Program- 4

Write a program for error detecting code using CRC-CCITT (16- bits).

Whenever digital data is stored or interfaced, data corruption might occur. Since the beginning of computer science, developers have been thinking of ways to deal with this type of problem. For serial data they came up with the solution to attach a parity bit to each sent byte. This simple detection mechanism works if an odd number of bits in a byte changes, but an even number of false bits in one byte will not be detected by the parity check. To overcome this problem developers have searched for mathematical sound mechanisms to detect multiple false bits. The CRC calculation or cyclic redundancy check was the result of this. Nowadays CRC calculations are used in all types of communications. All packets sent over a network connection are checked with a CRC. Also each data block on your hard disk has a CRC value attached to it. Modern computer world cannot do without these CRC calculations. So let's see why they are so widely used. The answer is simple; they are powerful, detect many types of errors and are extremely fast to calculate especially when dedicated hardware chips are used.

The idea behind CRC calculation is to look at the data as one large binary number. This number is divided by a certain value and the remainder of the calculation is called the CRC. Dividing in the CRC calculation at first looks to cost a lot of computing power, but it can be performed very quickly if we use a method similar to the one learned at school. We will as an example calculate the remainder for the character 'm'—which is 1101101 in binary notation—by dividing it by 19 or 10011. Please note that 19 is an odd number. This is necessary as we will see further on. Please refer to your schoolbooks as the binary calculation method here is not very different from the decimal method you learned when you were young. It might only look a little bit strange. Also notations differ between countries, but the method is similar.

```
                    1 0 1 = 5
            -------------
1 0 0 1 1 / 1 1 0 1 1 0 1
            1 0 0 1 1 | |
            --------- | |
              1 0 0 0 0 |
              0 0 0 0 0 |
              --------- |
              1 0 0 0 0 1
                1 0 0 1 1
                ---------
                  1 1 1 0 = 14 = remainder
```

With decimal calculations you can quickly check that 109 divided by 19 gives a quotient of 5 with 14 as the remainder. But what we also see in the scheme is that every bit extra to check only costs one binary comparison and in 50% of the cases one binary subtraction.

You can easily increase the number of bits of the test data string—for example to 56 bits if we use our example value "Lammert"—and the result can be calculated with 56 binary comparisons and an average of 28 binary subtractions. This can be implemented in hardware directly with only very few transistors involved. Also software algorithms can be very efficient.

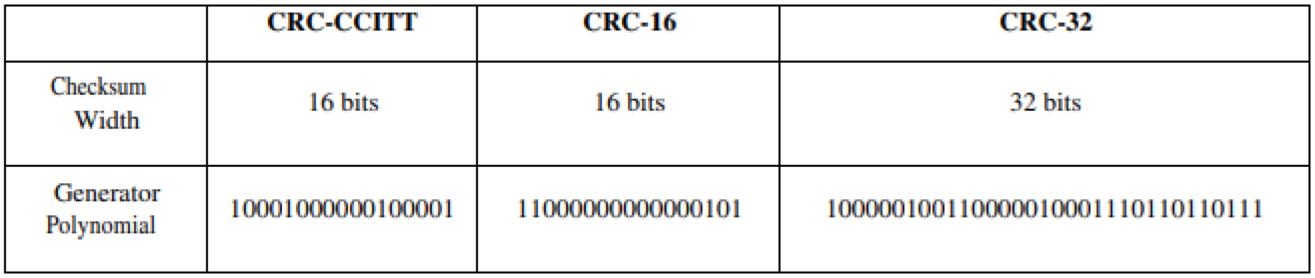
All of the CRC formulas you will encounter are simply checksum algorithms based on modulo-2 binary division where we ignore carry bits and in effect the subtraction will be equal to an exclusive or operation. Though some differences exist in the specifics across different CRC formulas, the basic mathematical process is always the same:

• The message bits are appended with c zero bits; this augmented message is the dividend

• A predetermined c+1-bit binary sequence, called the generator polynomial, is the divisor

• The checksum is the c-bit remainder that results from the division operation.

Table 1 lists some of the most commonly used generator polynomials for 16- and 32-bit CRCs. Remember that the width of the divisor is always one bit wider than the remainder. So, for example, you'd use a 17-bit generator polynomial whenever a 16-bit checksum is required.



