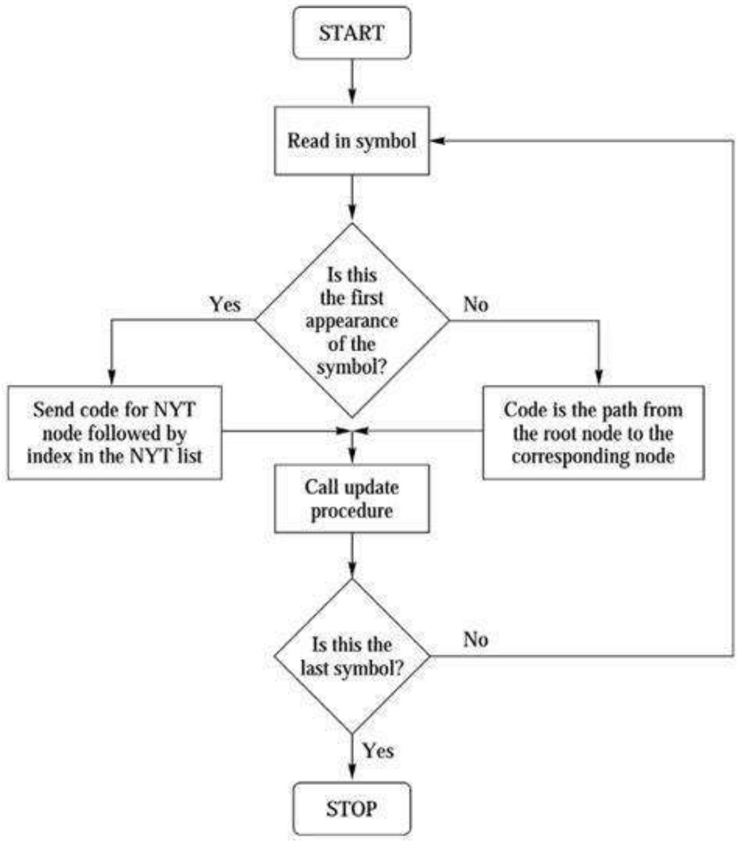
Name – Mohammed Gaus
Batch – SY – IT C3

ITC : Tutorial – 9

1. **A program for any source coding (compression) method along with the flowchart:**

➔ **Huffman Coding : Flow Chart**

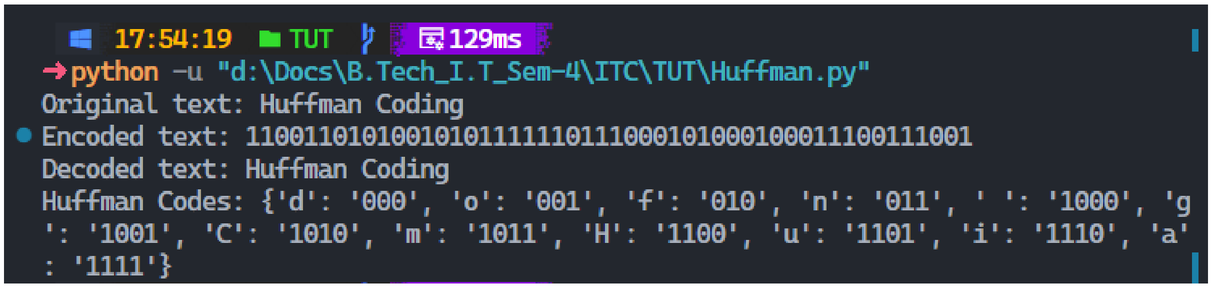

START
Read in symbol
Is this the first appearance of the symbol?
Yes
No
Send code for NYT node followed by index in the NYT list
Code is the path from the root node to the corresponding node
Call update procedure
Is this the last symbol?
No
Yes
STOP

