Problem-1 (1a & 1b)

Design and Implement a program to handle variable length frames in Data Link Layer.

Framing:

Datalink layer packs the bits into frames, so that each frame is distinguishable from other. Framing in datalink layer separates a message from one source to a destination, or from other messages to other destination, by adding a sender address and a destination address.

Fixed-Size Framing

In fixed-size framing, there is no need for defining the boundaries of the frames; the size itself can be used as a delimiter.

Variable-Size Framing

In variable-size framing, we need a way to define the end of the frame and the beginning of the next. Historically, two approaches were used for this purpose: **a character-oriented approach and a bit-oriented approach.**

To separate one frame from the next, an 8-bit (or 1-byte) flag is added at the beginning and the end of a frame. **But the problem** with that is, any pattern used for the flag could also be part of the information. So, there are two ways to overcome this problem.

The header, which normally carries the source and destination addresses and other control information

And the trailer, which carries error detection or error correction redundant bits, are also multiples of 8 bits.

To separate one frame from the next, an 8-bit (1-byte) flag is added at the beginning and the end of a frame.

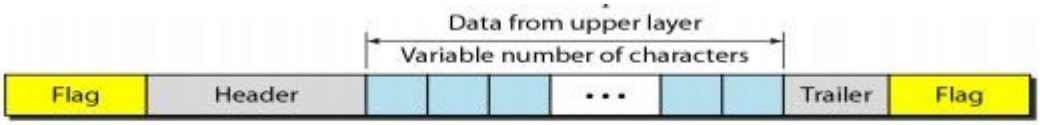


Figure: *A frame in a character-oriented protocol*

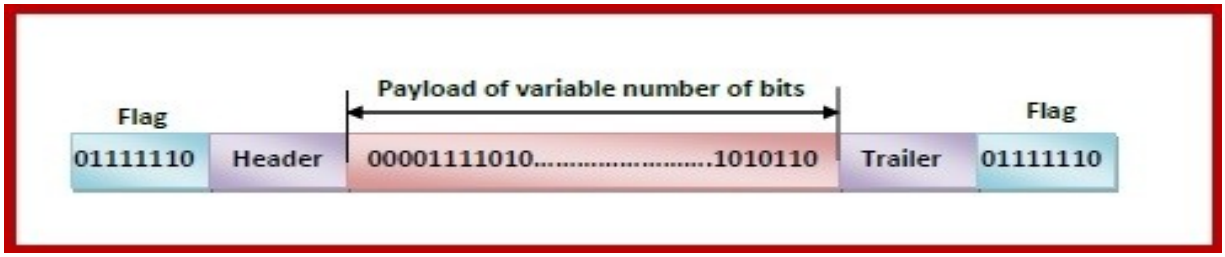
A frame is a digital data transmission unit in computer networking and telecommunication. In packet switched systems, a frame is a simple container for a single network packet. In other telecommunications systems, a frame is a repeating structure supporting time-division multiplexing.

In the OSI model of computer networking, a frame is the protocol data unit at the data link layer. ... A frame is "the unit of transmission in a link layer protocol, and consists of a link layer header followed by a packet." Each frame is separated from the next by an interframe gap.

Frame in a Bit -

A frame has the following parts −

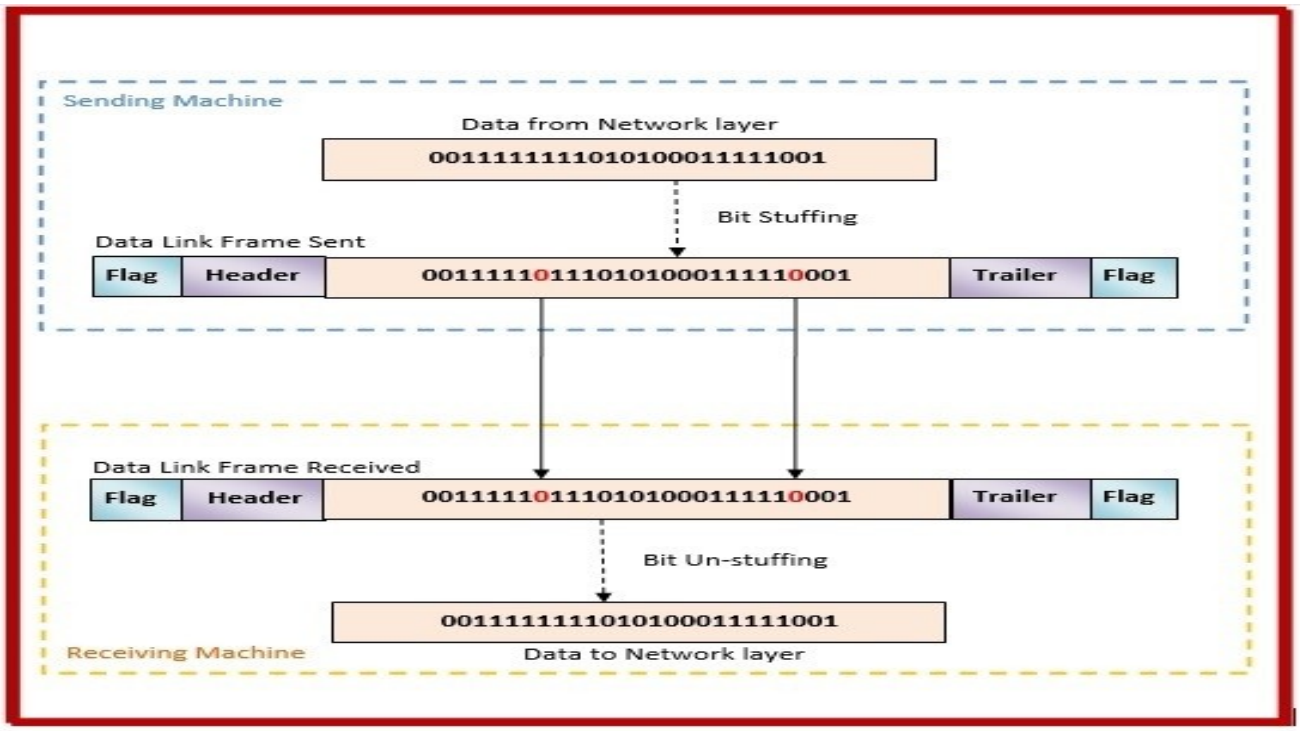
- **Frame Header** − It contains the source and the destination addresses of the frame.
- **Payload field** − It contains the message to be delivered.
- **Trailer** − It contains the error detection and error correction bits.
- **Flags** − A bit pattern that defines the beginning and end bits in a frame. It is generally of 8-bits. Most protocols use the 8-bit pattern 01111110 as flag.