	CRC-CCITT	**CRC-16**	**CRC-32**
Checksum Width	16 bits	16 bits	32 bits
Generator Polynomial	10001000000100001	11000000000000101	100000100110000010001110110110111

International Standard CRC Polynomials

Source code:

```
import java.util.*;
class crc
{
void div(int a[],int k)
{
int gp[]={1,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,1};
int count=0;
for(int i=0;i<k;i++)
{
if(a[i]==gp[0])
if(a[i]==gp[0])
{
for(int j=i;j<17+i;j++)
{
a[j]=a[j]^gp[count++];
}
count=0;
}
}
}
public static void main(String args[])
{
int a[]=new int[100];
int b[]=new int[100];
int len,k;
crc ob=new crc();
System.out.println("Enter the length of Data Frame:");
Scanner sc=new Scanner(System.in);
len=sc.nextInt();
int flag=0;
System.out.println("Enter the Message:");
for(int i=0;i<len;i++)
{ a[i]=sc.nextInt();
}
for(int i=0;i<16;i++)
{ a[len++]=0;
}
k=len-16;
for(int i=0;i<len;i++)
{ b[i]=a[i];
}
ob.div(a,k);
for(int i=0;i<len;i++)
a[i]=a[i]^b[i];
System.out.println("Data to be transmitted: ");
for(int i=0;i<len;i++)
{
```

```
System.out.print(a[i]+" ");
}
System.out.println();
System.out.println("Enter the Received Data: ");
for(int i=0;i<len;i++)
{
a[i]=sc.nextInt();
}
ob.div(a, k);
for(int i=0;i<len;i++)
{
if(a[i]!=0)
{
flag=1;
break;
}
}
if(flag==1)
System.out.println("ERROR in data");
else
System.out.println("NO ERROR");
}
}
```

OUTPUT-1

```
Enter the length of Data Frame: 4
Enter the Message: 1 0 1 1
Data to be transmitted: 1 0 1 1 1 0 1 1 0 0 0 1 0 1 1 0 1 0 1 1
Enter the Received Data: 1 0 1 1 1 0 1 1 0 0 0 0 0 1 1 0 1 0 1 1
ERROR in Data
```

OUTPUT-2

```
Enter the length of Data Frame: 4
Enter the Message: 1 0 1 1
Data to be transmitted: 1 0 1 1 1 0 1 1 0 0 0 1 0 1 1 0 1 0 1 1
Enter the Received Data: 1 0 1 1 1 0 1 1 0 0 0 1 0 1 1 0 1 0 1 1
NO ERROR
```

Program- 5

Write a program to implement sliding window protocol in datalink layer.

The sliding window protocol is a method used in network communications to manage the flow of data between two devices and ensure that data is transmitted efficiently and accurately. It's particularly useful in scenarios where data is sent over a network with potential delays or errors. Here’s a breakdown of how it works:

1. **Window Size**: Both sender and receiver maintain a "window" of frames or packets that can be sent or received. This window size determines the maximum number of frames that can be sent without receiving an acknowledgment.
2. **Sender's Window**: The sender can transmit a number of frames up to the window size before needing to wait for an acknowledgment from the receiver. Once the sender receives an acknowledgment for some of the frames, it can slide the window forward and send new frames.
3. **Receiver's Window**: The receiver also has a window size that specifies the number of frames it can buffer. It keeps track of which frames it has received correctly and sends acknowledgments back to the sender. If the receiver’s buffer is full, it will need to wait before it can accept more frames.
4. **Sliding the Window**: As acknowledgments are received, both the sender and receiver "slide" their windows forward. This sliding process means that as old frames are acknowledged and removed from the window, new frames can be sent or received.
5. **Error Handling**: The protocol includes mechanisms for handling errors. If a frame is lost or corrupted, the receiver may request retransmission of that frame, and the sender will retransmit it, usually sliding the window back to the point where the error occurred.
6. **Flow Control**: The sliding window protocol helps with flow control by preventing the sender from overwhelming the receiver with too many frames at once. The receiver's window size ensures that it only processes a manageable number of frames at any time.

This protocol is commonly used in various data link layer and transport layer protocols, such as the Transmission Control Protocol (TCP) in the Internet Protocol Suite.