CODE :

```
import heapq
from collections import defaultdict, Counter


class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq


def build_huffman_tree(text):
    frequency = Counter(text)
```

```python
    heap = [Node(char, freq) for char, freq in frequency.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)

        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]


def generate_codes(node, prefix="", code_dict=None):
    if code_dict is None:
        code_dict = {}

    if node is not None:
        if node.char is not None:
            code_dict[node.char] = prefix
        generate_codes(node.left, prefix + "0", code_dict)
        generate_codes(node.right, prefix + "1", code_dict)

    return code_dict


def encode(text, codes):
    return "".join(codes[char] for char in text)


def decode(encoded_text, root):
    decoded_text = []
    current_node = root
    for bit in encoded_text:
        current_node = current_node.left if bit == "0" else current_node.right
        if current_node.char is not None:
            decoded_text.append(current_node.char)
            current_node = root
    return "".join(decoded_text)


if __name__ == "__main__":
    text = "Huffman Coding"
    root = build_huffman_tree(text)
    codes = generate_codes(root)
    encoded_text = encode(text, codes)
    decoded_text = decode(encoded_text, root)
    print("Original text:", text)
    print("Encoded text:", encoded_text)
    print("Decoded text:", decoded_text)
    print("Huffman Codes:", codes)
```

OUTPUT :



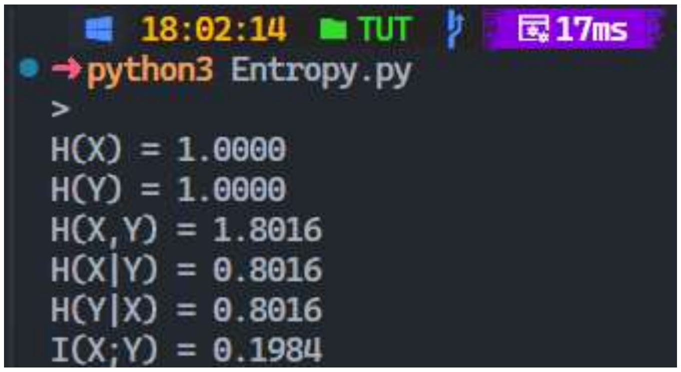
```
17:54:19  TUT  129ms
→python -u "d:\Docs\B.Tech_I.T_Sem-4\ITC\TUT\Huffman.py"
Original text: Huffman Coding
Encoded text: 11001101010010101111110111000101000100011100111001
Decoded text: Huffman Coding
Huffman Codes: {'d': '000', 'o': '001', 'f': '010', 'n': '011', ' ': '1000', 'g
': '1001', 'C': '1010', 'm': '1011', 'H': '1100', 'u': '1101', 'i': '1110', 'a'
: '1111'}
```

2. **Program to calculate H(X), H(Y) , H(X|Y), H(Y|X), H(X,Y) and I(X;Y) given the channel matrix and the symbol probabilities.**

➔ **CODE :**

```python
import numpy as np


def entropy(prob_dist):
    return -np.sum([p * np.log2(p) for p in prob_dist if p > 0])


def joint_entropy(joint_dist):
    return -np.sum([p * np.log2(p) for p in joint_dist.flatten() if p > 0])


def conditional_entropy(joint_dist, marginal):
    joint_entropy_val = joint_entropy(joint_dist)
    marginal_entropy = entropy(marginal)
    return joint_entropy_val - marginal_entropy


def mutual_information(joint_dist, marginal_x, marginal_y):
    h_x = entropy(marginal_x)
    h_y = entropy(marginal_y)
    h_xy = joint_entropy(joint_dist)
    return h_x + h_y - h_xy


def main():
    p_x = np.array([0.5, 0.5])
    channel_matrix = np.array([[0.8, 0.3], [0.2, 0.7]])

    p_y = channel_matrix.T @ p_x
    joint_distribution = np.array(
        [p_x[i] * channel_matrix[:, i] for i in range(len(p_x))]
    ).T

    h_x = entropy(p_x)
    h_y = entropy(p_y)
    h_xy = joint_entropy(joint_distribution)
    h_x_given_y = conditional_entropy(joint_distribution, p_y)
    h_y_given_x = conditional_entropy(joint_distribution.T, p_x)
    i_x_y = mutual_information(joint_distribution, p_x, p_y)

    print(f"H(X) = {h_x:.4f}")
    print(f"H(Y) = {h_y:.4f}")
    print(f"H(X,Y) = {h_xy:.4f}")
    print(f"H(X|Y) = {h_x_given_y:.4f}")
    print(f"H(Y|X) = {h_y_given_x:.4f}")
    print(f"I(X;Y) = {i_x_y:.4f}")


if __name__ == "__main__":
    main()
```