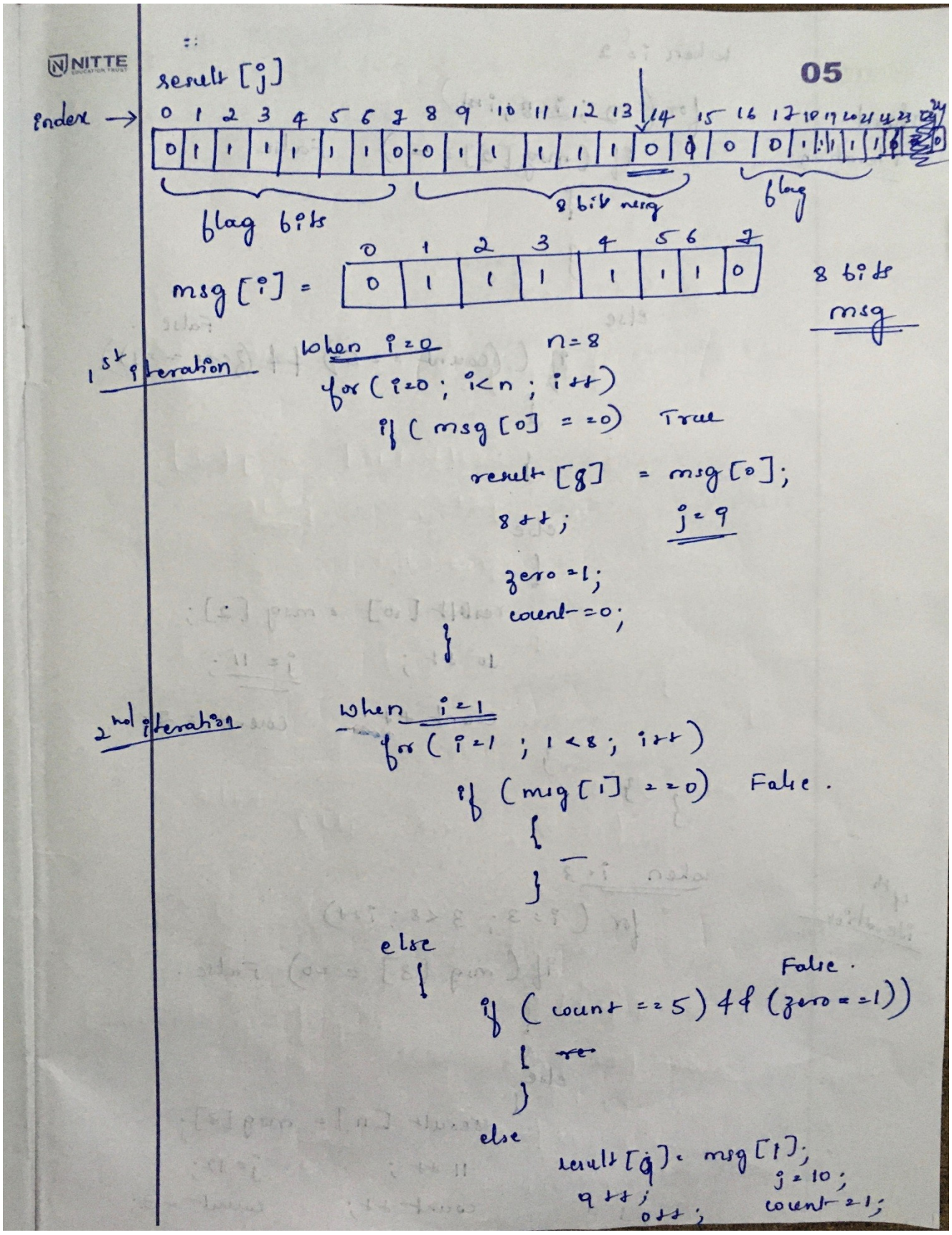


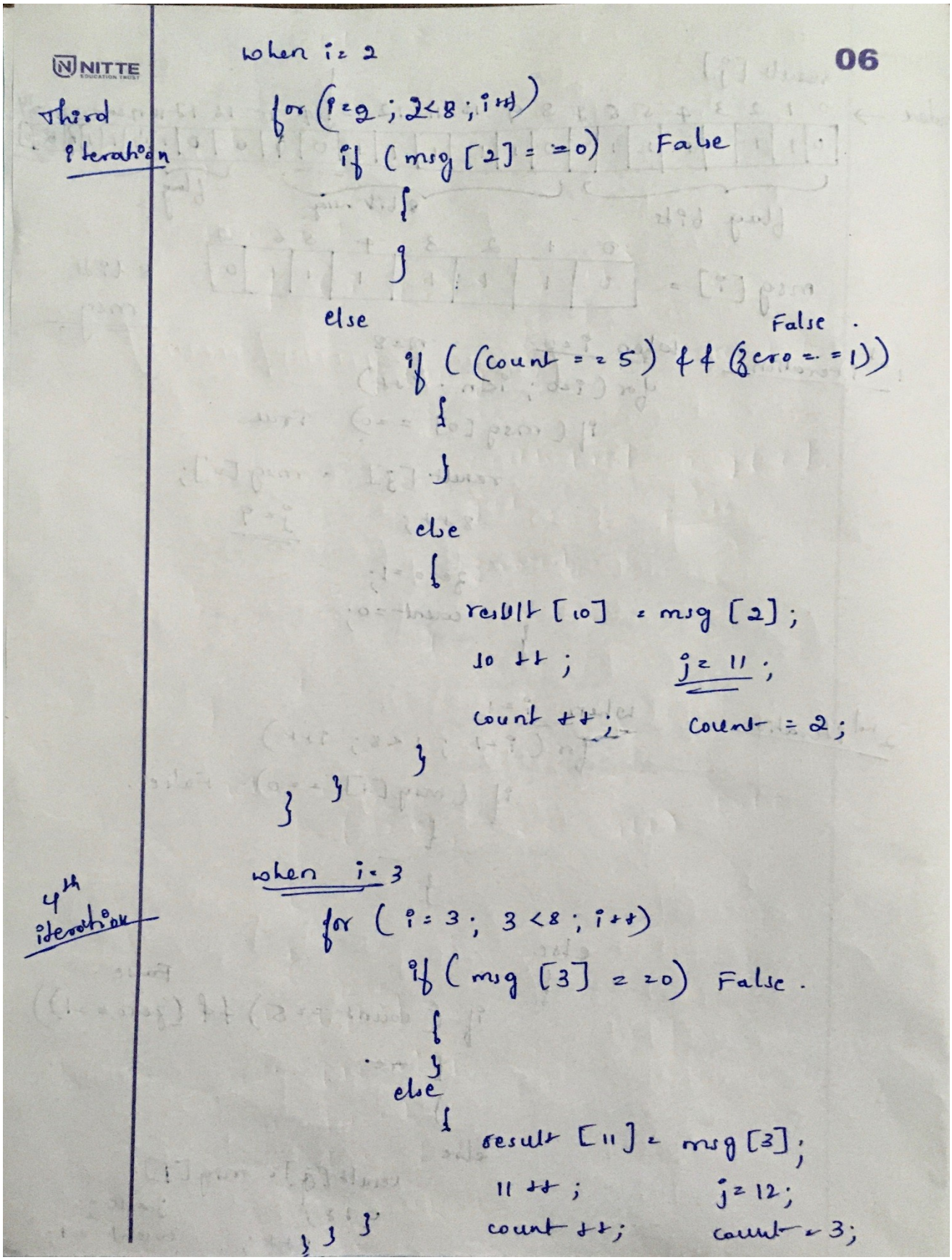
Bit Stuffing Mechanism

In a data link frame, the delimiting flag sequence generally contains six or more consecutive 1s. In order to differentiate the message from the flag in case of the same sequence, a single bit is stuffed in the message. Whenever a 0 bit is followed by five consecutive 1bits in the message, an extra 0 bit is stuffed at the end of the five 1s.

When the receiver receives the message, it removes the stuffed 0s after each sequence of five 1s. The un-stuffed message is then sent to the upper layers.




NITTE
05
result [j]
index →
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
flag bits
8 bit msg
flag
msg [i] =
0 1 2 3 4 5 6 7
8 bits
msg
1st iteration
when i=0
n=8
for (i=0; i<n; i++)
if (msg[0] == 0) True
result [8] = msg[0];
8++;
j=9
zero=1;
count=0;
2nd iteration
when i=1
for (i=1; 1<8; i++)
if (msg[1] == 0) False.
else
if (count == 5) && (zero == 1))
result[9] = msg[1];
9++;
j=10;
count=1;



NITTE EDUCATION TRUST

06

when i = 2

Third iteration

for (i=2 ; 2<8 ; i++)

if (msg[2] == 0) False

{

}

else False .

if ((count == 5) && (zero == 1))

{

}

else

{

result[10] = msg[2];

10 ++ ; j = 11 ;

count ++ ; count = 2 ;

}

}

}

when i = 3

4th iteration

for (i = 3 ; 3 < 8 ; i++)

if (msg[3] == 0) False .