Source code:

```
import java.util.LinkedList;
import java.util.Queue;
public class SlidingWindowProtocol {

    private static final int WINDOW_SIZE = 4;  // Size of the sliding window
    private static final int TOTAL_PACKETS = 10; // Total number of packets to send

    // Simulates the sender's sliding window protocol
    public static void sender() {
        Queue<Integer> window = new LinkedList<>();
        for (int i = 0; i < WINDOW_SIZE; i++) {
            if (i < TOTAL_PACKETS) {
                window.add(i);
                System.out.println("Sent packet: " + i);
            }
        }

        int nextPacketToSend = WINDOW_SIZE;

        while (nextPacketToSend < TOTAL_PACKETS || !window.isEmpty()) {
            if (!window.isEmpty()) {
                // Simulate receiving acknowledgment for the packet at the start of the window
                int ack = window.poll();
                System.out.println("Acknowledgment received for packet: " + ack);
            }

            // Slide the window
            if (nextPacketToSend < TOTAL_PACKETS) {
                window.add(nextPacketToSend);
System.out.println("Sent packet: " + nextPacketToSend);
                nextPacketToSend++;
            }

            // Simulate a delay for demonstration purposes
            try {
                Thread.sleep(1000); // 1 second delay
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("All packets sent and acknowledged.");
    }

    public static void main(String[] args) {
        sender();
    }

}
```

Explanation

1. **Initialization**:
   - The window size (WINDOW_SIZE) determines how many packets can be sent before receiving acknowledgments.
   - TOTAL_PACKETS represents the total number of packets to send.
2. **Sender Function**:
   - Initializes the window with the first set of packets.
   - Simulates sending packets and receiving acknowledgments.
   - Slides the window by removing the acknowledged packet and adding a new packet.
3. **Sliding the Window**:
   - After sending packets, the window slides by removing the packet at the start of the window (simulating acknowledgment) and adding new packets as they are sent.
4. **Simulation**:
   - The Thread.sleep(1000) call simulates a delay, representing the time between sending packets and receiving acknowledgments.

OUTPUT

```
Sent packet: 0
Sent packet: 1
Sent packet: 2
Sent packet: 3
Acknowledgment received for packet: 0
Sent packet: 4
Acknowledgment received for packet: 1
Sent packet: 5
Acknowledgment received for packet: 2
Sent packet: 6
Acknowledgment received for packet: 3
Sent packet: 7
Acknowledgment received for packet: 4
Sent packet: 8
Acknowledgment received for packet: 5
Sent packet: 9
Acknowledgment received for packet: 6
Acknowledgment received for packet: 7
Acknowledgment received for packet: 8
Acknowledgment received for packet: 9
All packets sent and acknowledged.
```

Program- 6

Write a program to find the shortest path between vertices using Bellman-ford algorithm.

Distance Vector Algorithm is a decentralized routing algorithm that requires that each router simply inform its neighbors of its routing table. For each network path, the receiving routers pick the neighbor advertising the lowest cost, then add this entry into its routing table for re-advertisement. To find the shortest path, Distance Vector Algorithm is based on one of two basic algorithms: the Bellman-Ford and the Dijkstra algorithms.

Routers that use this algorithm have to maintain the distance tables (which is a one-dimension array -- "a vector"), which tell the distances and shortest path to sending packets to each node in the network. The information in the distance table is always up date by exchanging information with the neighboring nodes. The number of data in the table equals to that of all nodes in networks (excluded itself). The columns of table represent the directly attached neighbors whereas the rows represent all destinations in the network. Each data contains the path for sending packets to each destination in the network and distance/or time to transmit on that path (we call this as "cost"). The measurements in this algorithm are the number of hops, latency, the number of outgoing packets, etc.

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.

Source code:

```
import java.util.Scanner;

public class BellmanFord

{

private int D[];

private int num_ver,n;

public static final int MAX_VALUE = 999;

public BellmanFord(int n)

{

this.n=n;

D = new int[n+1];

}
```

```
public void shortest(int s,int A[][])
{
for (int i=1;i<=n;i++)
{
D[i]=MAX_VALUE;
}
D[s] = 0;
for(int k=1;k<=n-1;k++)
{
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
{
if(A[i][j]!=MAX_VALUE)
{
if(D[j]>D[i]+A[i][j])
D[j]=D[i]+A[i][j];
}
}
}
}
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
{
if(A[i][j]!=MAX_VALUE)
{
if(D[j]>D[i]+A[i][j])
{
System.out.println("The Graph contains negative egde cycle");
return;
}
}
}
}
for(int i=1;i<=n;i++)
{
System.out.println("Distance of source " + s + " to "+ i + " is " + D[i]);
}
}
public static void main(String[ ] args)
{
int n=0,s;
Scanner sc = new Scanner(System.in);
System.out.println("Enter the number of vertices");
n = sc.nextInt();
int A[][] = new int[n+1][n+1];
System.out.println("Enter the Weighted matrix");
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
{
A[i][j]=sc.nextInt();
if(i==j)
{
A[i][j]=0;
continue;
}
if(A[i][j]==0)
{
```

```
A[i][j]=MAX_VALUE;
}
}}
System.out.println("Enter the source vertex");
s=sc.nextInt();
BellmanFord b = new BellmanFord(n);
b.shortest(s,A);
sc.close();
}
}
```

