# CRC

def crc\_remainder(input\_bitstring, polynomial, initial\_fill):

# Append zeros to the input bitstring (equal to the degree of the polynomial)

dividend = input\_bitstring + '0' \* (len(polynomial) - 1)

dividend = list(dividend)

polynomial = list(polynomial)

for i in range(len(input\_bitstring)):

if dividend[i] == '1':

for j in range(len(polynomial)):

dividend[i + j] = str(int(dividend[i + j]) ^ int(polynomial[j]))

# Get the remainder (checksum)

remainder = ''.join(dividend)[-len(polynomial) + 1:]

# Append the remainder to the original message

codeword = input\_bitstring + remainder

return remainder, codeword

def crc\_check(codeword, polynomial):

# Calculate the remainder using the received codeword and the same polynomial

remainder, \_ = crc\_remainder(codeword, polynomial, '0' \* ((polynomial) - 1))

return remainder == '0' \* (len(polynomial) - 1)

if \_\_name\_\_ == "\_\_main\_\_":

message = eval(input(" Enter the Message in bit format : "))

polynomial = eval(input(" Enter the Polynomial divisior in bit format : "))

remainder, codeword = crc\_remainder(message, polynomial, '0' \* ((polynomial) - 1))

print(" Original Message:", message)

print(" Polynomial:", polynomial)

print(" Transmitted Codeword:", codeword)

print(" Remainder:", remainder)

print(" Enter the choice \n 1. Error Free Tramsmission \n 2. Simulate Error : ")

choice = eval(input())

if choice == 1:

is\_valid = crc\_check(codeword, polynomial)

else:

# Simulate an error by flipping a bit

error\_position = eval(input("Enter the bit position to introduce Error : "))

codeword = codeword[:error\_position] + ('1' if codeword[error\_position] == '0' else '0') + codeword[error\_position + 1:]

print("Received Codeword with Error:", codeword)

is\_valid = crc\_check(codeword, polynomial)

if is\_valid:

print("No error detected. The message is valid and the message is : ", codeword)

else:

print("Error detected. The message is invalid and the message is : ", codeword)

# DVR

import sys

class Network:

def \_\_init\_\_(self, nodes):

self.nodes = nodes

self.graph = {} # Dictionary to store network topology

self.distance\_vector = {} # Dictionary to store distance vectors

def add\_link(self, node1, node2, cost):

# Add a link between two nodes with a given cost

if node1 not in self.graph:

self.graph[node1] = {}

self.graph[node1][node2] = cost

if node2 not in self.graph:

self.graph[node2] = {}

self.graph[node2][node1] = cost

def initialize\_distance\_vector(self, node):

# Initialize the distance vector for a node

self.distance\_vector[node] = {node: 0}

for n in self.nodes:

if n != node:

self.distance\_vector[node][n] = sys.maxsize

def update\_distance\_vector(self, node):

# Update the distance vector for a node

for dest in self.nodes:

if dest != node:

min\_cost = sys.maxsize

for neighbor in self.graph[node]:

if dest in self.distance\_vector[neighbor]:

cost = self.distance\_vector[neighbor][dest] + self.graph[node][neighbor]

if cost < min\_cost:

min\_cost = cost

self.distance\_vector[node][dest] = min\_cost

def print\_routing\_table(self, node):

# Print the routing table for a node

print(f"Routing table for Node {node}:")

print("Destination\tCost")

for dest, cost in self.distance\_vector[node].items():

if dest != node:

print(f"{dest}\t\t{cost}")

print()

if \_\_name\_\_ == "\_\_main\_\_":

nodes = [1, 2, 3, 4, 5]

network = Network(nodes)

network.add\_link(1, 2, 2)

network.add\_link(1, 3, 2)

network.add\_link(1, 4, 1)

network.add\_link(2, 3, 3)

network.add\_link(2, 1, 2)

network.add\_link(2, 5, 1)

network.add\_link(3, 1, 2)

network.add\_link(3, 4, 4)

network.add\_link(3, 2, 3)

network.add\_link(3, 5, 1)

network.add\_link(4, 1, 1)

network.add\_link(4, 3, 4)

network.add\_link(5, 2, 1)

network.add\_link(5, 3, 1)

for node in nodes:

network.initialize\_distance\_vector(node)

num\_iterations = 6 # Number of iterations to update the distance vectors

for \_ in range(num\_iterations):

for node in nodes:

network.update\_distance\_vector(node)

for node in nodes:

network.print\_routing\_table(node)