{

}

else

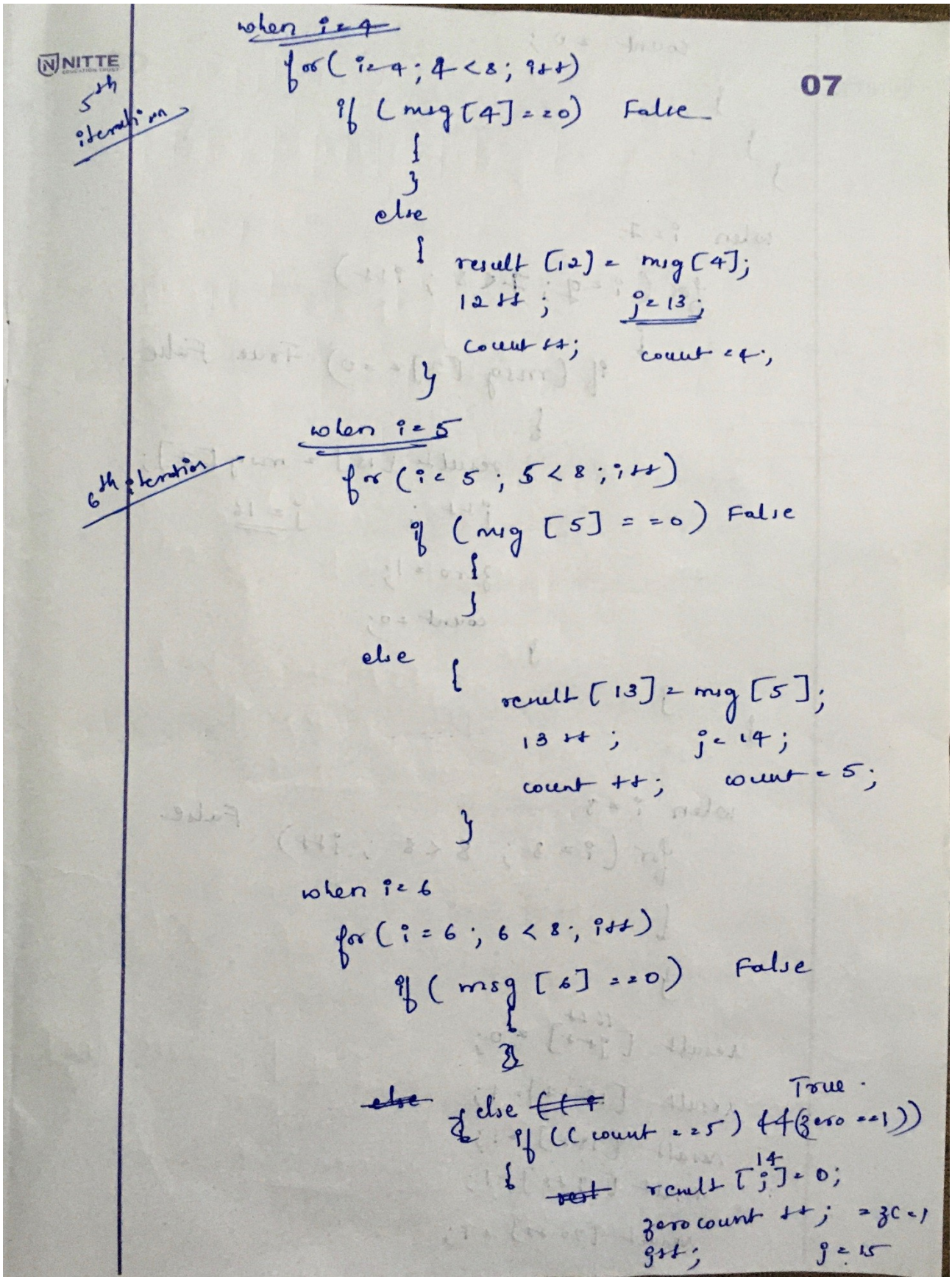
{

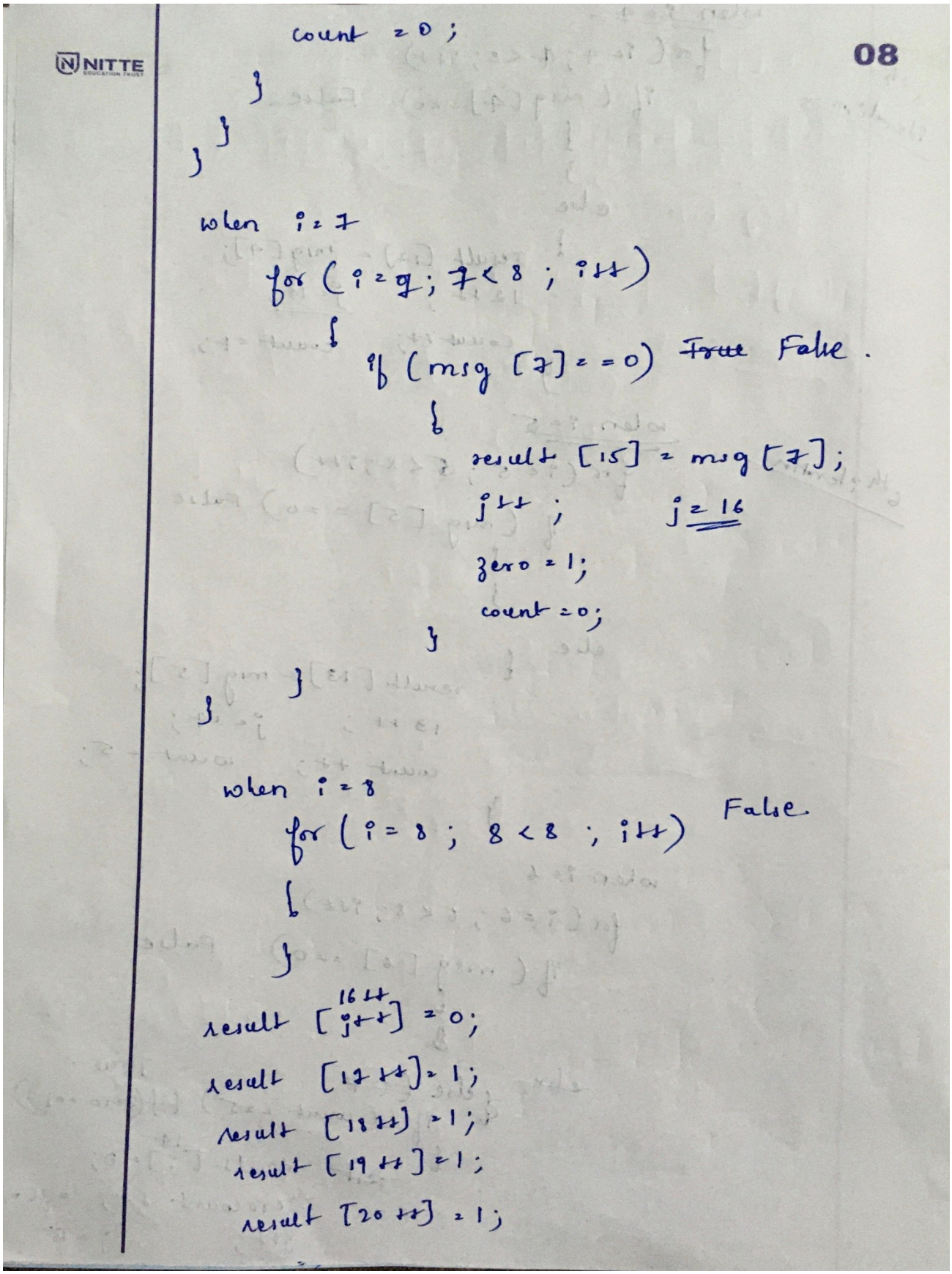
result[11] = msg[3];

11 ++ ; j = 12 ;

count ++ ; count = 3 ;

} } }


NITTE
07
5th iteration
when i=4
for(i=4; 4<8; i++)
if (msg[4]==0) False
{
}
else
{
result[12] = msg[4];
12++; j=13;
count++; count=4;
}
6th iteration
when i=5
for(i=5; 5<8; i++)
if (msg[5]==0) False
{
}
else
{
result[13] = msg[5];
13++; j=14;
count++; count=5;
}
when i=6
for(i=6; 6<8; i++)
if (msg[6]==0) False
{
}
else
True.
if ((count==5) && (zero==1))
{
result[14]=0;
zerocount++; zc=1
j++; j=15



NITTE EDUCATION TRUST

08

```
        count = 0;
      }
    }
  }
```

when i = 7

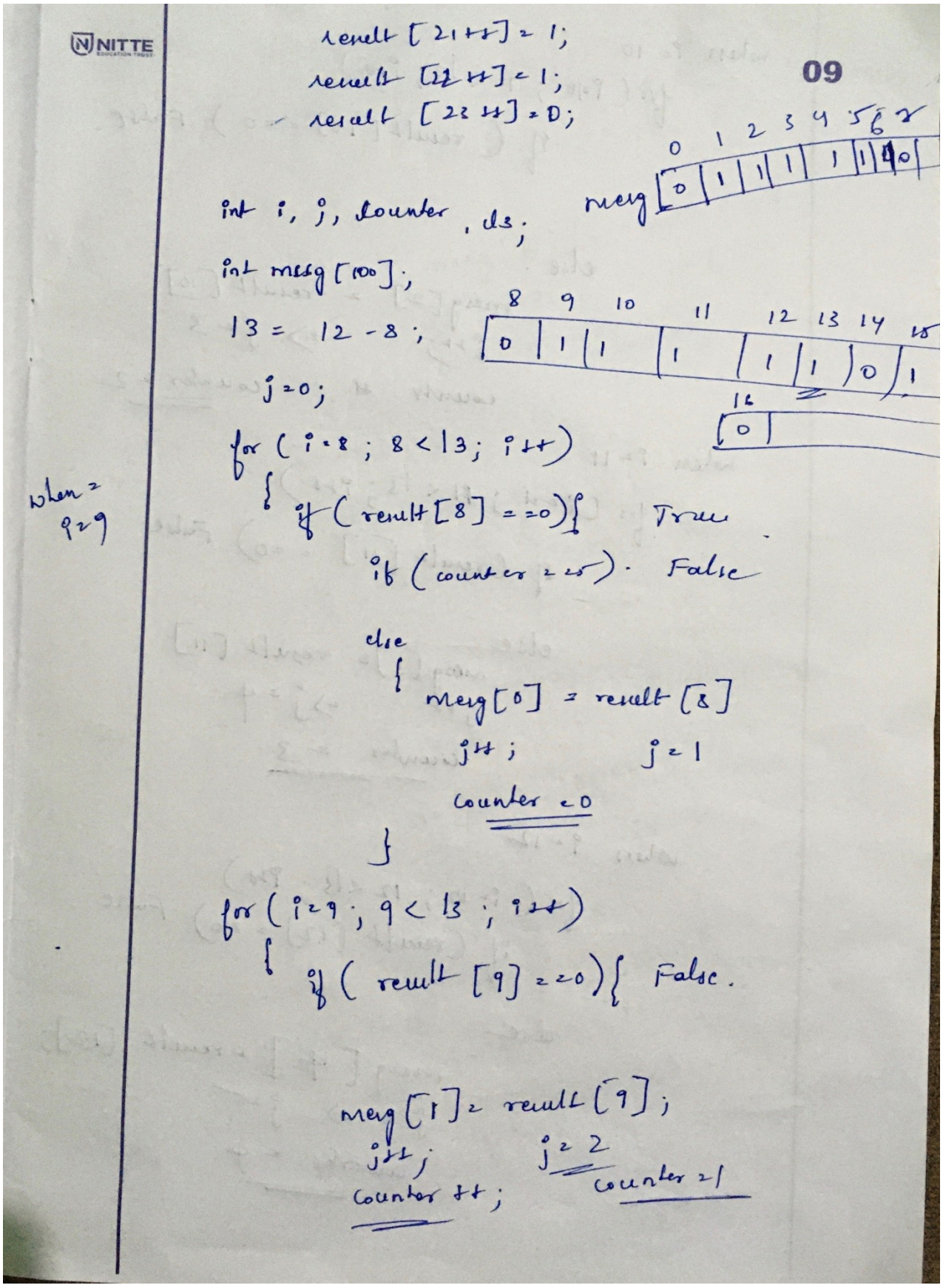
```
for (i=7; 7<8 ; i++)
    {
      if (msg[7]==0)   True  False.
        {
          result[15] = msg[7];
          j++;          j = 16
          zero = 1;
          count = 0;
        }
    }
}
```

when i = 8

```
for (i = 8 ; 8 < 8 ; i++)
{
}
```

False.

```
result [16++j++] = 0;
result [17++] = 1;
result [18++] = 1;
result [19++] = 1;
result [20++] = 1;
```



result [21++] = 1;
result [22++] = 1;
result [23++] = 0;

