```
#HAMMING CODE
import math
def calc_redundant_bits(m):
    r = 0
    while (2 ** r) < (m + r + 1):
        r += 1
    return r
def insert_parity_positions(data, r):
    j = 0
    k = 0
    res = []
    m = len(data)
    for i in range(1, m + r + 1):
        if i == 2 ** j:
            res.append(-1)
            j += 1
        else:
            res.append(int(data[k]))
            k += 1
    return res
def calc_parity_bits(arr, r):
    n = len(arr)
    parity = {}
    for i in range(r):
        pos = 2 ** i
        val = 0
        for j in range(1, n + 1):
            if (j & pos) == pos:
                if arr[j - 1] != -1:
                    val ^= arr[j - 1]
        arr[pos - 1] = val
        parity[pos] = val
    return arr, parity
m = int(input("Enter the number of bits in a message: "))
message = input("Enter the message in binary numbers: ")
r = calc_redundant_bits(m)
print("Number of parity bits are:", r)
print("Number of bits in the message:", m + r)
arr = insert_parity_positions(message[::-1], r)
arr, parity = calc_parity_bits(arr, r)
print("Values of parity bits:", end=" ")
parity_str = []
for p in sorted(parity.keys()):
    parity_str.append(f"R{p} = {parity[p]}")
print(", ".join(parity_str))
print("Message sent by the sender:", end=" ")
for bit in arr[::-1]:
    print(bit, end=" ")
print()
```

```
#CRC
def xor(a, b):
   """XOR two bit-strings a and b (same length). Return result excluding first bit."""
   res = []
   for i in range(1, len(b)):
      res.append('0' if a[i] == b[i] else '1')
   return ''.join(res)
def binary_division(dividend, divisor):
   """Perform mod-2 division and return remainder of length len(divisor)-1."""
   pick = len(divisor)
   tmp = dividend[0:pick]
   while pick < len(dividend):
      if tmp[0] == '1':
         tmp = xor(divisor, tmp) + dividend[pick]
      else:
         tmp = xor('0' * pick, tmp) + dividend[pick]
      pick += 1
   if tmp[0] == '1':
      tmp = xor(divisor, tmp)
   else:
      tmp = xor('0' * len(divisor), tmp)
   return tmp
def main():
   n = int(input("Enter the size of dataword (in bits): "))
   k = int(input("Enter the size of codeword (in bits): "))
   d1 = int(input("Enter the number of bits in divisor: "))
   r = k - n
   if r != d1 - 1:
      print(f"\nError: Number of redundant bits (r={r}) must be one less than the divisor size
(d1={d1}).")
      print("Because r = k-n and number of bits in divisor must be r+1.")
      return
   divisor = input(f"Enter the {d1}-bit divisor: ").replace(" ", "")
   print(f"divisor is:{divisor}")
   dataword = input(f"Enter the {n}-bit dataword: ").replace(" ", "")
   augmented = dataword + ('0' * r)
   print(f"Augmented dataword is:{augmented}")
   crc = binary_division(augmented, divisor)
   print(f"CRC code is:{crc}")
   sent = dataword + crc
   print(f"Sent Codeword is:{sent}")
   received = input("Enter received Codeword: ").replace(" ", "")
   recv_crc = binary_division(received, divisor)
   print(f"CRC code is:{recv_crc}")
   if '1' in recv_crc:
      print("Errors in the received Codeword")
   else:
      print("No errors in the received Codeword")
if __name__ == "__main__":
   main()
```

```
#STOP AND WAIT
import time
def stop_and_wait(packets):
    print("\n--- STOP AND WAIT ARQ ---")
    for i in range(packets):
        print(f"Sender: Sending packet {i}")
        time.sleep(0.2)
        if i % 2 == 0:
            print(f"Packet {i} lost, sender times out and retransmits...")