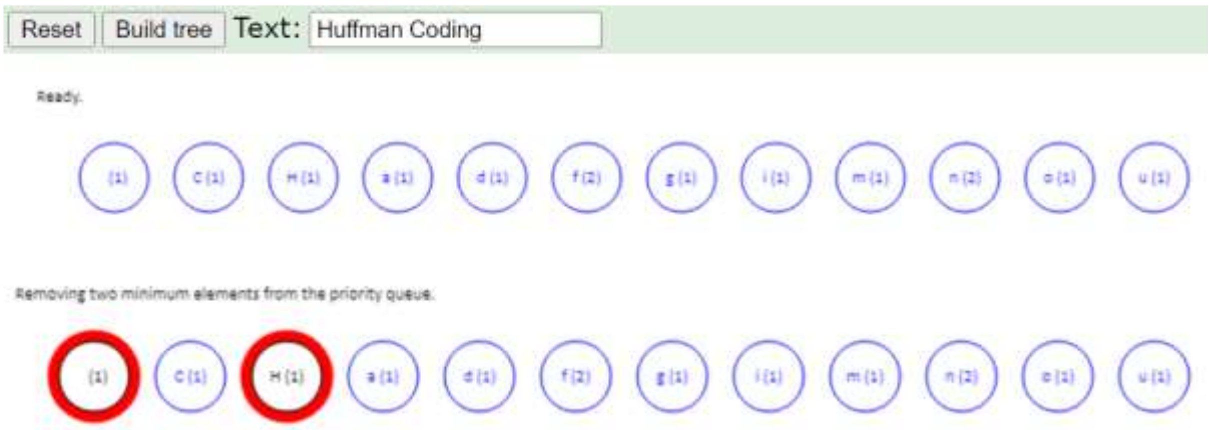
OUTPUT :



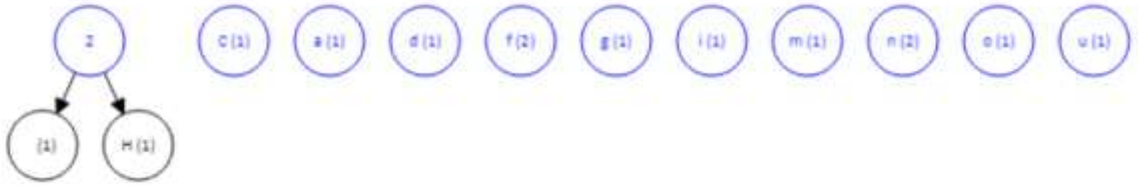
```
   18:02:14  TUT    17ms
 ➜python3 Entropy.py
 >
 H(X) = 1.0000
 H(Y) = 1.0000
 H(X,Y) = 1.8016
 H(X|Y) = 0.8016
 H(Y|X) = 0.8016
 I(X;Y) = 0.1984
```