09

int i, j, counter, d3;
int mesg [100];
13 = 12 - 8;
j = 0;

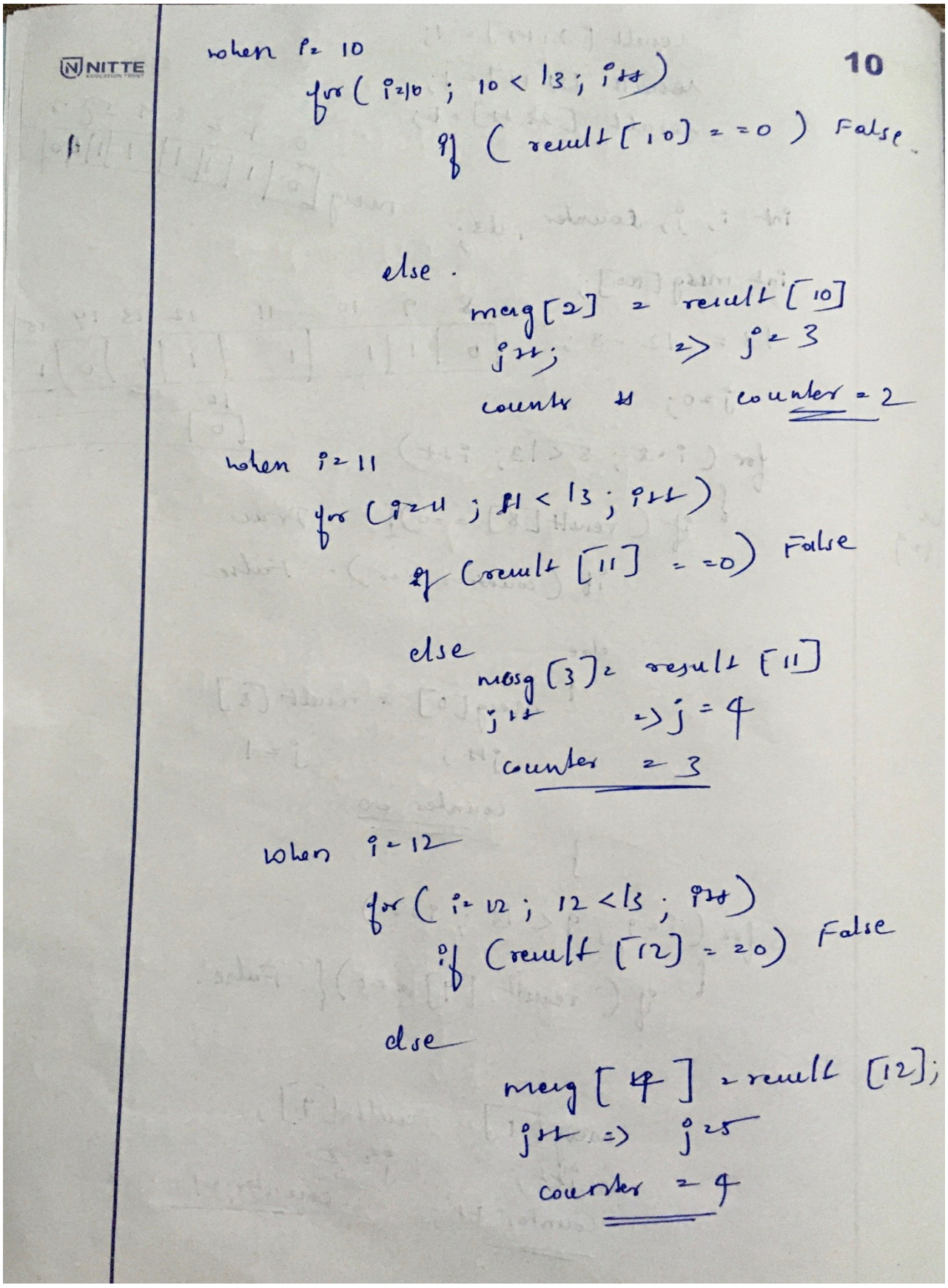
for (i = 8 ; 8 < 13; i++)
{
if (result [8] == 0){ True
if (counter == 25). False

else
{
mesg [0] = result [8]
j++; j = 1
counter = 0
}

for (i = 9; 9 < 13 ; 9++)
{
if (result [9] == 0){ False.

mesg [1] = result [9];
j++; j = 2
counter ++; counter = 1

when = 9 = 9



when i = 10

for (i=10 ; 10 < 13 ; i++)

if (result[10] == 0) False

else.

msg[2] = result[10]

j++; => j = 3

counter ++ counter = 2

when i = 11

for (i=11 ; 11 < 13 ; i++)

if (result[11] == 0) False

else

msg[3] = result[11]

j++ => j = 4

counter = 3

when i = 12

for (i = 12 ; 12 < 13 ; i++)

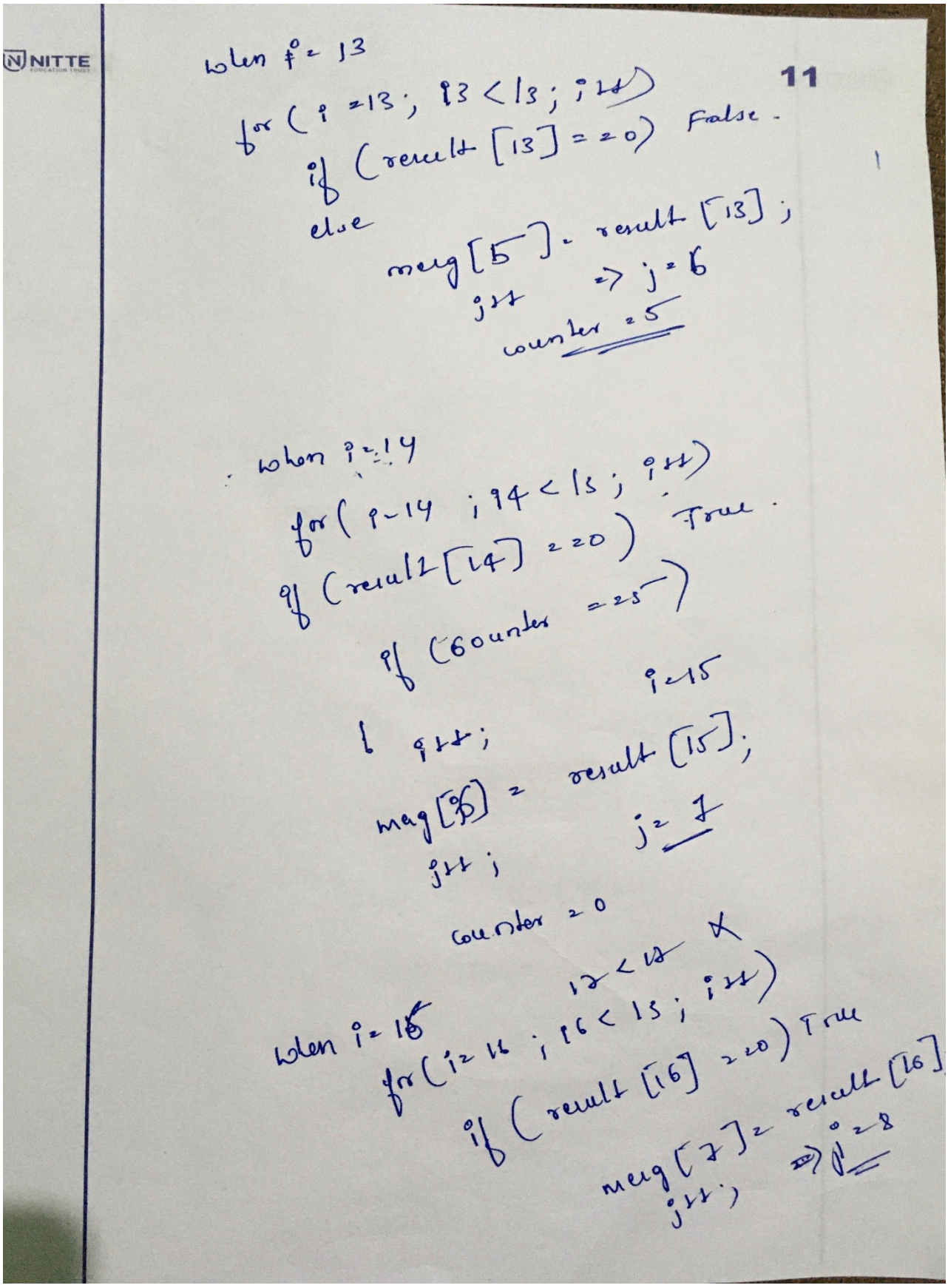
if (result[12] == 0) False

else

msg[4] = result[12];

