\bar{L} = 0.4(2) + 0.2(2) + 0.2(2) + 0.1(3) + 0.1(3)$$
$$= 2.2$$

The entropy of the specified discrete memoryless source is calculated as follows [see Equation (9.9)]:

$$H(\mathscr{S}) = 0.4\log_2\left(\frac{1}{0.4}\right) + 0.2\log_2\left(\frac{1}{0.2}\right) + 0.2\log_2\left(\frac{1}{0.2}\right)$$

**Lab-06(Conditional joint entropy math):**

| $X$ / $Y$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | $^1/_8$ | $^1/_{16}$ | $^1/_{32}$ | $^1/_{32}$ |
| 2 | $^1/_{16}$ | $^1/_8$ | $^1/_{32}$ | $^1/_{32}$ |
| 3 | $^1/_{16}$ | $^1/_{16}$ | $^1/_{16}$ | $^1/_{16}$ |
| 4 | $^1/_4$ | 0 | 0 | 0 |

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

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

**Definition:** Given (X,Y) ~ P(x,y), the conditional entropy H(Y|X) is defined as:

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

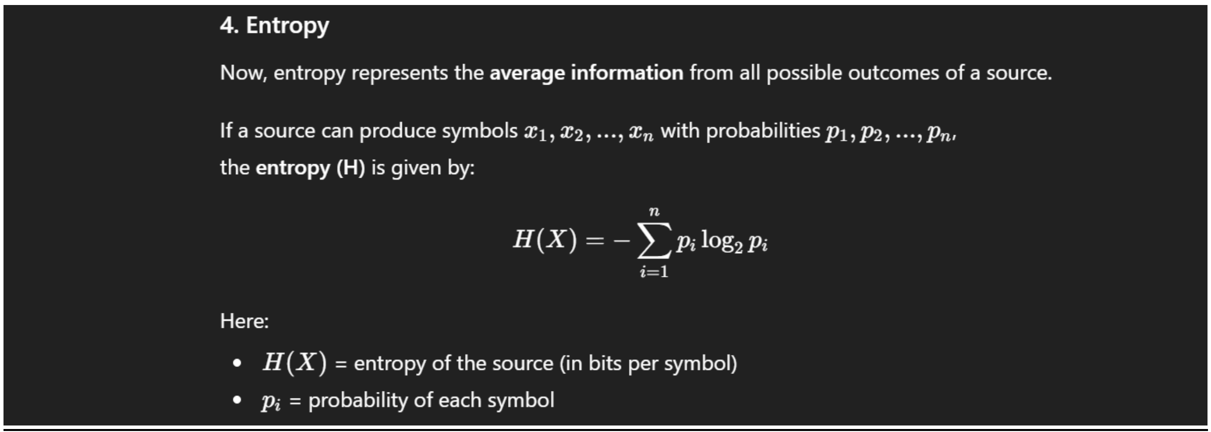
The marginal distribution of X is $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$, and the marginal distribution of Y is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, and hence:

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

Also,

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

Similarly, $H(X|Y) = \frac{13}{8}$ bits and $H(X,Y) = \frac{27}{8}$ bits.



### 4. Entropy

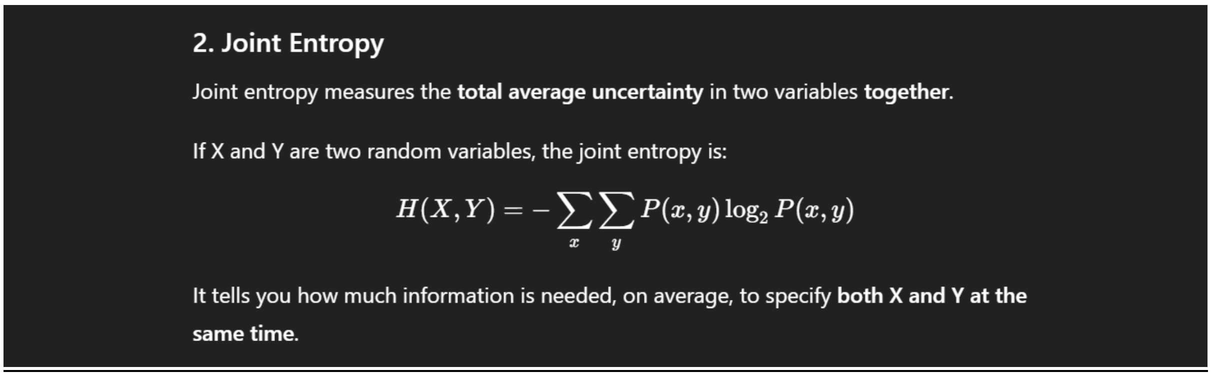
Now, entropy represents the **average information** from all possible outcomes of a source.