Output:

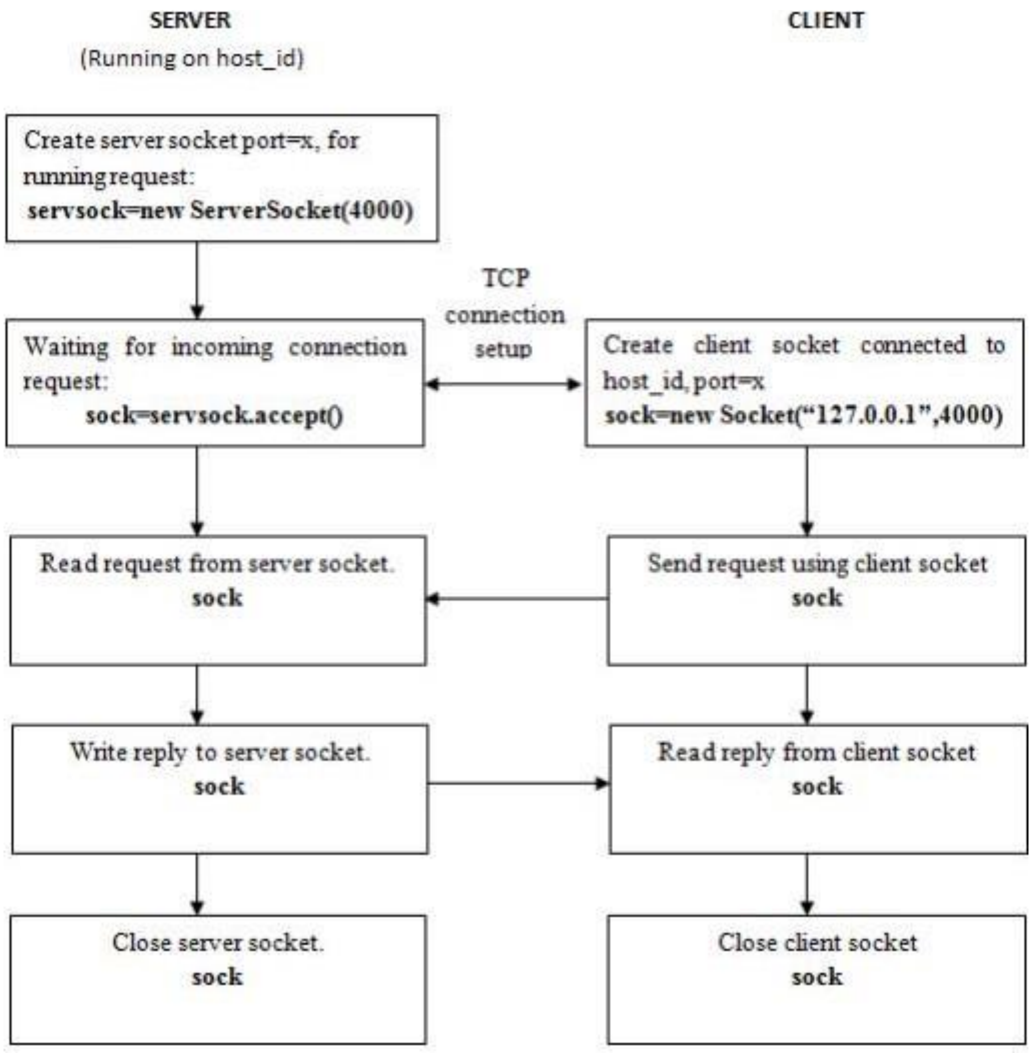
```
Enter the number of vertices
4
Enter the adjacency matrix
0
5
0
0
5
0
3
4
0
3
0
2
0
4
2
0
```

```
Enter the source vertex

2

Distance of source 2 to 1 is 5

Distance of source 2 to 2 is 0

Distance of source 2 to 3 is 3

Distance of source 2 to 4 is 4
```

Program- 7

Using TCP/IP sockets, write a client – server program to make the client send the file name and to make the server send back the contents of the requested file if present.