j++ => j = 5

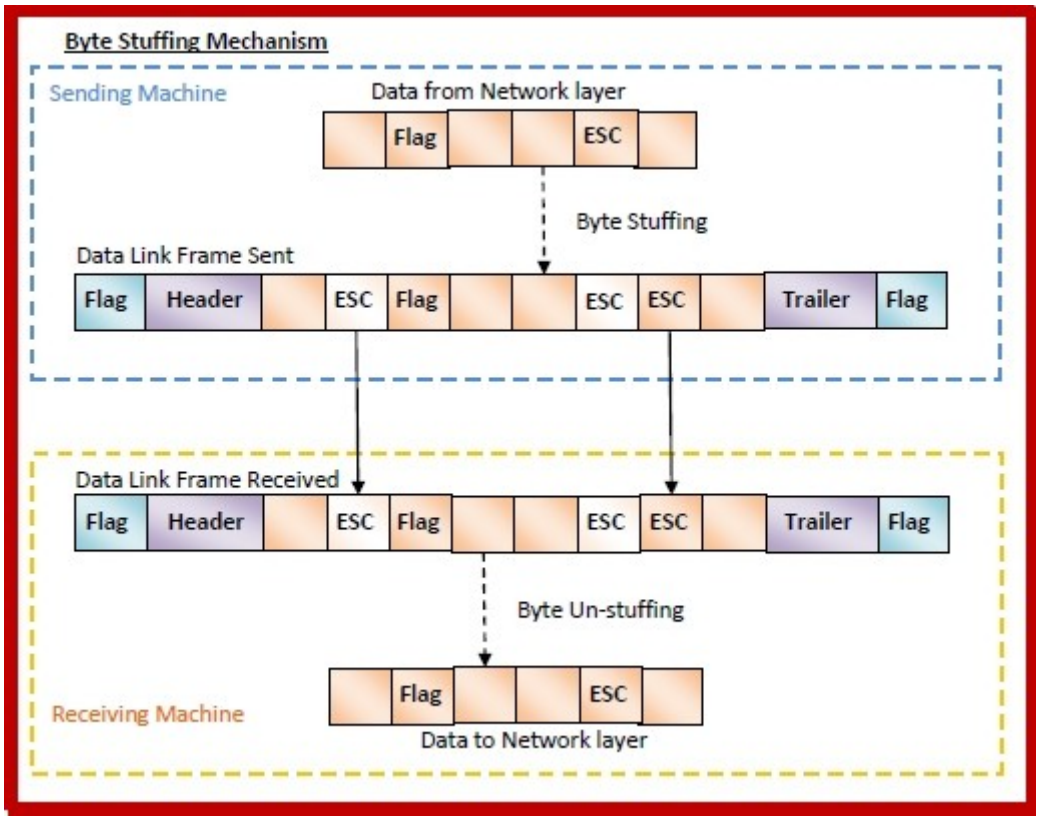
counter = 4


NITTE
11
when i = 13
for (i = 13; i3 < 13; i++)
if (result [13] = = 0) False.
else
msg [5] = result [13];
j++ => j = 6
counter = 5
when i = 14
for (i = 14 ; i4 < 15 ; i++)
if (result [14] = = 0) True.
if (counter = = 5)
i = 15
{ i++;
msg [6] = result [15];
j++; j = 7
counter = 0
when i = 16
for (i = 16 ; i6 < 15 ; i++)
if (result [16] = = 0) True
msg [7] = result [16]
j++; => j = 8

Byte stuffing (or character stuffing):

- In this sender's data link layer insert a special escape byte (ESC) just before each "accidental" flag byte in the data.
- The data link layer on the receiving end removes the escape byte before the data are given to the network layer.
- This technique is called **byte stuffing or character stuffing**.

If the pattern of the flag byte is present in the message byte sequence, there should be a strategy so that the receiver does not consider the pattern as the end of the frame. Here, a special byte called the escape character (ESC) is stuffed before every byte in the message with the same pattern as the flag byte. If the ESC sequence is found in the message byte, then another ESC byte is stuffed before it.



Problem-2

Design and Implement error detection methods used in Data link layer.

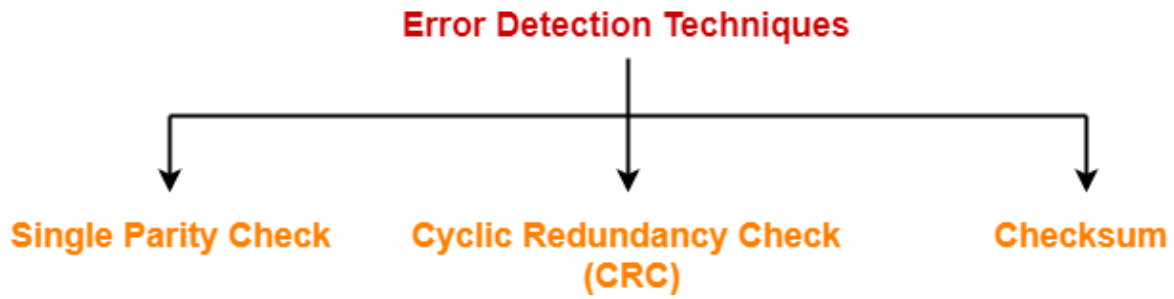
Data communication requires at least two devices working together, one to send and the other to receive. Even such a basic arrangement requires a great deal of coordination for an intelligible exchange to occur. The most important responsibilities of the data link layer are flow control and **error control**

Error Control

Error control is both error detection and error correction. It allows the receiver to inform the sender of any frames lost or damaged in transmission and coordinates the retransmission of those frames by the sender. In the data link layer, the term error control refers primarily to methods of error detection and retransmission. Error control in the data link layer is often implemented simply: Any time an error is detected in an exchange, specified frames are retransmitted. This process is called **automatic repeat request (ARQ)**.

Error detection is a technique that is used to check if any error occurred in the data during the transmission.

Some popular error detection methods are-



- Single Parity Check
- **Cyclic Redundancy Check (CRC)**
- Checksum

Cyclic Redundancy Check (CRC)

- Cyclic Redundancy Check (CRC) is an error detection method.
- It is based on binary division.

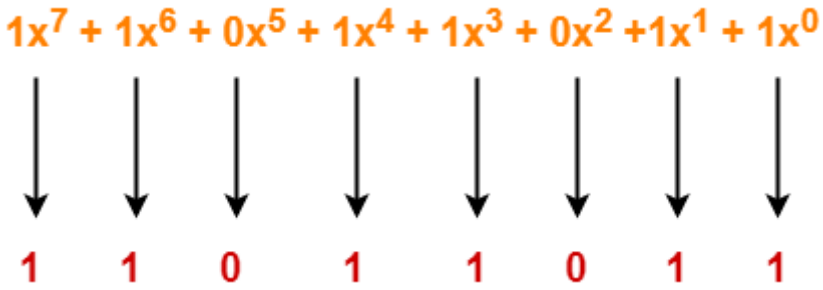
CRC Generator-

- CRC generator is an algebraic polynomial represented as a bit pattern.
- Bit pattern is obtained from the CRC generator using the following rule-
    - The power of each term gives the position of the bit and the coefficient gives the value of the bit.

Example-

Consider the CRC generator is x7 + x6 + x4 + x3 + x + 1.

The corresponding binary pattern is obtained as-



Thus, for the given CRC generator, the corresponding binary pattern is **11011011**.

Properties Of CRC Generator-

The algebraic polynomial chosen as a CRC generator should have at least the following properties-

- It should not be divisible by x.
- It should be divisible by x+1.

Steps Involved-

Error detection using CRC technique involves the following steps-

Step-01: Calculation Of CRC At Sender Side-

At sender side,

- A string of n 0's is appended to the data unit to be transmitted.
- Here, n is one less than the number of bits in CRC generator.
- Binary division is performed of the resultant string with the CRC generator.
- After division, the remainder so obtained is called as **CRC**.
- It may be noted that CRC also consists of n bits.

Step-02: Appending CRC To Data Unit-

At sender side,

- The CRC is obtained after the binary division.
- The string of n 0's appended to the data unit earlier is replaced by the CRC remainder.

Step-03: Transmission To Receiver-

- The newly formed code word (Original data + CRC) is transmitted to the receiver.

Step-04: Checking at Receiver Side-

At receiver side,

- The transmitted code word is received.
- The received code word is divided with the same CRC generator.
- On division, the remainder so obtained is checked.

The following two cases are possible-

Case-01: Remainder = 0

If the remainder is zero,

- Receiver assumes that no error occurred in the data during the transmission.
- Receiver accepts the data.

Case-02: Remainder ≠ 0

If the remainder is non-zero,

- Receiver assumes that some error occurred in the data during the transmission.
- Receiver rejects the data and asks the sender for retransmission.

