Program No: 1	**Roll Number:** 23071A0583	**Date:** 30/7/25
Program Title: Bit Stuffing		

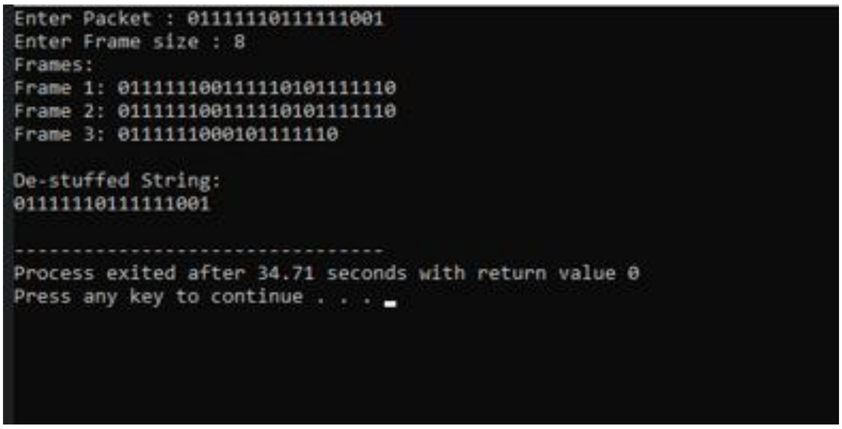
Aim: To implement Bit Stuffing and Bit De-Stuffing with framing.

Description: This C++ program demonstrates the concept of bit stuffing and de-stuffing along with framing, which is used to ensure that certain patterns in data do not interfere with frame boundaries during transmission. The program takes a binary data string and a frame size as input. While processing, it inserts a 0 after every sequence of five consecutive 1s to avoid confusion with frame delimiters. The stuffed data is then divided into frames of the given size, and each frame is enclosed with a predefined flag sequence 01111110 at both the start and end to clearly indicate the frame boundaries. On the receiving side, the program removes these flags, combines the frame data, and applies bit de-stuffing to eliminate the extra 0s inserted during transmission, thus retrieving the original data.

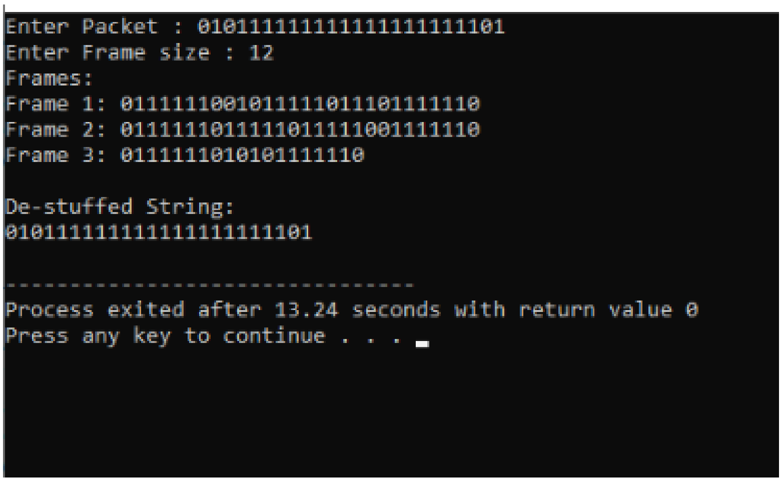
Program:

```
#include<bits/stdc++.h>
using namespace std;
int main() {
     int fs ;
    string s;
    int c = 0;
    cout << "Enter Packet : ";
    cin >> s;
    cout << "Enter Frame size : ";
    cin >> fs;
    vector<string> vs;
    string ss = "", flag = "01111110";
    for (int i = 0; i < s.size(); i++) {
        if (ss.size() == fs) {
            vs.push_back(ss);
            ss = "";
            c = 0;
        }

         if (c == 5) {
            ss += '0';
            c = 0;
            i--;
            continue;
        }
        if (s[i] == '1') {
            ss += '1';
            c++;
        } else {
            ss += '0';
            c = 0;
        } }
    if (ss.size())
        vs.push_back(ss);
    cout << "Frames:" << endl;
    for (int i = 0; i < vs.size(); i++) {
        string frame = flag + vs[i] + flag;
        cout << "Frame " << i + 1 << ": " << frame << endl;
    }
    cout << endl;
    string ds = "";
    for (int i = 0; i < vs.size(); i++) {
        string chunk = vs[i];
        c = 0;
        for (int j = 0; j < chunk.size(); j++) {
            char y = chunk[j];
            if (y == '1') {
                c++;
                ds += '1';
            } else {
                if (c == 5) {
                    c = 0;
                    continue;
                }
                ds += '0';
                c = 0;
            }  }
    }
    cout << "De-stuffed String:" << endl;
    cout << ds << endl;
    return 0;
}
```

Input/Output:



```
Enter Packet : 01111110111111001
Enter Frame size : 8
Frames:
Frame 1: 011111100111110101111110
Frame 2: 011111100111110101111110
Frame 3: 0111111000101111110

De-stuffed String:
01111110111111001


--------------------------------
Process exited after 34.71 seconds with return value 0
Press any key to continue . . .
```



```
Enter Packet : 010111111111111111111101
Enter Frame size : 12
Frames:
Frame 1: 0111111001011111011101111110
Frame 2: 0111111011111011111001111110
Frame 3: 0111111010101111110

De-stuffed String:
010111111111111111111101

--------------------------------
Process exited after 13.24 seconds with return value 0
Press any key to continue . . .
```