If a source can produce symbols $x_1, x_2, \ldots, x_n$ with probabilities $p_1, p_2, \ldots, p_n$, the **entropy (H)** is given by:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Here:

- $H(X)$ = entropy of the source (in bits per symbol)
- $p_i$ = probability of each symbol



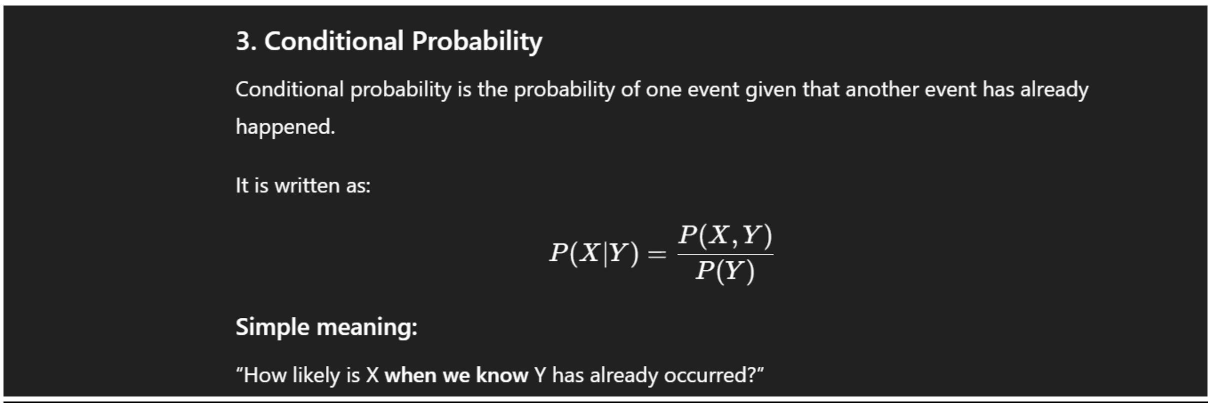
## 2. Joint Entropy

Joint entropy measures the **total average uncertainty** in two variables **together**.

If X and Y are two random variables, the joint entropy is:

$$H(X,Y) = -\sum_{x}\sum_{y} P(x,y) \log_2 P(x,y)$$

It tells you how much information is needed, on average, to specify **both X and Y at the same time**.



## 3. Conditional Probability

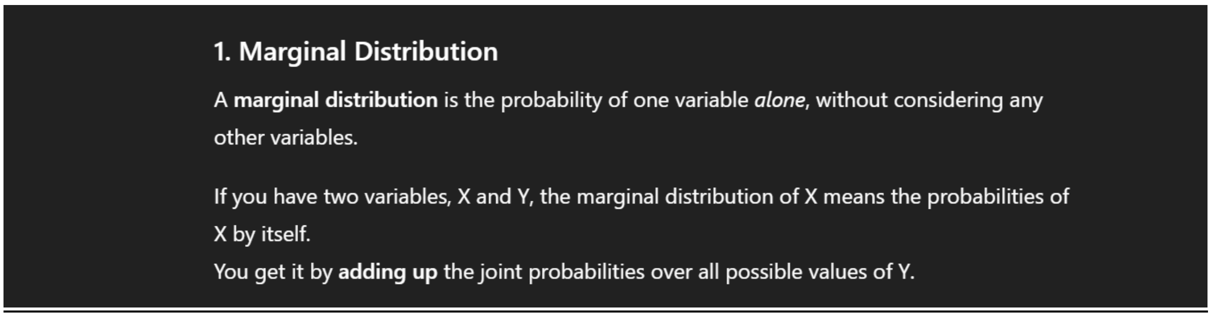
Conditional probability is the probability of one event given that another event has already happened.

It is written as:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

### Simple meaning:

"How likely is X **when we know** Y has already occurred?"