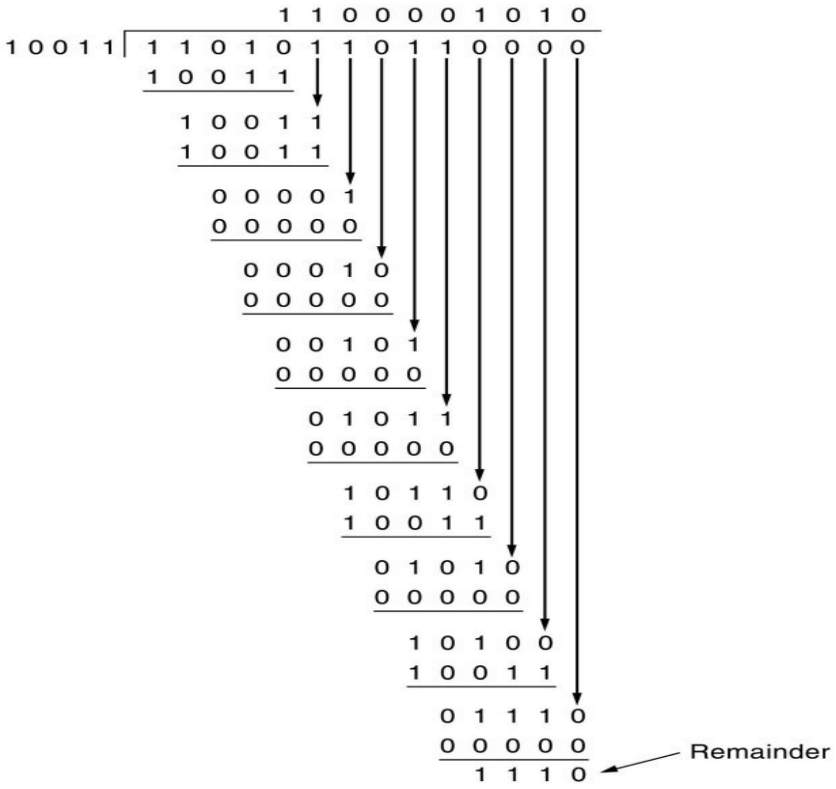
Problem:

A bit stream **1101011011** is transmitted using the standard CRC method. The generator polynomial is **x4+x+1**. What is the actual bit string transmitted?

Solution-

- The generator polynomial G(x) = x4 + x + 1 is encoded as **10011.**
- Clearly, the generator polynomial consists of 5 bits.
- So, a string of 4 zeroes is appended to the bit stream to be transmitted.
  11010110110000
- The resulting bit stream is 1101011011**0000**.

Now, the binary division is performed as-



From here, CRC = 1110.

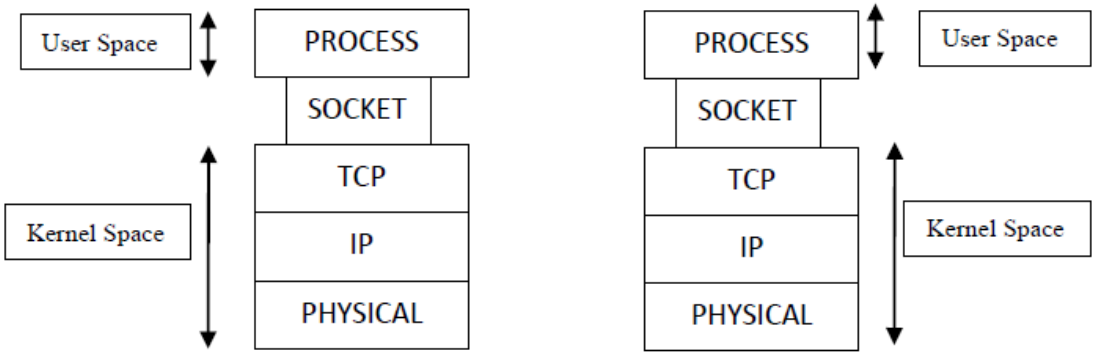
Now,

- The code word to be transmitted is obtained by replacing the last 4 zeroes of 11010110110**0000** with the CRC.
- Thus, the code word transmitted to the receiver = 11010110111**1110**.

Problem-3 (TCP) & 4 (UDP)

Write a C program to implement client-server model using socket programming.

- In layman's term, a Socket is an end point of communication between two systems on a network.
- To be a bit precise, a socket is a combination of IP address and port on one system.
- On each system a socket exists for a process interacting with the socket on other system over the network.
- Socket helps in identifying a process on a host machine through it's protocol, IP address and port number

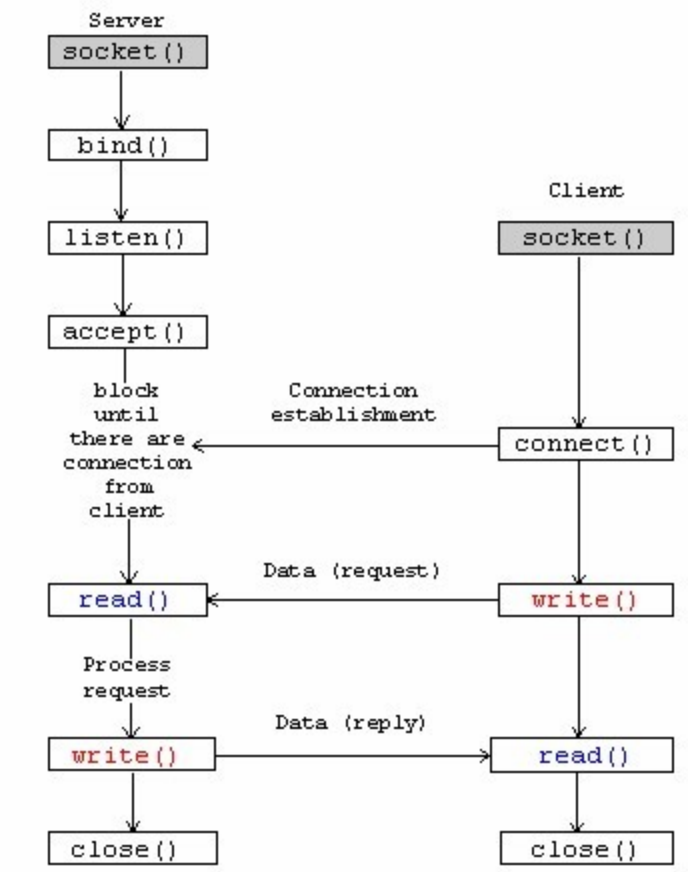


- Primary Socket Calls
  - socket() → create a new socket and return its descriptor
  - bind() → Associate a socket with a port and address
  - listen() → Establish queue for connection requests
  - accept() → Accept a connection request
  - connect() → Initiate a connection to a remote host
  - recv() → Receive data from a socket descriptor
  - send() → Send data to a socket descriptor
  - close() → "one-way" close of a socket descriptor
- Functions which help us in converting from host specific details to network specific details
  - Gethostname→ Given a hostname, returns a structure which specifies its DNS name(s) and IP address

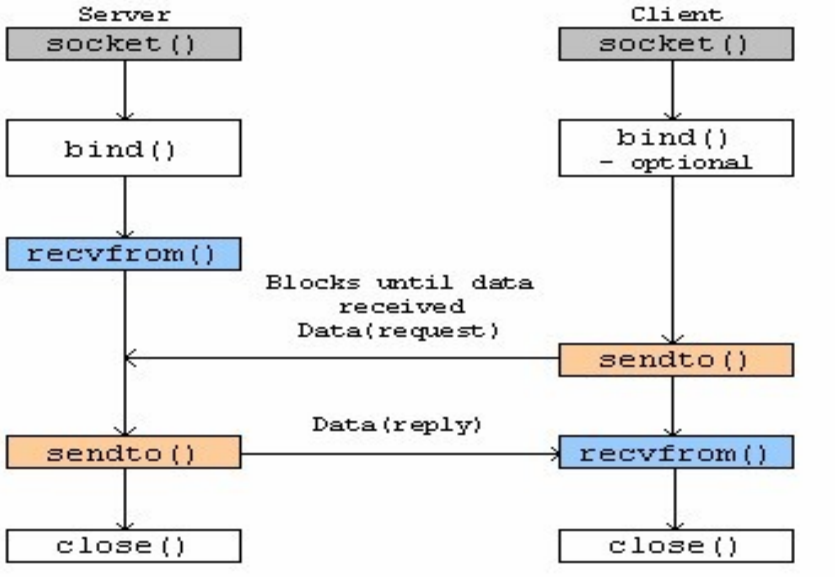
- Getservbyname→ Given service name and protocol, returns a structure which specifies its name(s) and its port address

- Socket Utility Functions
  - ntohs/ntohl→ To convert from network byte order to host byte order
  - htons/htonl→ To convert from host byte order to network byte order
  - inet_itoa/inet_addr→ Convert 32 bit IP address from network byte order to dotted decimal order
- Header Files
  - <sys/types.h>
  - <errno.h>
  - <sys/socket.h>→structsockaddr; system prototypes and constants; system prototypes and constants
  - <netdb.h>→ network info lookup prototypes and structure
  - <netinet/in.h>→structsockaddr_in; byte ordering macros_in; byte ordering macros
  - <arpa/inet.h>
- Two types of (TCP/IP) sockets
  - Stream sockets (e.g. uses TCP) → provide reliable byte-stream service
  - Datagram sockets (e.g. uses UDP) → provide best-effort datagram service and messages up to 65.500 byte
- Client-Server communication
  - Server
    - passively waits for and responds to clients
  - Client
    - initiates the communication
    - must know the address and the port of the server
    - active socket

TCP Based client-server communication



UDP based client server communication



Socket creation in C

intsockid= socket(family, type, protocol);

- sockid→ socket descriptor, an integer (like a file-handle)
- family → integer, communication domain, e.g.,
  - AF_INET, IPv4 protocols, Internet addresses (typically used)
  - AF_UNIX, Local communication, File addresses
- Type →Communication type
  - SOCK_STREAM – reliable TCP, 2-way, connection-based service
  - SOCK_DGRAM – unreliable UDP, connectionless, messages of maximum length
- Protocol → specifies protocol IPPROTO_TCP IPPROTO_UDP
  - usually set to 0 (i.e., use default protocol)
- upon failure returns -1