Manual calculations:

Output 1:

- Input data= 01111110111111001, Frame size=8, Flag=01111110
- Split Frames (by frame size=8):
  01111110,11111100,1
- Bit Stuffing:
  Insert 0 after every sequence of five consecutive 1s:
  Frames after bit stuffing:
  01111101,01111101,001
- Add Flags:
  F1: 011111100111110101111110
  F2:011111100111110101111110
  F3:0111111000101111110
- Receiver: Remove flags
  DeStuff= 01111110111111001

Output 2:

- Input data=010111111111111111111101, Frame Size=12 , Flag=01111110
- Split Frames(by frame Size=12):
  010111111111, 111111111101
- Bit Stuffing:
  Insert 0 after every sequence of five consecutive 1s:
  Frames after bit stuffing:
  010111110111,111110111110,101
- Add Flags:
  F1:0111111001011111011101111110
  F2:0111111011111011111001111110
  F3:0111111010101111110
- Receiver: remove flags
  Destuff= 010111111111111111111101

Program No: 2	**Roll Number:** 23071A0583	**Date:** 13/8/25
Program Title: Character Stuffing		

Aim: To implement the data link layer framing method: Character Stuffing

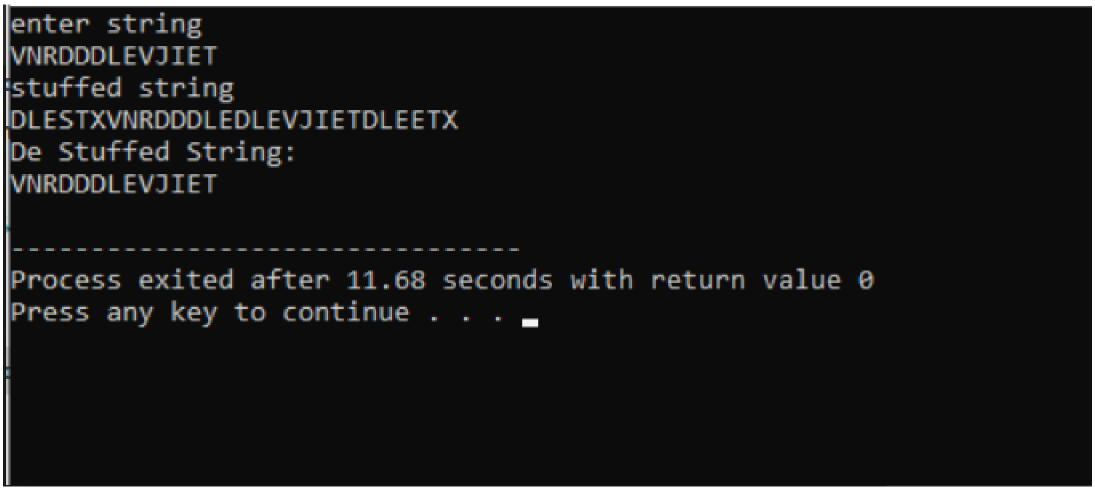
Description: This C++ program demonstrates the concept of character stuffing and de-stuffing using sentinel characters. It takes a string as input and encloses it between the special delimiters "DLESTX" at the start and "DLEETX" at the end to mark the frame boundaries. During stuffing, if the substring "DLE" (or "dle") is found inside the data, it is replaced with "DLEDLE" (or "dledle") to avoid confusion with the delimiters. On the receiving side, the program de-stuffs the data by scanning between the start and end markers, converting "DLEDLE" back into "DLE" (and "dledle" back into "dle"). Finally, it reconstructs the original data and prints it, ensuring that the delimiters are preserved for framing without altering the actual message content.

Program:

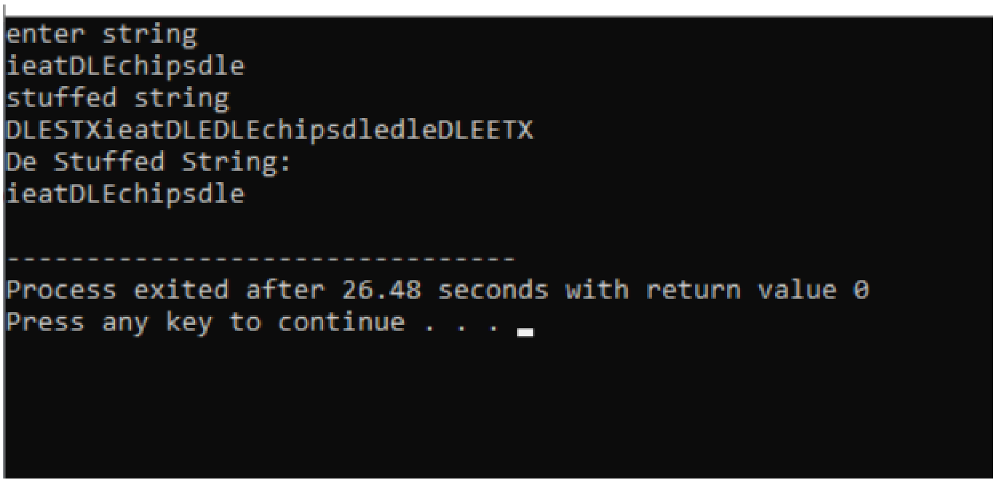
```
#include<bits/stdc++.h>
using namespace std;
string stuffed(string s){
string res="";
res+="DLESTX";
for(int i=0;i<s.size();i++){
if(i+2<s.size()&&s[i]=='D'&&s[i+1]=='L'&&s[i+2]=='E'){
res+="DLEDLE";
i+=2;
}
else if(i+2<s.size()&&s[i]=='d'&&s[i+1]=='l'&&s[i+2]=='e'){
res+="dledle";
i+=2;
}
else{
res+=s[i];
}
}
res+="DLEETX";
return res;
}
string destuff(string s){
string res="";
int n=s.size();
for(int i=6;i<n-6;i++){
if(i+5<n&&s.substr(i,6)=="DLEDLE"){
```

```
res+="DLE";
i+=5;
}else if(i+5<n&&s.substr(i,6)=="dledle"){
res+="dle";
i+=5;
}
else{
res+=s[i];
}
}
return res;
}
int main(){
cout << "enter string" << endl;
string s;
cin >> s;
string stuff_str=stuffed(s);
cout << "stuffed string" << endl;
cout << stuff_str << endl;
string de_stuff=destuff(stuff_str);
cout << "De Stuffed String:" << endl;
cout << de_stuff << endl;
}
```

Input/Output:



```
enter string
VNRDDDLEVJIET
stuffed string
DLESTXVNRDDDLEDLEVJIETDLEETX
De Stuffed String:
VNRDDDLEVJIET

--------------------------------
Process exited after 11.68 seconds with return value 0
Press any key to continue . . .
```



```
enter string
ieatDLEchipsdle
stuffed string
DLESTXieatDLEDLEchipsdledleDLEETX
De Stuffed String:
ieatDLEchipsdle

--------------------------------
Process exited after 26.48 seconds with return value 0
Press any key to continue . . .
```

Manual calculations:

Output 1:

Data: VNRDDDLEVJIET , **Start flag:** DLESTX , **End flag**: DLEETX
Byte (Character) stuffing:

- Found DLE → replace with DLEDLE (case-sensitive).
- Add start and end flags.

Stuffed string: DLESTXVNRDDDDLEDLEVJIETDLEETX
Byte destuffing:

- Remove start flag DLESTX and end flag DLEETX.
- Replace every DLEDLE with DLE.

De-stuffed string: VNRDDDLEVJIET

Output 2:

Data: ieatDLEchipsdle, **Start flag:** DLESTX , **End flag**: DLEETX
Byte (Character) stuffing:

- Found DLE → replace with DLEDLE (case-sensitive).
- Add start and end flags.

Stuffed string: DLESTXieatDLEDLEchipsdledleDLEETX
Byte destuffing:

- Remove start flag DLESTX and end flag DLEETX.
- Replace every DLEDLE with DLE.

De-stuffed string: ieatDLEchipsdle

Program No: 3	**Roll Number:** 23071A0570	**Date:** 10/9/25
Program Title: Cyclic Redundancy Check (CRC) Error Detection		

Aim: To implement a program that performs Cyclic Redundancy Check (CRC) encoding and error detection.

Description: This program implements Cyclic Redundancy Check (CRC), an error-detection technique used in data communication. The sender encodes the binary data using a generator key by appending zeros and performing modulo-2 division to generate a codeword. The remainder from the division is attached to the original data to form the encoded message. At the receiver side, the received codeword is again divided by the same key. If the remainder is zero, the data is correct; otherwise, an error is detected.