3. Animation of any of the data compression methods mentioned above

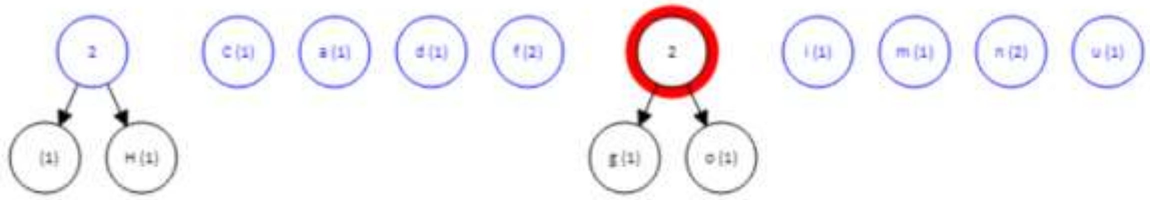
➔



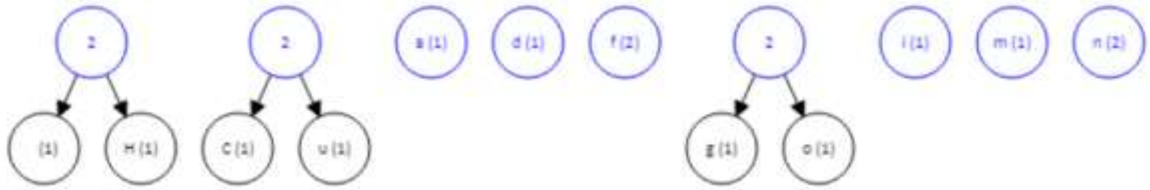
Reinserting the new root node in the priority queue.



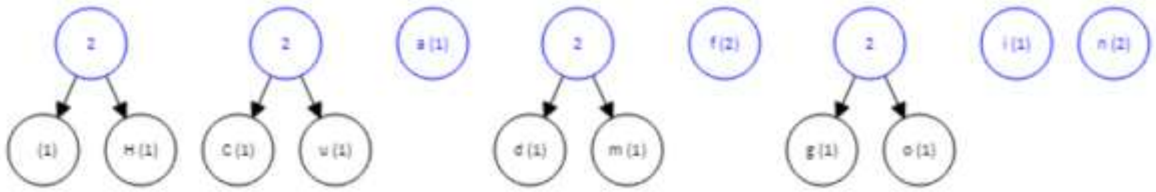
Reinserting the new root node in the priority queue.



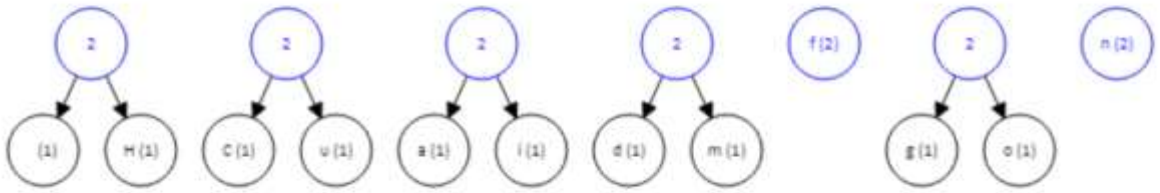
Reinserting the new root node in the priority queue.



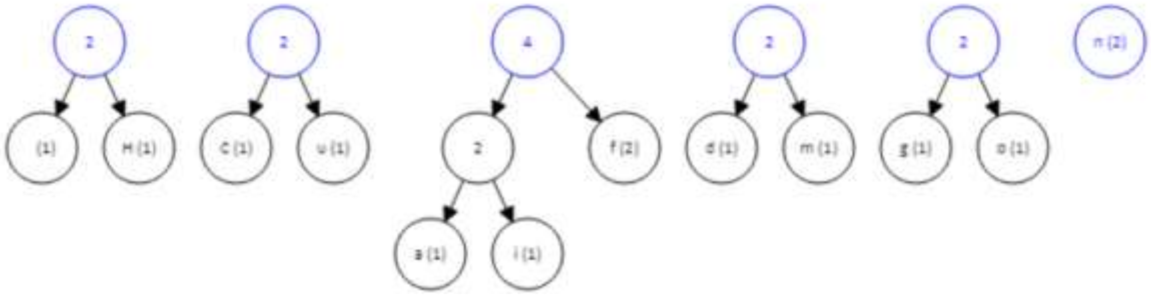
Reinserting the new root node in the priority queue.