Bind Function

- associates and reserves a port for use by the socket

int status = bind(sockid, &addrport, size);

- sockid: integer, socket descriptor
- addrport: structsockaddr, the (IP) address and port of the machine for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
- size: the size (in bytes) of the addrport structure
- status: upon failure -1 is returned

Listen Function

- Instructs TCP protocol implementation to listen for connections
  - int status = listen(sockid, queueLimit);
- sockid: integer, socket descriptor
- queuelen: integer, number of active participants that can "wait" for a connection
- status: 0 if listening, -1 if error

Connect Function

- The client establishes a connection with the server by calling connect()

```
int status = connect(sockid, &foreignAddr, addrlen)
```

- sockid: integer, socket to be used in connection
- foreignAddr : structsockaddr: address of the passive participant
- addrlen: integer, sizeof(name)
- status: 0 if successful connect, -1 otherwise

Accept Function

- The server gets a socket for an incoming client connection by calling accept()

```
int s= accept(sockid, &clientAddr, &addrLen);
```

- s: integer, the new socket (used for data-transfer)
- sockid: integer, the orig. socket (being listened on)
- clientAddr: structsockaddr, address of the active participant filled in upon return
- addrLen: sizeof(clientAddr): value/result parameter must be set appropriately before call adjusted upon return

Exchanging data with stream socket

```
int count = send(sockid, msg, msgLen, flags);
```

- msg: const void[], message to be transmitted
- msgLen: integer, length of message (in bytes) to transmit
- flags: integer, special options, usually just 0
- count: Number bytes transmitted (-1 if error)

```
int count = recv(sockid, recvBuf, bufLen, flags);
```

- recvBuf: void[], stores received bytes
- bufLen: Number bytes received
- flags: integer, special options, usually just 0
- count: Number bytes received (-1 if error)

Exchanging Data with Datagram Sockets

```
int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);
```

- msg, msgLen, flags, count: same with send()
- foreignAddr : structsockaddr, address of the destination
- addrLen: sizeof(foreignAddr)

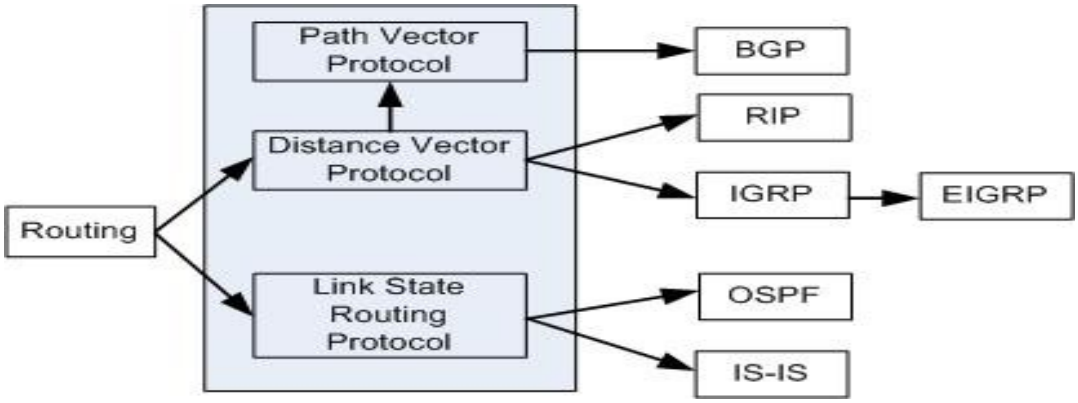
```
int count = recvfrom(sockid, recvBuf, bufLen, flags,&clientAddr, addrlen);
```

- recvBuf, bufLen, flags, count: same with recv()
- clientAddr: structsockaddr, address of the client
- addrLen: sizeof(clientAddr)

Problem-5

Design and Implement a C solution for Routing Packets.

- Routing/Forwarding is the process to place the packet in its route to its destination. Forwarding requires a host or a router to have a routing table.
- When a host has a packet to send or when a router has received a packet to be forwarded, it looks at this table to find the route to the final destination.
- Forwarding Techniques
    - Next-Hop Method Versus Route Method
    - Network-Specific Method Versus Host-Specific Method
- UNICAST ROUTING PROTOCOLS
    - Intra-Domain
    - Inter-Domain
- Intra-Domain
    - Distance Vector Routing
    - Link-State Routing
    - Path Vector Routing

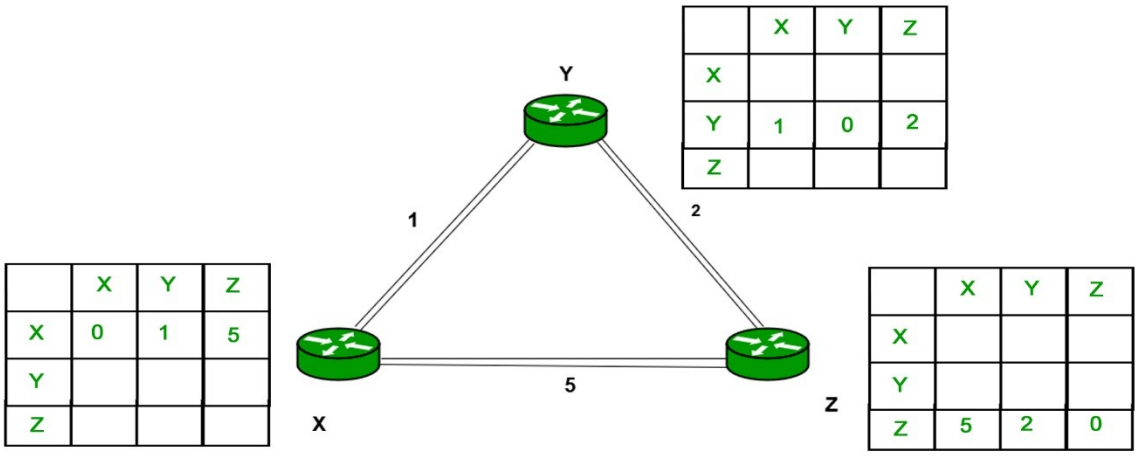


Distance Vector Routing

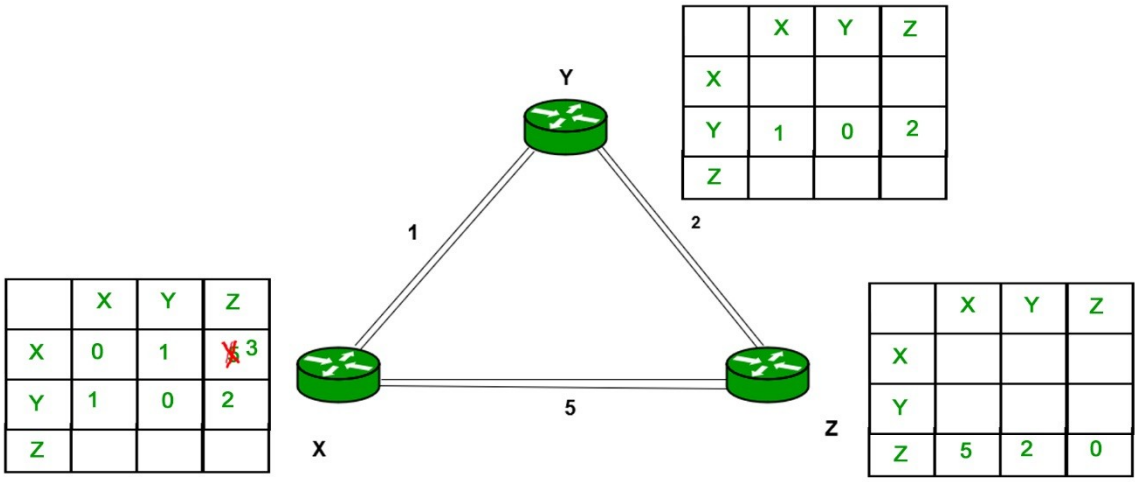
- **distance-vector routing** protocol in data networks determines the best route for data packets based on **distance**. **Distance-vector routing** protocols measure the **distance** by the number of routers a packet has to pass, one **router** counts as one hop.
- In distance vector routing, the least-cost route between any two nodes is the route with minimum distance.
- As the name implies, each node maintains a vector (table) of minimum distances to every node.
- The table at each node also guides the packets to the desired node by showing the next stop in the route
- Routing Table Mainly Consists of
  - TO
  - Cost
  - Next Hop
- Initialization
  - Each node can know only the distance between itself and its immediate neighbours, those directly connected to it.
  - Each node can send a message to the immediate neighbours and find the distance between itself and these neighbours.
- Sharing
  - The whole idea of distance vector routing is the sharing of information between neighbours.
  - Sharing of First two fields (TO and COST)
- Updating: When a node receives a two-column table from a neighbour, it needs to update its routing table. Updating takes three steps:
  - The receiving node needs to add the cost between itself and the sending node to each value in the second column. The logic is clear. If node ‘C’ claims that its distance to a destination is x miles, and the distance between ‘A’ and ‘C’ is y miles, then the distance between ‘A’ and that destination, via ‘C’, is x + y miles.
  - The receiving node needs to add the name of the sending node to each row as the third column if the receiving node uses information from any row. The sending node is the next node in the route.

