1.CRC

def binary\_division(dividend, divisor, n):

remainder = dividend[:9]

for i in range(n):

if remainder[0] == 1:

for j in range(9):

remainder[j] ^= divisor[j]

if i + 9 < n + 8:

remainder = remainder[1:] + [dividend[i + 9]]

return remainder

def main():

n = int(input("Enter the size of the data bits: "))

dividend = [0] \* (n + 8)

print("Enter the data bits:")

for i in range(n):

dividend[i] = int(input())

divisor = [1, 0, 0, 0, 0, 0, 1, 1, 1]

remainder = binary\_division(dividend, divisor, n)

for i in range(8):

dividend[n + i] = remainder[i + 1]

print("Encoded data bits are:", \*dividend)

print("Enter data bits received at receiver's end:")

received = [int(input()) for \_ in range(n + 8)]

received\_remainder = binary\_division(received, divisor, n)

if all(bit == 0 for bit in received\_remainder[:8]):

print("NO ERROR DETECTED....")

else:

print("ERROR DETECTED!!!!")

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

main()

2.hamming code

import math

def calculate\_parity\_bits(encoded\_data, data\_length, redundant\_bits):

for i in range(redundant\_bits):

parity\_bit\_position = 2 \*\* i

for j in range(1, data\_length + redundant\_bits + 1):

if ((j >> i) & 1) == 1:

encoded\_data[parity\_bit\_position] ^= encoded\_data[j]

def find\_error\_position(received\_data, data\_length, redundant\_bits):

error\_position = 0

for i in range(redundant\_bits):

parity\_bit\_position = 2 \*\* i

parity = 0

for j in range(1, data\_length + redundant\_bits + 1):

if ((j >> i) & 1) == 1:

parity ^= received\_data[j]

if parity != 0:

error\_position += parity\_bit\_position

return error\_position

def main():

data\_length = int(input("Enter the length of data bits: "))

redundant\_bits = 0

while (2 \*\* redundant\_bits) < (data\_length + redundant\_bits + 1):

redundant\_bits += 1

print("Enter the data bits:")

data\_bits = [0] + [int(input()) for \_ in range(data\_length)]

encoded\_data = [0] \* (data\_length + redundant\_bits + 1)

data\_index = 1

for i in range(1, data\_length + redundant\_bits + 1):

if (i & (i - 1)) == 0:

continue

encoded\_data[i] = data\_bits[data\_index]

data\_index += 1

calculate\_parity\_bits(encoded\_data, data\_length, redundant\_bits)

print("Encoded data:", \*encoded\_data[1:])

print("Enter the received data bits:")

received\_data = [0] + [int(input()) for \_ in range(data\_length + redundant\_bits)]

error\_position = find\_error\_position(received\_data, data\_length, redundant\_bits)

if error\_position == 0:

print("No error in the data bits.")

else:

print(f"Error found in the data bit. The position of the error bit is: {error\_position}")

received\_data[error\_position] ^= 1

print("The data after error detection and correction is:", \*received\_data[1:])

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

main()

3.subnet

# Function to split a string into parts based on a delimiter

def split(string, delimiter):

return string.split(delimiter)

# Function to get the class of an IP address

def get\_class(ip):

first\_byte = int(split(ip, '.')[0])

if first\_byte < 128:

return 'A'

elif first\_byte < 192:

return 'B'

elif first\_byte < 224:

return 'C'

elif first\_byte < 240:

return 'D'

else:

return 'E'

# Function to get the network mask based on the class and subnet bits

def get\_network\_mask(ip\_class, subnet\_bits):

if ip\_class == 'A':

return f"255.{256 - (1 << (16 - subnet\_bits))}.0.0"

elif ip\_class == 'B':

return f"255.255.{256 - (1 << (8 - subnet\_bits))}.0"

elif ip\_class == 'C':

return f"255.255.255.{256 - (1 << (8 - subnet\_bits))}"

else:

return "Invalid"

# Function to generate the subnet IP address

def generate\_subnet(ip, subnet\_mask):

ip\_parts = list(map(int, split(ip, '.')))

subnet\_mask\_parts = list(map(int, split(subnet\_mask, '.')))

subnet\_ip = '.'.join(str(ip\_parts[i] & subnet\_mask\_parts[i]) for i in range(4))

return subnet\_ip

def main():

ip = input("Enter IP address: ")

subnet\_bits = int(input("Enter subnet bits: "))

ip\_class = get\_class(ip)

print(f"Class of IP address: {ip\_class}")

if ip\_class not in 'ABC':

print("Subnetting is not applicable for this IP class.")