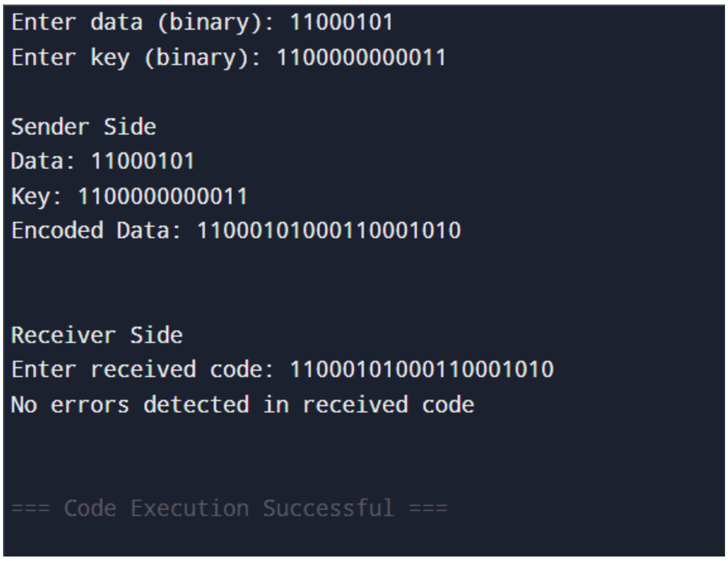
Program:

```
#include <bits/stdc++.h>
using namespace std;
string findXor(string a, string b) {
    int n = b.length();
    string r = "";
    for (int i = 1; i < n; i++) {
        if (a[i] == b[i]){
            r += "0"  }
        else{
            r+= "1"; }
    }
    return r;
}
string mod2div(string dividend, string divisor) {
    int n = dividend.length();
    int pick = divisor.length();
    string tmp = dividend.substr(0, pick);
    while (pick < n) {
        if (tmp[0] == '1'){
            tmp = findXor(divisor, tmp) + dividend[pick]; }
        else{
            tmp = findXor(string(pick, '0'), tmp) + dividend[pick]; }
        pick++;
    }
    if (tmp[0] == '1') {
        tmp = findXor(divisor, tmp); }
    else {
        tmp = findXor(string(pick, '0'), tmp);
    }
    return tmp;
}
```

```
string encodeData(string data, string key) {
    int n = key.length();
    string paddedData = data + string(n - 1, '0');
    string remainder = mod2div(paddedData, key);
    return data + remainder;
}
int receiver(string code, string key) {
    string remainder = mod2div(code, key);
    return (remainder.find('1') == string::npos) ? 1 : 0;
}
int main() {
    string d, k, r;
    cout << "Enter data (binary): ";
    cin >> d;
    cout << "Enter key (binary): ";
    cin >> k;
    cout << "\nSender Side\n";
    cout << "Data: " << d << endl;
    cout << "Key: " << k << endl;
    string c = encodeData(d, k);
    cout << "Encoded Data: " << c << endl << endl;
    cout << "\nReceiver Side\n";
    cout << "Enter received code: ";
    cin >> r;
    if (receiver(r, k)) {
        cout << "No errors detected in received code" << endl; }
    else {
        cout <<"Error detected in received code "<< endl;
}
    return 0;
}
```

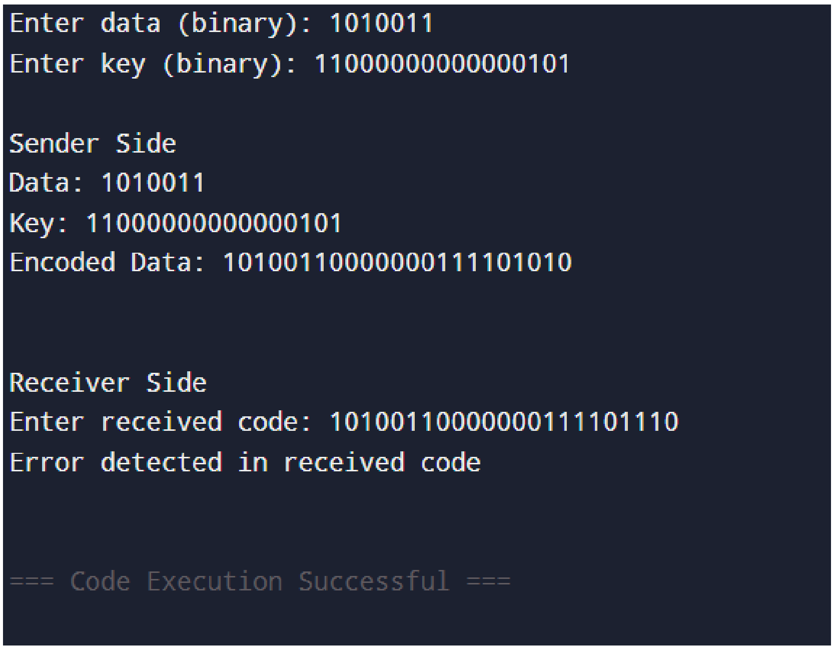
Input/Output:

CRC-12



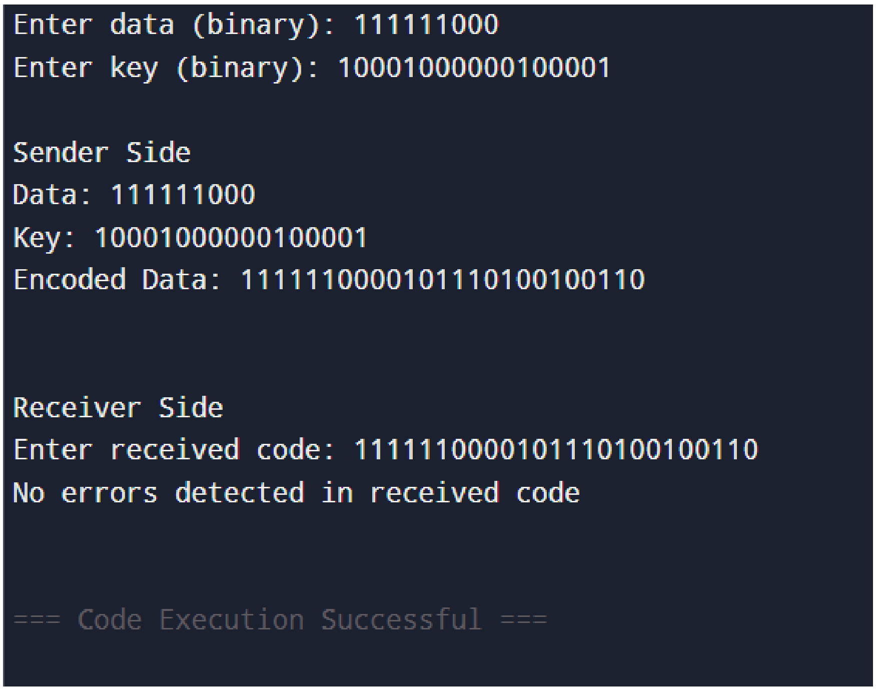
```
Enter data (binary): 11000101
Enter key (binary): 1100000000011

Sender Side
Data: 11000101
Key: 1100000000011
Encoded Data: 11000101000110001010



Receiver Side
Enter received code: 11000101000110001010
No errors detected in received code



=== Code Execution Successful ===
```