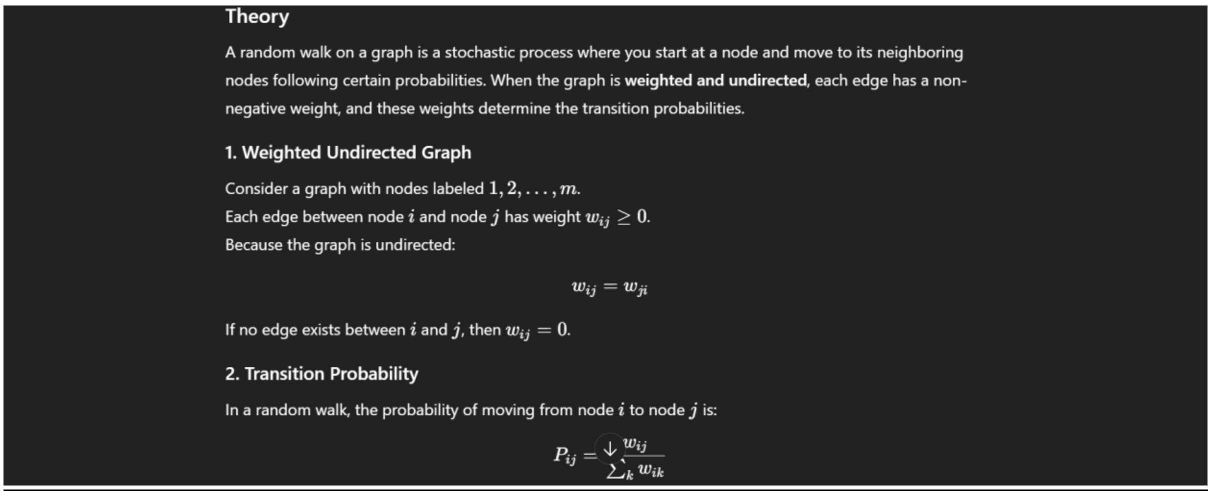


## 1. Marginal Distribution

A **marginal distribution** is the probability of one variable *alone*, without considering any other variables.

If you have two variables, X and Y, the marginal distribution of X means the probabilities of X by itself.
You get it by **adding up** the joint probabilities over all possible values of Y.

# Lab-07: (Weighted graph):



### Theory

A random walk on a graph is a stochastic process where you start at a node and move to its neighboring nodes following certain probabilities. When the graph is **weighted and undirected**, each edge has a non-negative weight, and these weights determine the transition probabilities.

#### 1. Weighted Undirected Graph

Consider a graph with nodes labeled $1, 2, \dots, m$.
Each edge between node $i$ and node $j$ has weight $w_{ij} \geq 0$.
Because the graph is undirected:

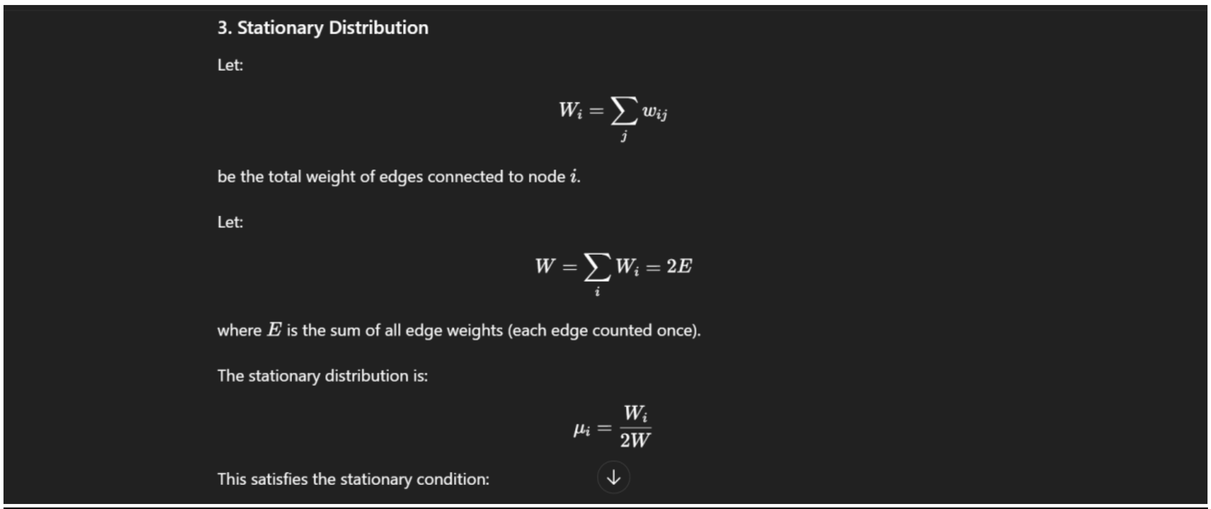
$$w_{ij} = w_{ji}$$

If no edge exists between $i$ and $j$, then $w_{ij} = 0$.

#### 2. Transition Probability

In a random walk, the probability of moving from node $i$ to node $j$ is:

$$P_{ij} = \frac{w_{ij}}{\sum_k w_{ik}}$$



#### 3. Stationary Distribution

Let:

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

be the total weight of edges connected to node $i$.

Let:

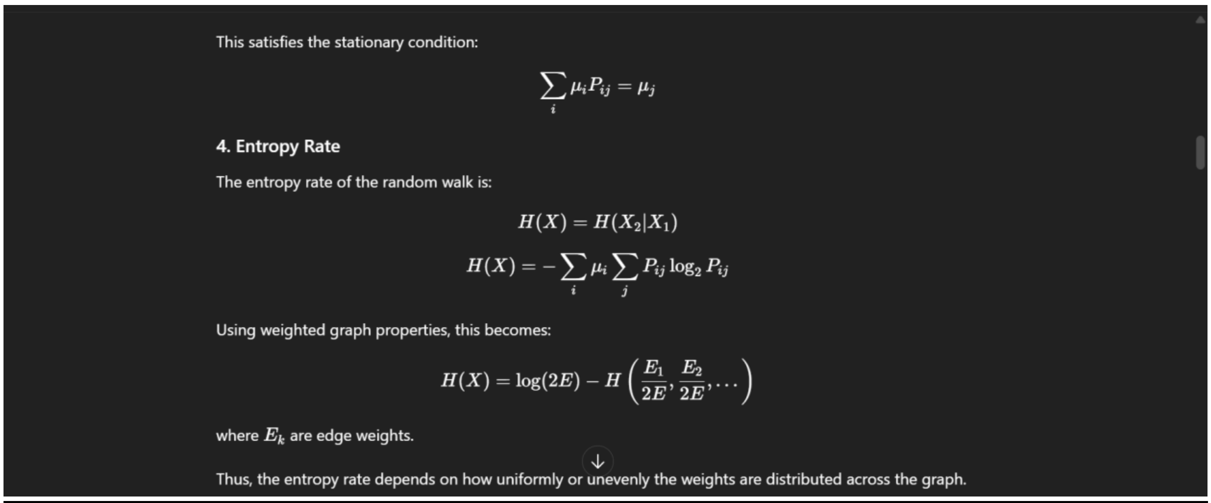
$$W = \sum_i W_i = 2E$$

where $E$ is the sum of all edge weights (each edge counted once).

The stationary distribution is:

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

This satisfies the stationary condition:



This satisfies the stationary condition:

$$\sum_i \mu_i P_{ij} = \mu_j$$

#### 4. Entropy Rate

The entropy rate of the random walk is:

$$H(X) = H(X_2|X_1)$$

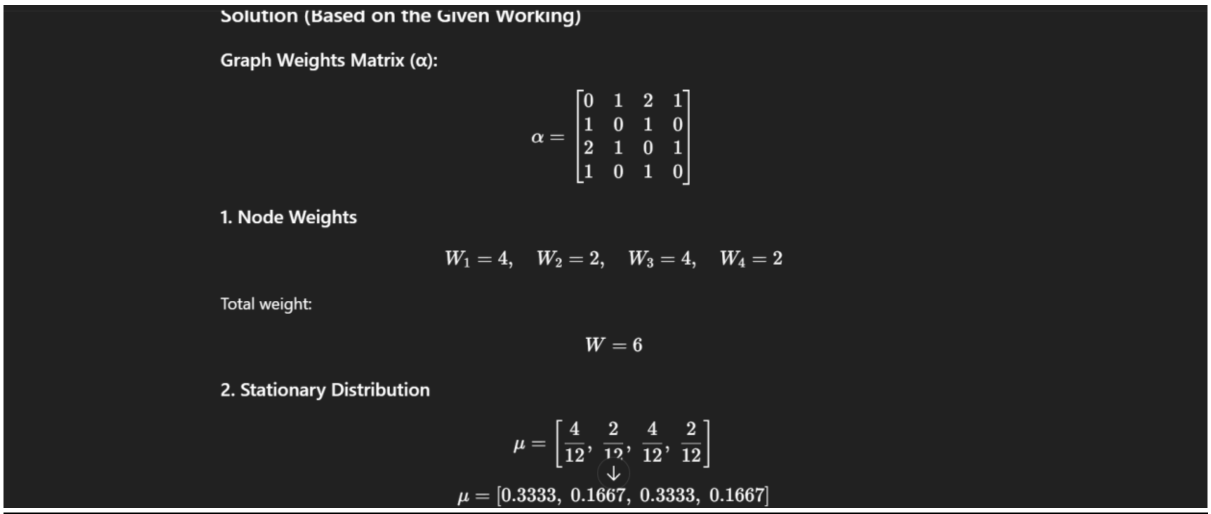
$$H(X) = -\sum_i \mu_i \sum_j P_{ij} \log_2 P_{ij}$$

Using weighted graph properties, this becomes:

$$H(X) = \log(2E) - H\left(\frac{E_1}{2E}, \frac{E_2}{2E}, \dots\right)$$

where $E_k$ are edge weights.

Thus, the entropy rate depends on how uniformly or unevenly the weights are distributed across the graph.



## Solution (Based on the Given Working)

**Graph Weights Matrix (α):**

$$\alpha = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

**1. Node Weights**

$$W_1 = 4, \quad W_2 = 2, \quad W_3 = 4, \quad W_4 = 2$$

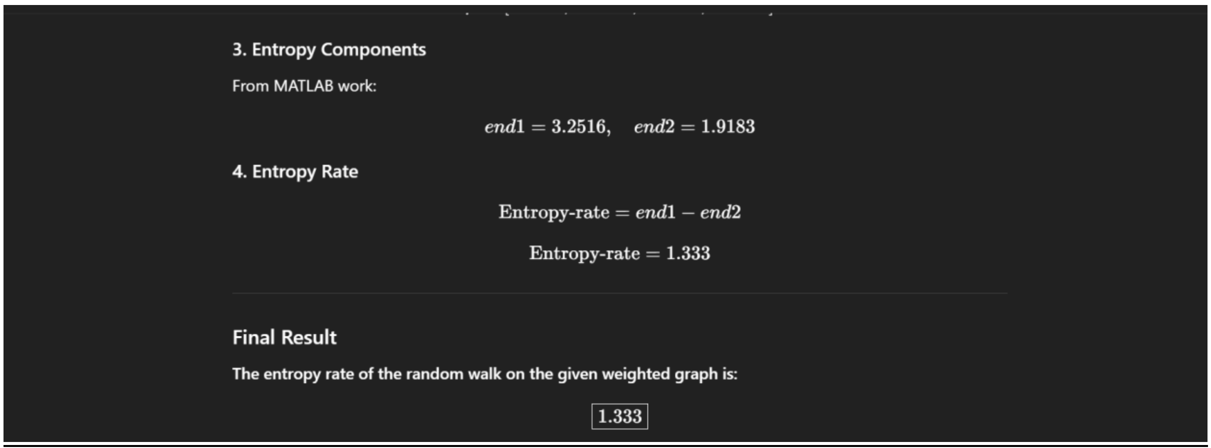
Total weight:

$$W = 6$$

**2. Stationary Distribution**

$$\mu = \left[\frac{4}{12}, \frac{2}{12}, \frac{4}{12}, \frac{2}{12}\right]$$

$$\mu = [0.3333,\ 0.1667,\ 0.3333,\ 0.1667]$$



**3. Entropy Components**

From MATLAB work:

$$end1 = 3.2516, \quad end2 = 1.9183$$

**4. Entropy Rate**

$$\text{Entropy-rate} = end1 - end2$$

$$\text{Entropy-rate} = 1.333$$

---

## Final Result

**The entropy rate of the random walk on the given weighted graph is:**

$$\boxed{1.333}$$