return

subnet\_mask = get\_network\_mask(ip\_class, subnet\_bits)

print(f"Network mask: {subnet\_mask}")

subnet\_ip = generate\_subnet(ip, subnet\_mask)

print(f"Subnet IP address: {subnet\_ip}")

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

main()

5. 1bit stop and wait

a) sender.py

import time

import random

TIMEOUT\_DURATION = 2

FRAME\_COUNT = 5

FRAME\_CORRUPTION\_PROBABILITY = 0.1

def sender():

frame\_number = 0

sent\_frames = 0

while sent\_frames < FRAME\_COUNT:

print(f"Sender: Sending frame {frame\_number}")

time.sleep(random.uniform(0.5, 1.5))

if random.random() < FRAME\_CORRUPTION\_PROBABILITY:

print(f"Sender: Frame {frame\_number} got corrupted in transmission!")

continue

start\_time = time.time()

ack\_received = False

while not ack\_received:

if receiver\_simulator(frame\_number):

print(f"Sender: Received ACK for frame {frame\_number}")

frame\_number = 1 - frame\_number

sent\_frames += 1

ack\_received = True

else:

if time.time() - start\_time > TIMEOUT\_DURATION:

print(f"Sender: Timeout occurred, resending frame {frame\_number}")

break

print("Sender: All frames transmitted successfully.")

def receiver\_simulator(expected\_frame):

if random.random() < 0.1:

print(f"Receiver: Frame {expected\_frame} lost in transmission.")

return False

if random.random() < 0.1:

print(f"Receiver: Received corrupted frame {expected\_frame}. Discarding.")

return False

print(f"Receiver: Received frame {expected\_frame} successfully.")

time.sleep(random.uniform(0.5, 1.0))

if random.random() < 0.1:

print(f"Receiver: ACK for frame {expected\_frame} lost in transmission.")

return False

print(f"Receiver: Sending ACK for frame {expected\_frame}")

return True

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

sender()

b)receiver.py

import random

import time

PACKET\_LOSS\_PROBABILITY = 0.1

FRAME\_CORRUPTION\_PROBABILITY = 0.1

ACK\_LOSS\_PROBABILITY = 0.1

def receiver(expected\_frame):

if random.random() < PACKET\_LOSS\_PROBABILITY:

print(f"Receiver: Frame {expected\_frame} lost in transmission.")

return False

if random.random() < FRAME\_CORRUPTION\_PROBABILITY:

print(f"Receiver: Received corrupted frame {expected\_frame}. Discarding.")

return False

print(f"Receiver: Received frame {expected\_frame} successfully.")

time.sleep(random.uniform(0.5, 1.0))

if random.random() < ACK\_LOSS\_PROBABILITY:

print(f"Receiver: ACK for frame {expected\_frame} lost in transmission.")

return False

print(f"Receiver: Sending ACK for frame {expected\_frame}")

return True

6. n bit stop and wait

a) sender.py

import socket

import time

import random

N = 4

window\_size = 2\*\*N

total\_frames = 10

host = 'localhost'

port = 12345

class Sender:

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

self.window\_start = 0

self.window\_end = window\_size - 1

self.next\_frame\_to\_send = 0

self.sock = sock

def send\_frame(self):

if self.next\_frame\_to\_send <= self.window\_end:

print(f"Sender: Sending frame {self.next\_frame\_to\_send}")

self.sock.sendall(str(self.next\_frame\_to\_send).encode())

return self.next\_frame\_to\_send

else:

print("Sender: Window full, waiting for ACK...")

return None

def receive\_ack(self):

ack = self.sock.recv(1024).decode()

print(f"Sender: Received ACK for frame {ack}")

ack = int(ack)

if self.window\_start <= ack <= self.window\_end:

self.window\_start = ack + 1

self.window\_end = self.window\_start + window\_size - 1

self.next\_frame\_to\_send = self.window\_start

else:

print(f"Sender: ACK {ack} is out of window bounds")

def main():

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.connect((host, port))

sender = Sender(sock)

while sender.next\_frame\_to\_send < total\_frames:

frame = sender.send\_frame()

if frame is not None:

time.sleep(1)

if random.random() < 0.8:

sender.receive\_ack()

else:

print(f"Sender: Frame {frame} lost during transmission.")

sender.next\_frame\_to\_send += 1

time.sleep(1)

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

main()

b) receiver.py

import socket

host = 'localhost'

port = 12345

class Receiver:

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

self.expected\_frame = 0

self.conn = conn

def receive\_frame(self):

frame = self.conn.recv(1024).decode().strip()

if not frame:

print("Receiver: Connection closed by the sender.")