CRC-16



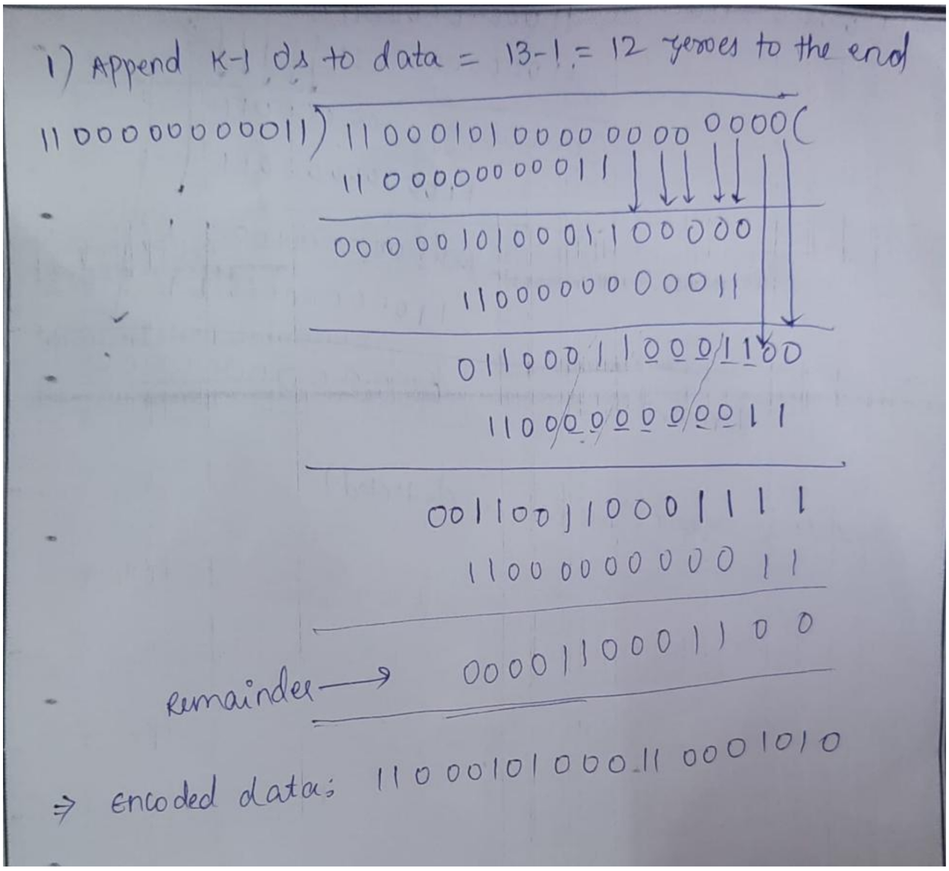
```
Enter data (binary): 1010011
Enter key (binary): 11000000000000101

Sender Side
Data: 1010011
Key: 11000000000000101
Encoded Data: 10100110000000111101010


Receiver Side
Enter received code: 10100110000000111101110
Error detected in received code


=== Code Execution Successful ===
```

CCITT



```
Enter data (binary): 111111000
Enter key (binary): 10001000000100001

Sender Side
Data: 111111000
Key: 10001000000100001
Encoded Data: 1111110000101110100100110


Receiver Side
Enter received code: 1111110000101110100100110
No errors detected in received code


=== Code Execution Successful ===
```