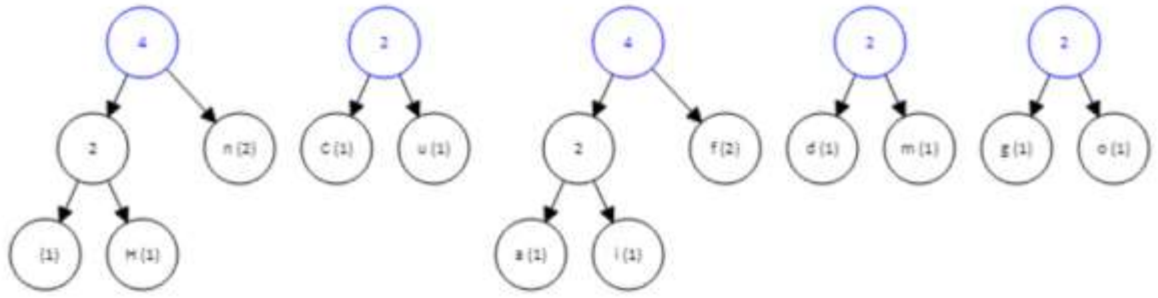
Reinserting the new root node in the priority queue.



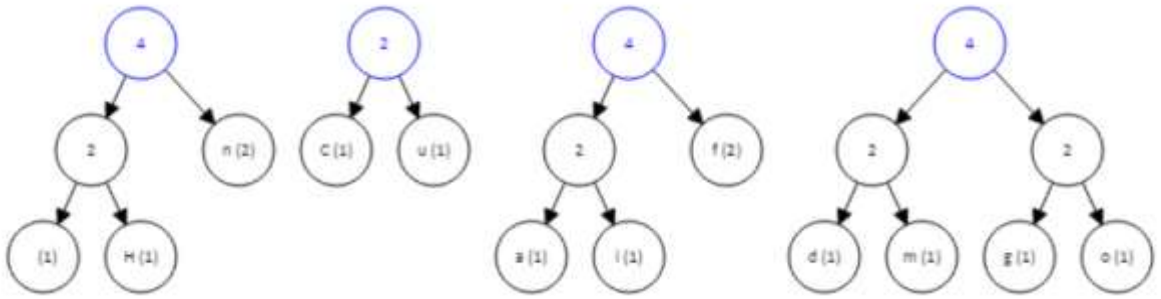
Reinserting the new root node in the priority queue.



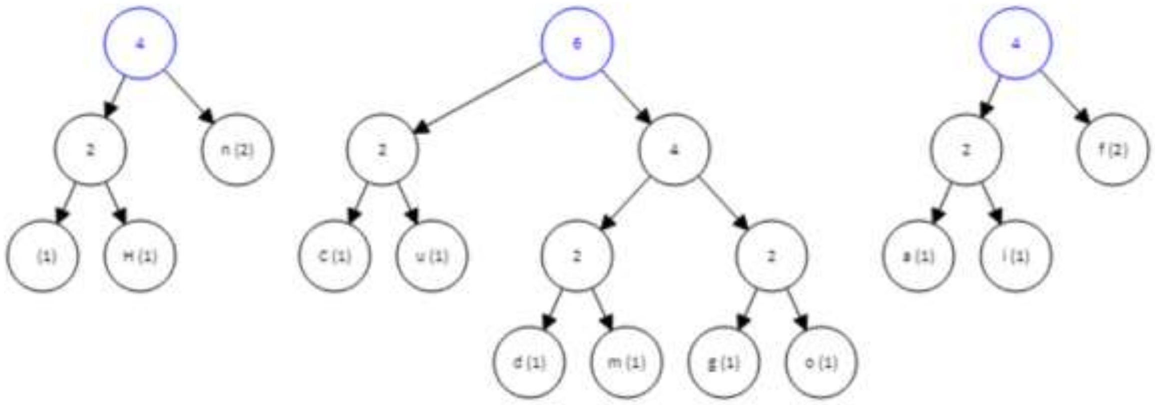
Reinserting the new root node in the priority queue.