            time.sleep(0.2)
            print(f"Sender: Retransmitting packet {i}")
            time.sleep(0.1)
        print(f"Receiver: Received packet {i}, sending ACK {i}")
        time.sleep(0.1)
if __name__ == "__main__":
    try:
        n = int(input("Enter number of packets: "))
        if n <= 0:
            raise ValueError("Must be positive")
    except Exception as e:
        print("Invalid input:", e)
    else:
        stop_and_wait(n)
```

```
#GO BACK N
import time
def go_back_n(packets, window_size):
    print("\n--- GO BACK N ---")
    if window_size < 1:
        window_size = 1
    base = 0
    while base < packets:
        end = min(base + window_size, packets)
        to_send = list(range(base, end))
        print(f"Sender: Sending packets {to_send}")
        time.sleep(0.25)
        if len(to_send) > 1:
            loss_packet = to_send[1]
            print(f"Receiver: Packet {loss_packet} lost. Receiver discards out-of-order packets.")
            time.sleep(0.2)
            print(f"Sender: Retransmitting from packet {loss_packet}")
            base = loss_packet
        else:
            print(f"Receiver: All packets {to_send} received, sending cumulative ACK {to_send[-1]}")
            base = end
        time.sleep(0.2)
if __name__ == "__main__":
    try:
        n = int(input("Enter number of packets: "))
        ws = int(input("Enter window size: "))
        if n <= 0 or ws <= 0:
            raise ValueError("Values must be positive")
    except Exception as e:
        print("Invalid input:", e)
    else:
        go_back_n(n, ws)
```

```
import time
def selective_repeat(packets, window_size):
    print("\n--- SELECTIVE REPEAT ---")
    if window_size < 1:
        window_size = 1
    received = [False] * packets
    base = 0
    while base < packets:
        end = min(base + window_size, packets)
        to_send = list(range(base, end))
        print(f"Sender: Sending packets {to_send}")
        time.sleep(0.25)
        loss_packet = to_send[-1]
        for p in to_send:
            if p == loss_packet:
                print(f"Receiver: Packet {p} lost")
            else:
                if not received[p]:
                    print(f"Receiver: Received packet {p}, sending ACK {p}")
                    received[p] = True
            time.sleep(0.08)
        print(f"Sender: Retransmitting packet {loss_packet}")
        time.sleep(0.15)
        print(f"Receiver: Received packet {loss_packet}, sending ACK {loss_packet}")
        received[loss_packet] = True
        while base < packets and received[base]:
            base += 1
        time.sleep(0.15)
if __name__ == "__main__":
    try:
        n = int(input("Enter number of packets: "))
        ws = int(input("Enter window size: "))
        if n <= 0 or ws <= 0:
            raise ValueError("Values must be positive")
    except Exception as e:
        print("Invalid input:", e)
    else:
        selective_repeat(n, ws)
```

```
#BYTE STUFFING
def byte_stuff(data):
   stuffed=""
   for ch in data:
      if ch=='E' or ch=='F':
         stuffed+='E'
      stuffed+=ch
   return stuffed

def byte_unstuff(stuffed):
   unstuffed=""
   i=0
   while i<len(stuffed):
      if stuffed[i]=='E':
         if i+1<len(stuffed) and stuffed[i+1] in ('E','F'):
            unstuffed+=stuffed[i+1]
            i+=2
         else:
            print("Data cannot be unstuffed, data should contain E or F after E character.")
            return None
      else:
         unstuffed+=stuffed[i]
         i+=1
   return unstuffed
if __name__=="__main__":
   data=input("Enter the contents of data: ")
   stuffed_frame=byte_stuff(data)
   print("Frame after byte stuffing:",stuffed_frame)
   stuffed_input=input("Enter the byte stuffed data to unstuff: ")
   unstuffed_frame=byte_unstuff(stuffed_input)
   if unstuffed_frame is not None:
      print("Unstuffed frame:",unstuffed_frame)
```

```
#bit stuffing
def bit_stuff(data):
    count,stuffed=0,""
    for bit in data:
        stuffed+=bit
        if bit=='1':
            count+=1
            if count==5:
                stuffed+='0'
                count=0
        else:
            count=0
    return stuffed

def bit_unstuff(data):
    count,unstuffed=0,""
    i=0
    while i<len(data):
        unstuffed+=data[i]
        if data[i]=='1':
            count+=1
            if count==5:
                i+=1
                count=0
        else:
            count=0
        i+=1
    return unstuffed

data=input("Enter bits for bit stuffing: ")
stuffed=bit_stuff(data)
print("Contents after bit stuffing:",stuffed)
stuffed_data=input("Enter the bit stuffed data to unstuff: ")
print("Unstuffed bits:",bit_unstuff(stuffed_data))
```