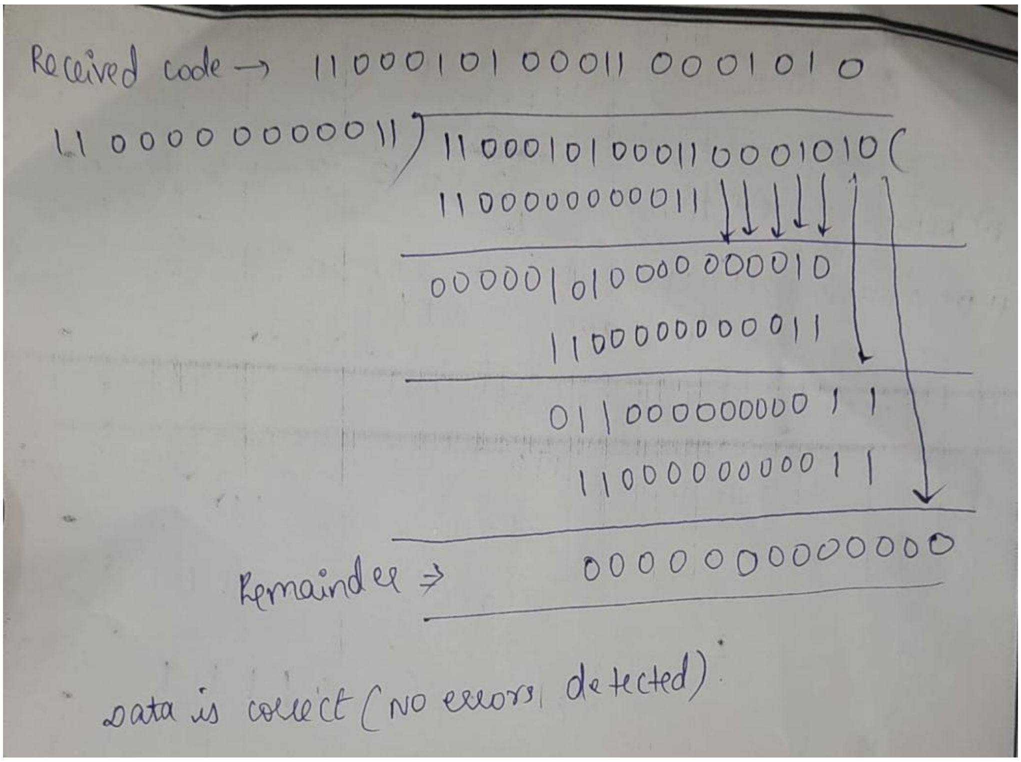
Manual calculations:



1) Append K-1 0s to data = 13-1 = 12 zeroes to the end

```
1100000000011 ) 11000101 00000000 0000(
                1100000000011
                000001010001100000
                     1100000000011
                     0110001100011100
                      1100000000011
                     0011001100011111
                       1100000000011
                     0000110001100
Remainder →          0000110001100
```

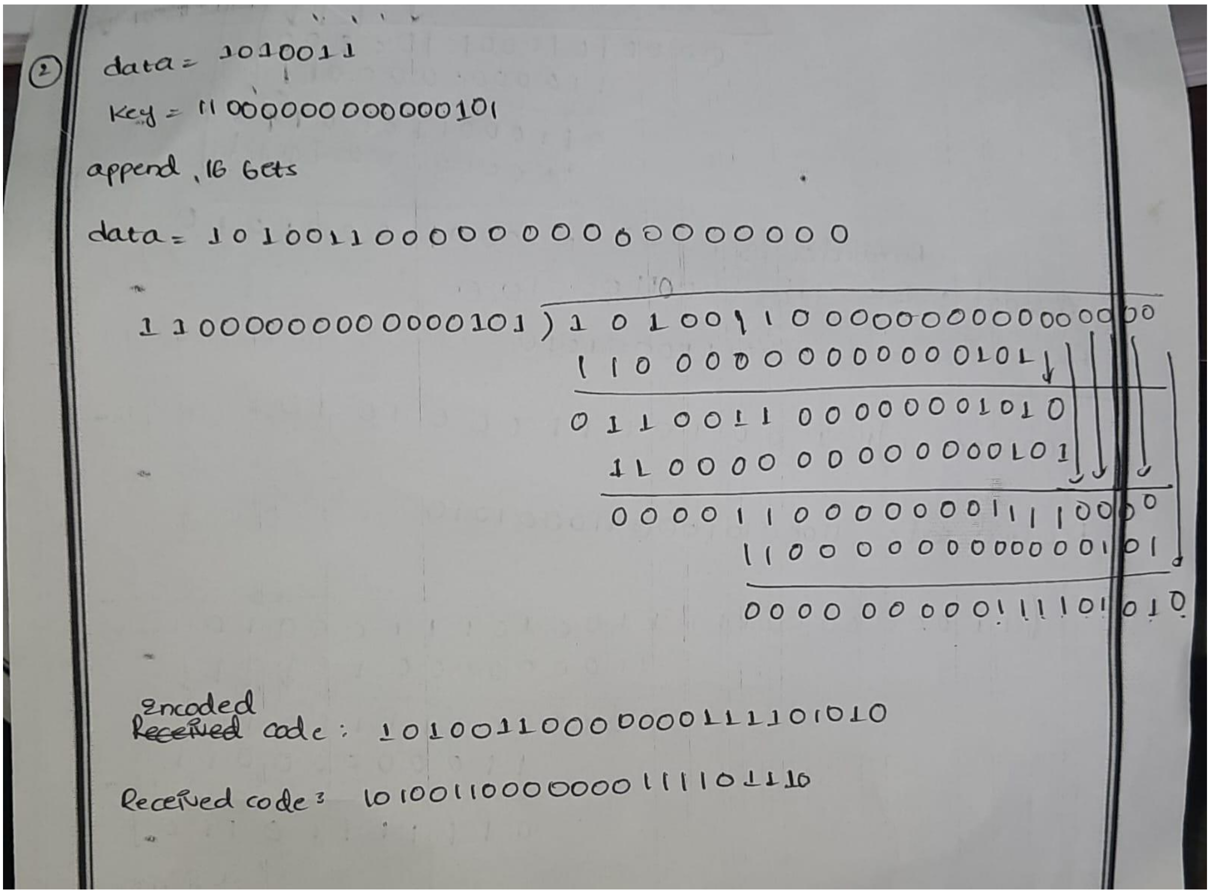
⇒ encoded data: 11000101000.11 0001010



Received code → 11000101 00011 0001010

```
1100000000011 ) 11000101000110001010(
                1100000000011
                00000101000000010
                     1100000000011
                     0110000000011
                      1100000000011
Remainder ⇒          0000000000000
```

data is correct (No errors detected)