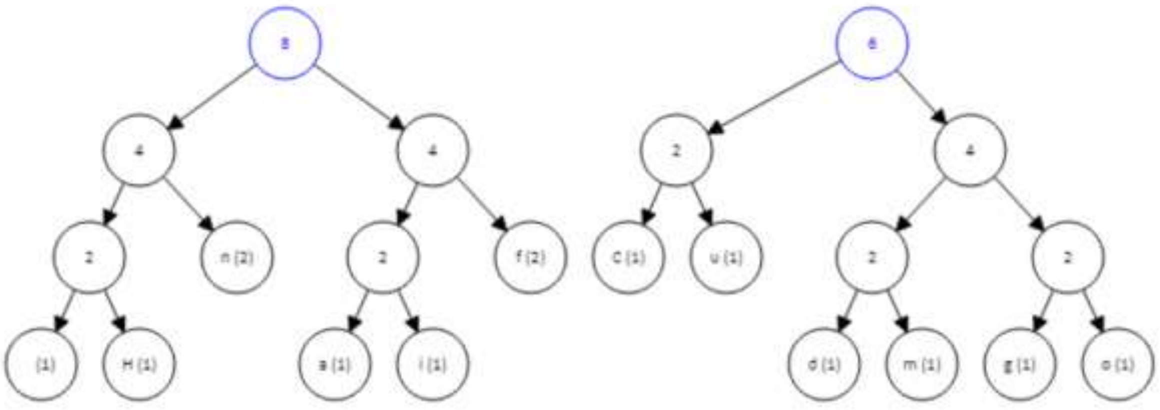
Reinserting the new root node in the priority queue.



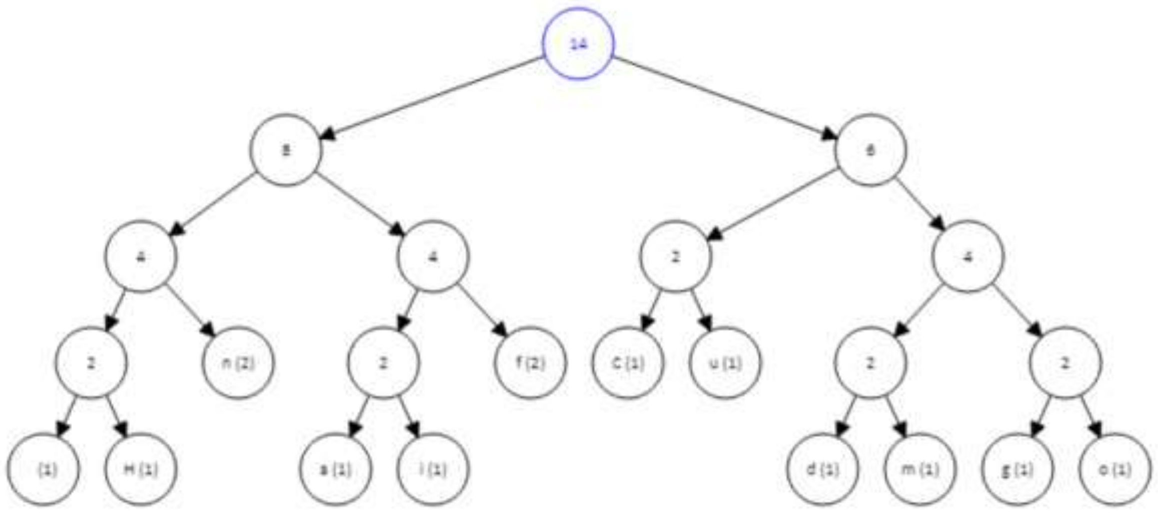
Reinserting the new root node in the priority queue.



Reinserting the new root node in the priority queue.



Reinserting the new root node in the priority queue.