- The receiving node needs to compare each row of its old table with the corresponding row of the modified version of the received table.
    - If the next-node entry is different, the receiving node chooses the row with the smaller cost. If there is a tie, the old one is kept.
    - If the next-node entry is the same, the receiving node chooses the new row. For example, suppose node 'C' has previously advertised a route to node 'X' with distance 3. Suppose that now there is no path between 'C' and 'X'; node C now advertises this route with a distance of infinity. Node 'A' must not ignore this value even though its old entry is smaller. The old route does not exist anymore. The new route has a distance of infinity.

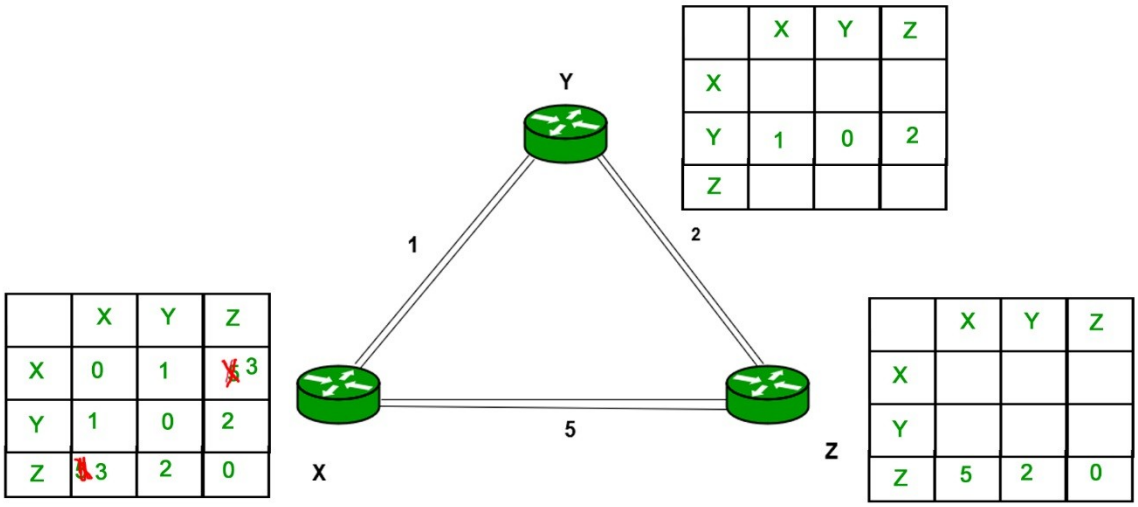
Example – Consider 3-routers X, Y and Z as shown in figure. Each router have their routing table. Every routing table will contain distance to the destination nodes.



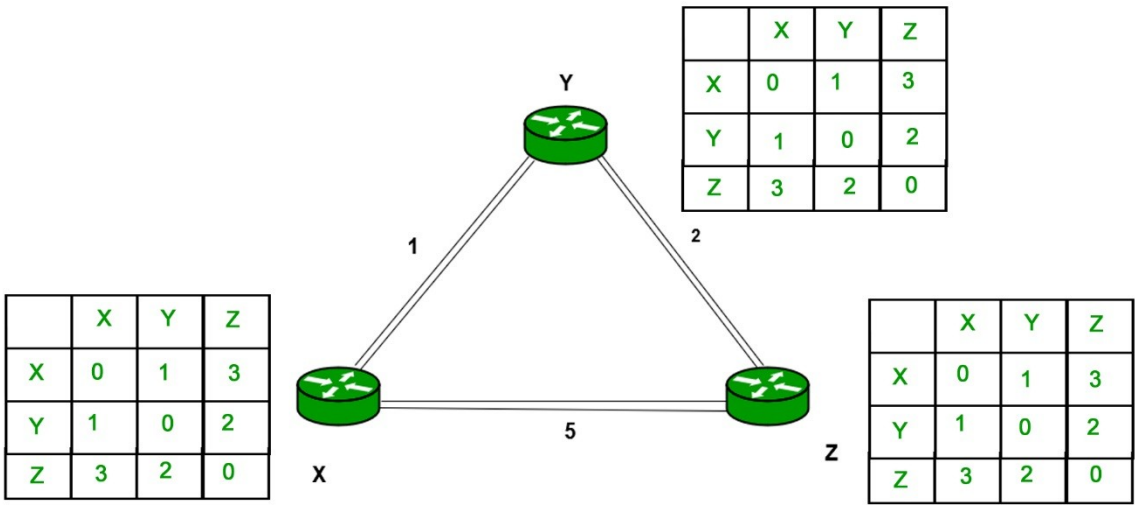
Consider router X , X will share it routing table to neighbors and neighbors will share it routing table to it to X and distance from node X to destination will be calculated



Similarly for Z also –



Finally the routing table for all –



Algorithm

```
struct table
{
        unsigned cost[20];
        unsigned next_hop[20];
}router[10];

Cost_matrix=cost_of_each_link;

/*INITIALIZATION*/
If(cost of link (i,j) is present in cost matrix)
{
        Router[i].cost[j]=cost(i,j)
        Router[i].next_hop[j]=j;
}
Else
{

        Router[i].cost[j]=infinity

}

/*UPDATION*/
for i from 1 to n-1:
      for each edge (u, v) with weight w in edges:
         if Router[u].cost[v] + w <Router[u].cost[v]:
Router[u].cost[v] := Router[u].cost[v] + w
```

Problem-6

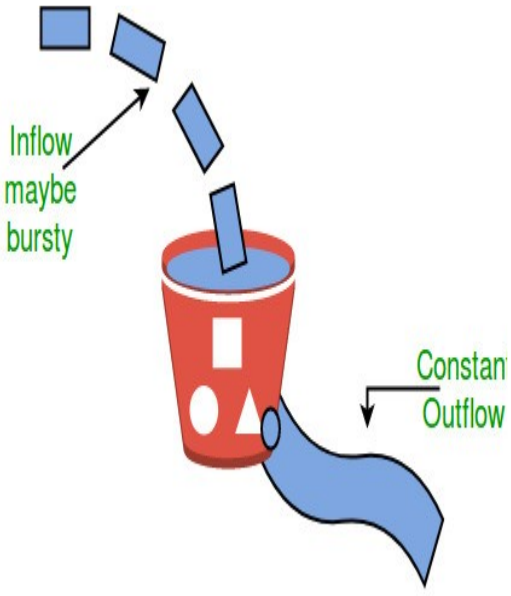
Design and Implement a C solution for congestion control in Transport Layer.

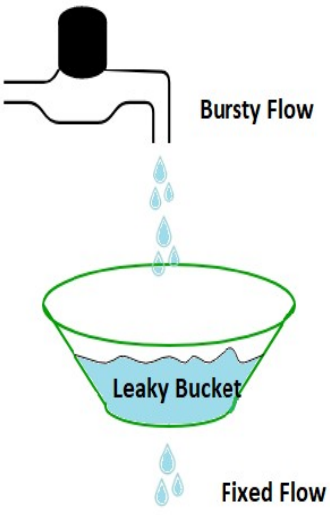
- A state occurring in network layer when the message traffic is so heavy that it slows down network response time.
- Congestion in a network may occur if the load on the network-the number of packets sent to the network-is greater than the capacity of the network-the number of packets a network can handle.
- Congestion control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity.
- TCP, unlike UDP, takes into account congestion in the network.
- **Effects of Congestion**
    - As delay increases, performance decreases.
    - If delay increases, retransmission occurs, making situation worse
- Congestion control algorithms
    - Leaky Bucket Algorithm
    - Token bucket Algorithm

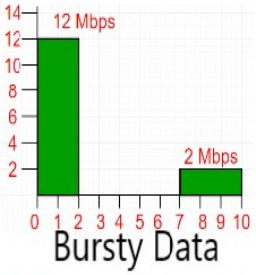
Leaky Bucket Algorithm

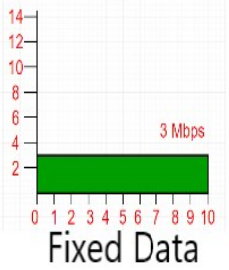
Let us consider an example to understand

- Imagine a bucket with a small hole in the bottom. No matter at what rate water enters the bucket, the outflow is at constant rate. When the bucket is full with water additional water entering spills over the sides and is lost.









- Similarly, each network interface contains a leaky bucket and the following steps are involved in leaky bucket algorithm:

1. When host wants to send packet, packet is thrown into the bucket.
2. The bucket leaks at a constant rate, meaning the network interface transmits packets at a constant rate.
3. Bursty traffic is converted to a uniform traffic by the leaky bucket.
4. In practice the bucket is a finite queue that outputs at a finite rate.

In the figure, we assume that the network has committed a bandwidth of 3 Mbps for a host. The use of the leaky bucket shapes the input traffic to make it conform to this commitment. In Figure the host sends a burst of data at a rate of 12 Mbps for 2 s, for a total of 24 Mbits of data. The host is silent for 5 s and then sends data at a rate of 2 Mbps for 3 s, for a total of 6 Mbits of data. In all, the host has sent 30 Mbits of data in 10 s. The leaky bucket smooths the traffic by sending out data at a rate of 3 Mbps during the same 10 s.
Without the leaky bucket, the beginning burst may have hurt the network by consuming more bandwidth than is set aside for this host. We can also see that the leaky bucket may prevent congestion.