(2) data = 1010011

Key = 11000000000000101

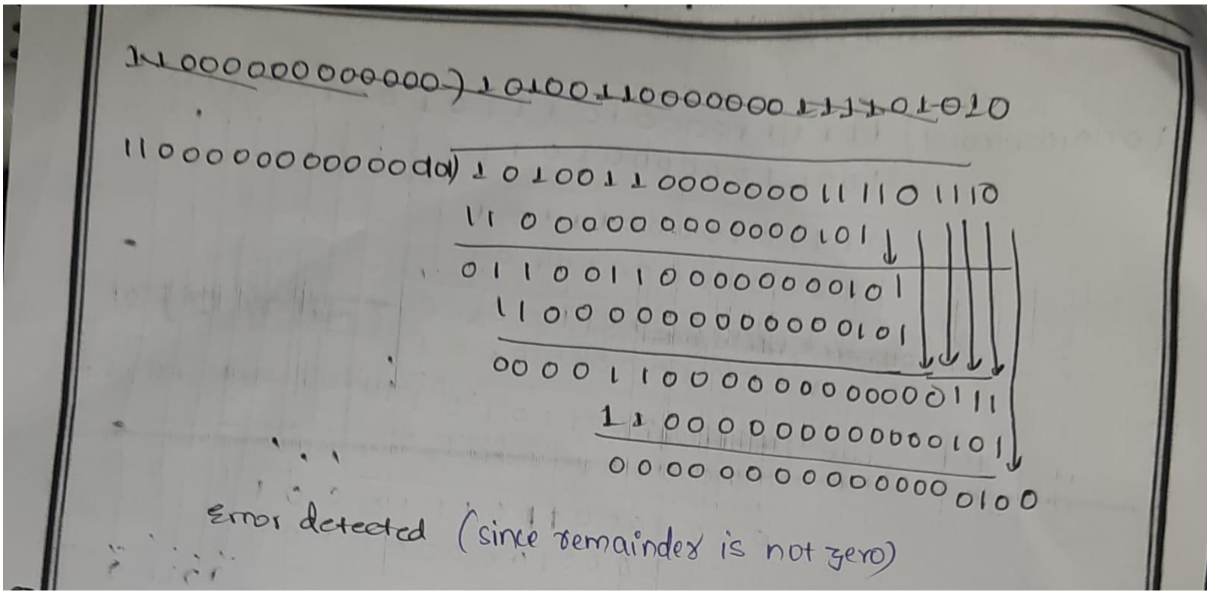
append, 16 bits

data = 10100110000000000000000

```
                       0
11000000000000101 ) 10100110000000000000000
                    110000000000000101
                    ------------------
                    011001100000001010
                     110000000000000101
                     ------------------
                     00001100000000111100000
                         11000000000000101
                         -----------------
                         0000000000111101010
```

Encoded ~~Received~~ code : 10100110000000011111101010

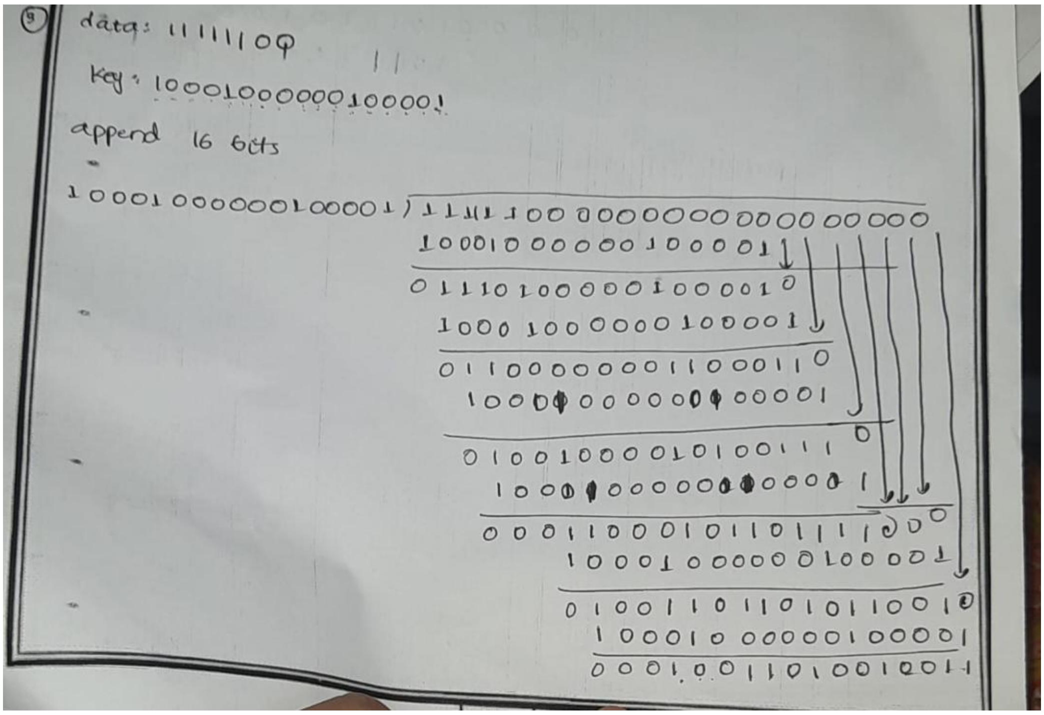
Received code : 1010011000000001111101110