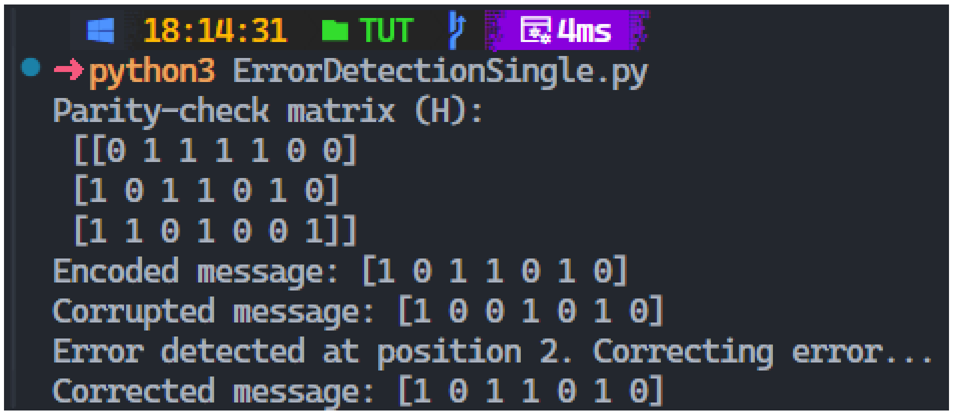
4. **Program to detect and correct single bit error using syndrome calculation. The (n,k) generator matrix will be given by the user**

➔ **CODE :**

```
import numpy as np


def generate_parity_check_matrix(G):
    k, n = G.shape
    r = n - k
    I = G[:, :k]
    P = G[:, k:]
    H = np.concatenate((P.T, np.identity(r, dtype=int)), axis=1)
    return H % 2


def encode_message(message, G):
    return np.dot(message, G) % 2


def add_error(encoded_msg, error_index):
    corrupted_msg = np.copy(encoded_msg)
    corrupted_msg[error_index] = 1 - corrupted_msg[error_index]
    return corrupted_msg


def detect_and_correct(corrupted_msg, H):
    syndrome = np.dot(H, corrupted_msg.T) % 2
    if np.any(syndrome):
        error_index = np.where(np.all(H == syndrome[:, None], axis=0))[0]
        if error_index.size:
            print(f"Error detected at position {error_index[0]}. Correcting
error...")
            corrupted_msg[error_index[0]] = 1 - corrupted_msg[error_index[0]]
```

```python
        else:
            print("Error detected, but unable to correct!")
    else:
        print("No error detected.")
    return corrupted_msg


def main():
    G = np.array(
        [
            [1, 0, 0, 0, 0, 1, 1],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 1, 0, 1, 1, 0],
            [0, 0, 0, 1, 1, 1, 1],
        ]
    )

    H = generate_parity_check_matrix(G)
    print("Parity-check matrix (H):\n", H)

    message = np.array([1, 0, 1, 1])
    encoded_msg = encode_message(message, G)
    print("Encoded message:", encoded_msg)

    error_index = 2
    corrupted_msg = add_error(encoded_msg, error_index)
    print("Corrupted message:", corrupted_msg)

    corrected_msg = detect_and_correct(corrupted_msg, H)
    print("Corrected message:", corrected_msg)


if __name__ == "__main__":
    main()
```

OUTPUT :



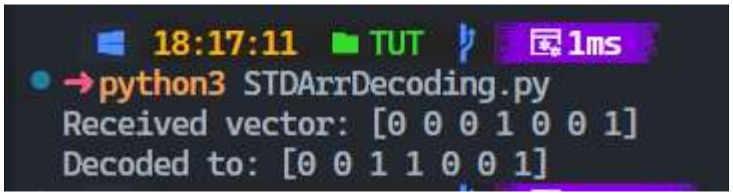
```
    18:14:31  TUT   4ms
➜ python3 ErrorDetectionSingle.py
Parity-check matrix (H):
 [[0 1 1 1 1 0 0]
 [1 0 1 1 0 1 0]
 [1 1 0 1 0 0 1]]
Encoded message: [1 0 1 1 0 1 0]
Corrupted message: [1 0 0 1 0 1 0]
Error detected at position 2. Correcting error...
Corrected message: [1 0 1 1 0 1 0]
```