return False

try:

frame = int(frame)

if frame == self.expected\_frame:

print(f"Receiver: Received expected frame {frame}, sending ACK")

self.conn.sendall(str(frame).encode())

self.expected\_frame += 1

else:

print(f"Receiver: Frame {frame} out of order, expected {self.expected\_frame}. Ignoring.")

self.conn.sendall(str(self.expected\_frame - 1).encode())

except ValueError:

print(f"Receiver: Received invalid frame data '{frame}', ignoring.")

return True

def main():

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:

sock.bind((host, port))

sock.listen(1)

print(f"Receiver: Listening on {host}:{port}")

conn, addr = sock.accept()

with conn:

print(f"Receiver: Connected by {addr}")

receiver = Receiver(conn)

while receiver.receive\_frame():

pass

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

main()

7.prims.py

import heapq

def primMST(n, adj):

key = [float('inf')] \* n

parent = [-1] \* n

inMST = [False] \* n

pq = [(0, 0)]

key[0] = 0

while pq:

weight, u = heapq.heappop(pq)

if inMST[u]:

continue

inMST[u] = True

for v, w in adj[u]:

if not inMST[v] and w < key[v]:

key[v] = w

parent[v] = u

heapq.heappush(pq, (key[v], v))

total\_cost = 0

print("Edges in the MST:")

for v in range(1, n):

if parent[v] != -1:

print(f"{parent[v]} - {v} : {key[v]}")

total\_cost += key[v]

else:

print(f"Node {v} is disconnected and not included in the MST.")

print(f"Total cost of MST: {total\_cost}")

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

n = 5

adj = [[] for \_ in range(n)]

adj[0].append((1, 10))

adj[1].append((0, 10))

adj[0].append((2, 6))

adj[2].append((0, 6))

adj[0].append((3, 5))

adj[3].append((0, 5))

adj[1].append((3, 15))

adj[3].append((1, 15))

adj[2].append((3, 4))

adj[3].append((2, 4))

primMST(n, adj)

8.djiksthras.py

import heapq

import sys

def dijkstra(n, adj, src):

dist = [sys.maxsize] \* n

dist[src] = 0

pq = []

heapq.heappush(pq, (0, src))

while pq:

d, u = heapq.heappop(pq)

if d > dist[u]:

continue

for weight, v in adj[u]:

newDist = dist[u] + weight

if newDist < dist[v]:

dist[v] = newDist

heapq.heappush(pq, (dist[v], v))

print(f"Shortest distances from source node {src}:")

for i in range(n):

print(f"Node {i} : ", end="")

if dist[i] == sys.maxsize:

print("Unreachable")

else:

print(dist[i])

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

n = 5

adj = [[] for \_ in range(n)]

adj[0].append((10, 1))

adj[0].append((3, 2))

adj[1].append((1, 2))

adj[1].append((2, 3))

adj[2].append((4, 3))

adj[3].append((2, 4))

adj[2].append((8, 4))

src = 0

dijkstra(n, adj, src)

9.tcp

a) client.py

import socket

def tcp\_client():

host = '127.0.0.1' # Server's address

port = 12345 # Server's port

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as client\_socket:

client\_socket.connect((host, port))

print("Connected to the TCP server...")

while True:

message = input("Enter message to send (or 'exit' to quit): ")

if message.lower() == 'exit':

break

client\_socket.sendall(message.encode())

data = client\_socket.recv(1024)

print(f"Received from server: {data.decode()}")

tcp\_client()

b) server.py

import socket

def tcp\_server():

host = '127.0.0.1' # localhost

port = 12345 # Port to listen on

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((host, port))

server\_socket.listen(5)

print(f"TCP Server listening on {host}:{port}...")

conn, addr = server\_socket.accept()

with conn:

print(f"Connected by {addr}")

while True:

data = conn.recv(1024)

if not data:

break

print(f"Received from client: {data.decode()}")

conn.sendall(data) # Echo back the received data

tcp\_server()

2) UDP

a)client.py

import socket

def udp\_client():

host = '127.0.0.1' # Server's address

port = 12345 # Server's port

with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as client\_socket:

print("Connected to the UDP server...")

while True:

message = input("Enter message to send (or 'exit' to quit): ")

if message.lower() == 'exit':

break

client\_socket.sendto(message.encode(), (host, port))

data, \_ = client\_socket.recvfrom(1024)

print(f"Received from server: {data.decode()}")

udp\_client()

b)server.py

import socket

def udp\_server():

host = '127.0.0.1' # localhost

port = 12345 # Port to listen on

with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as server\_socket:

server\_socket.bind((host, port))

print(f"UDP Server listening on {host}:{port}...")

while True:

data, addr = server\_socket.recvfrom(1024)

print(f"Received from {addr}: {data.decode()}")

server\_socket.sendto(data, addr) # Echo back the received data

udp\_server()