~~110000000000000) 10100110000000 1111101010~~

```
11000000000000101) 1010011000000001111101110
                   110000000000000101
                   ------------------
                   011001100000000101
                    11000000000000101
                    -----------------
                    0000110000000000000111
                         11000000000000101
                         -----------------
                         00000000000000000100
```

Error detected (since remainder is not zero)

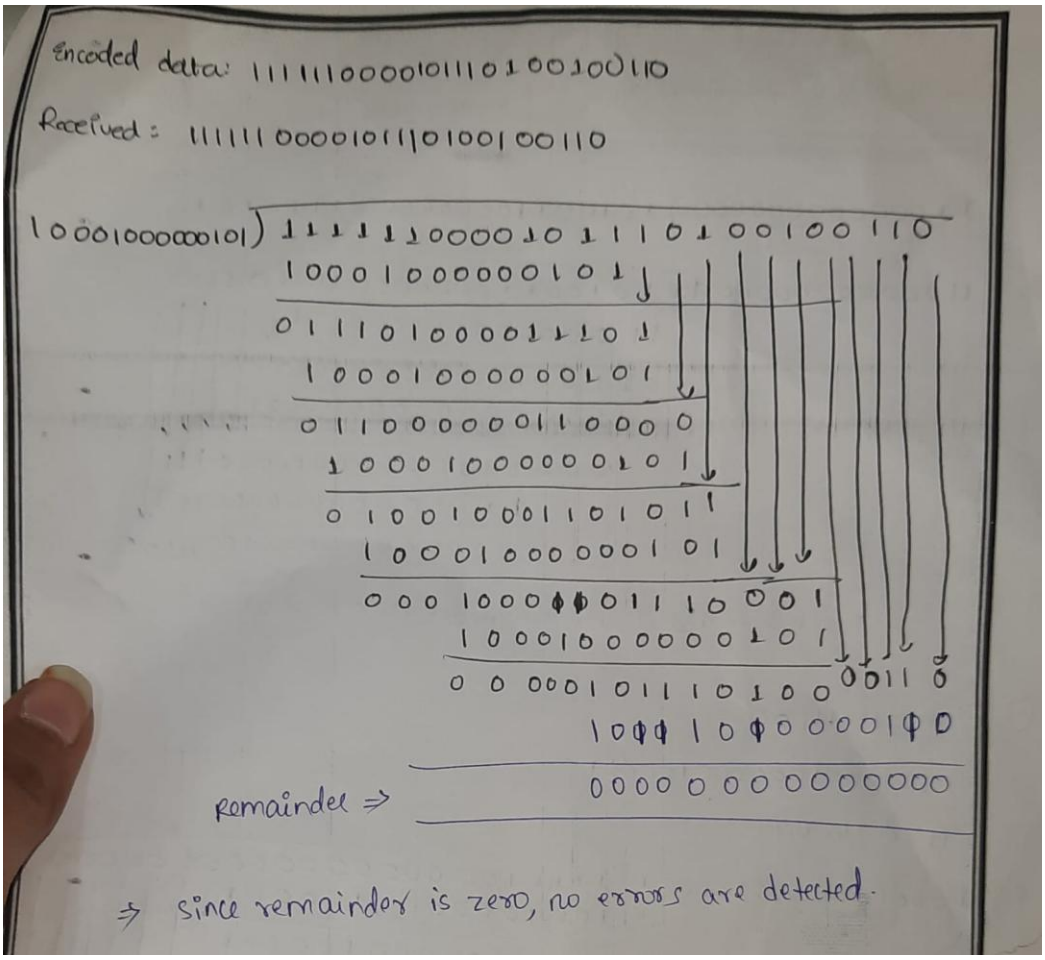


(9) data: 1111110φ

Key: 10001000000100001

append 16 bits

```
10001000000100001) 111111000000000000000000
                   10001000000100001
                   0111010000001000010
                    10001000000100001
                    0110000000011000110
                     1000φ000000φ00001
                     0100100000101001110
                      1000φ000000φ00001
                      000110001011011111000
                         10001000000100001
                         010011011010110010
                          10001000000100001
                          000100111010010011
```



Encoded data: 1111110000101110100100110

Received: 1111110000101110100100110

```
100010000000101) 1111110000101110100100110
                 100010000000101
                 011101000011101
                  100010000000101
                  0110000001100000
                   100010000000101
                   0100100011010111
                    100010000000101
                    0001000φφ01110001
                       100010000000101
                       0000010111010000110
                             10φφ10φ000001φ0
                             0000000000000000
```

Remainder ⇒

⇒ since remainder is zero, no errors are detected.