5. **Program to implement Standard array Decoding for an (n,k) code.**

➔ **CODE :**

```python
import numpy as np


def generate_codewords(G):
    k = G.shape[0]
    messages = [np.array(list(f"{i:0{k}b}"), dtype=int) for i in range(2**k)]

    return [np.dot(msg, G) % 2 for msg in messages]


def generate_error_patterns(n):
    error_patterns = [np.zeros(n, dtype=int)]
    for i in range(n):
        e = np.zeros(n, dtype=int)
        e[i] = 1
        error_patterns.append(e)
    return error_patterns


def form_standard_array(codewords, error_patterns):
    standard_array = []
    for codeword in codewords:
        row = [(codeword + e) % 2 for e in error_patterns]
        standard_array.append(row)
    return standard_array


def find_coset(standard_array, received):
    n = len(standard_array[0])
    for row in standard_array:
        for codeword in row:
            if np.array_equal(received, codeword):
                return row[0]
    return None


def main():
    G = np.array(
        [
            [1, 0, 0, 0, 0, 1, 1],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 1, 0, 1, 1, 0],
            [0, 0, 0, 1, 1, 1, 1],
        ]
    )
    n = G.shape[1]
    codewords = generate_codewords(G)
    error_patterns = generate_error_patterns(n)

    standard_array = form_standard_array(codewords, error_patterns)

    received = (codewords[3] + np.array([0, 0, 1, 0, 0, 0, 0])) % 2
    print("Received vector:", received)

    decoded = find_coset(standard_array, received)
    print("Decoded to:", decoded)


if __name__ == "__main__":
    main()
```

OUTPUT :



```
    18:17:11  TUT  1ms
 → python3 STDArrDecoding.py
   Received vector: [0 0 0 1 0 0 1]
   Decoded to: [0 0 1 1 0 0 1]
```

6. **Case study of any of the cryptography methods given in Module 5.**

➔ **Chinese Remainder Theorem –**

The **Chinese remainder theorem** is commonly employed in large integer computing because it permits a computation bound on the size of the result to be replaced by numerous small integer computations. This **remainder theorem definition** provides an effective solution to major ideal domains.

According to the **Chinese remainder theorem**, every primary ideal domain is actual (expressed in terms of congruences). It has been generalized to any ring using a two-sided ideal formulation.

Formula for Chinese remainder theorem

$$x = \sum_{i=1}^{k} a_i M_i N_i .$$

* 1 (MOD N)

$M = m1 * m2 * m3$

$Xi = MiXi \equiv 1(Mod\ mi)$

Application for Chinese remainder theorem

1. Cryptography: CRT is widely used in public-key cryptography algorithms such as RSA. In RSA encryption and decryption, large prime numbers are used, and CRT provides a way to speed up the computations involved in modular exponentiation by decomposing the modular exponentiation into smaller, independent modular exponentiations.
2. Error Correction Codes: CRT is used in error correction codes such as Reed-Solomon codes. In error correction coding, CRT helps to recover the original message by decoding information from multiple smaller fields.
3. Discrete Fourier Transform (DFT): CRT is used in the Cooley-Tukey Fast Fourier Transform (FFT) algorithm, which decomposes a DFT of composite size into smaller DFTs. This decomposition allows for efficient computation of the DFT, especially for large input sizes.
4. Parallel Processing: CRT is used in parallel computing to break down a problem into smaller independent subproblems that can be solved concurrently. Each processor handles a part of the problem, and CRT is used to combine the solutions from individual processors to obtain the final result.
5. Scheduling and Timetabling: CRT is used in scheduling and timetabling problems to find feasible solutions by decomposing the problem into smaller, independent subproblems. This allows for efficient scheduling of resources, such as classrooms, machines, or workers.
6. Digital Signal Processing (DSP): CRT is used in digital signal processing algorithms for efficient computation of convolution and filtering operations. By decomposing the signal into smaller parts, CRT allows for parallel processing and faster computation.
7. Modular Arithmetic: CRT is used in various applications of modular arithmetic, such as solving systems of linear congruences, solving simultaneous modular equations, and finding solutions to modular exponentiation problems efficiently.