Socket is an interface which enables the client and the server to communicate and pass on information from one another. Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server. When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.


SERVER
(Running on host_id)
CLIENT
Create server socket port=x, for running request:
servsock=new ServerSocket(4000)
TCP connection setup
Waiting for incoming connection request:
sock=servsock.accept()
Create client socket connected to host_id, port=x
sock=new Socket("127.0.0.1",4000)
Read request from server socket.
sock
Send request using client socket
sock
Write reply to server socket.
sock
Read reply from client socket
sock
Close server socket.
sock
Close client socket
sock

Client program

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.net.Socket;
import java.util.Scanner;
class Client
{
public static void main(String args[])throws Exception
{
String address = "";
Scanner sc=new Scanner(System.in);
System.out.println("Enter Server Address: ");
address=sc.nextLine();
//create the socket on port 5000
Socket s=new Socket(address,5000);
DataInputStream din=new DataInputStream(s.getInputStream());
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Send Get to start...");
String str="",filename="";
try
{
while(!str.equals("start"))
str=br.readLine();
dout.writeUTF(str);
dout.flush();
filename=din.readUTF();
System.out.println("Receving file: "+filename);
filename="client"+filename;
System.out.println("Saving as file: "+filename);
long sz=Long.parseLong(din.readUTF());
System.out.println ("File Size: "+(sz/(1024*1024))+" MB");
byte b[]=new byte [1024];
System.out.println("Receving file..");
FileOutputStream fos=new FileOutputStream(new File(filename),true);
long bytesRead;
do
{
bytesRead = din.read(b, 0, b.length);
fos.write(b,0,b.length);
}while(!(bytesRead<1024));
System.out.println("Completed");
fos.close();
dout.close();
s.close();
}
catch(EOFException e)
{
}
}
}
```

<u>SERVER PROGRAM</u>

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
public class Server
    public static void main(String[] args) {
        String filename;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter File Name: ");
        filename = sc.nextLine();
        sc.close();
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            System.out.println("Server started and waiting for a request...");
            while (true) {
                try (Socket socket = serverSocket.accept();
                     DataInputStream din = new DataInputStream(socket.getInputStream());
                     DataOutputStream dout = new DataOutputStream(socket.getOutputStream()))
{
                  System.out.println("Connected with " + socket.getInetAddress().toString())
                    String request = din.readUTF();
                    if (request.equals("start")) {
                        System.out.println("Received 'start' request.");
                        File file = new File(filename);
                        if (!file.exists()) {
                            System.out.println("File not found: " + filename);
                            dout.writeUTF("File not found");
                            dout.flush();
                            continue;  }
```

```
  // Send the filename and file size
            dout.writeUTF(filename);
            dout.flush();
            long fileSize = file.length();
            dout.writeUTF(Long.toString(fileSize));
            dout.flush();
            System.out.println("Sending file: " + filename);
            System.out.println("File size: " + (fileSize / (1024 * 1024)) + " MB");
            try (FileInputStream fin = new FileInputStream(file)) {
              byte[] buffer = new byte[1024];
              int bytesRead;
              while ((bytesRead = fin.read(buffer)) != -1) {
                dout.write(buffer, 0, bytesRead);
                dout.flush();
              }
            }
            System.out.println("File sent successfully.");
          } else {
            dout.writeUTF("Invalid request");
            dout.flush();
          }
      } catch (IOException e) {
          e.printStackTrace();
          System.out.println("An error occurred while handling the client request.");
      }
  }
   } catch (IOException e) {
  e.printStackTrace();
  System.out.println("Failed to start the server.");
} }}
```

Note: Create two different files Client.java and Server.java. Follow the steps given:

1. Open a terminal run the server program and provide the filename to send

2. Open one more terminal run the client program and provide the IP address of the server. We can give localhost address “127.0.0.1” as it is running on same machine or give the IP address of the machine.

3. Send any start bit to start sending file.

4. Refer https://www.tutorialspoint.com/java/java_networking.htm for all the parameters, methods description in socket communication.

OUTPUT:

AT SERVER SIDE

```
user@user-OptiPlex-3050:~/Desktop$ javac Server.java
user@user-OptiPlex-3050:~/Desktop$ java Server
Enter File Name:
flower.jpg
Server started and waiting for a request...
Connected with /127.0.0.1
Received &apos;start&apos; request.
Sending file: flower.jpg
File size: 0 MB
File sent successfully
```

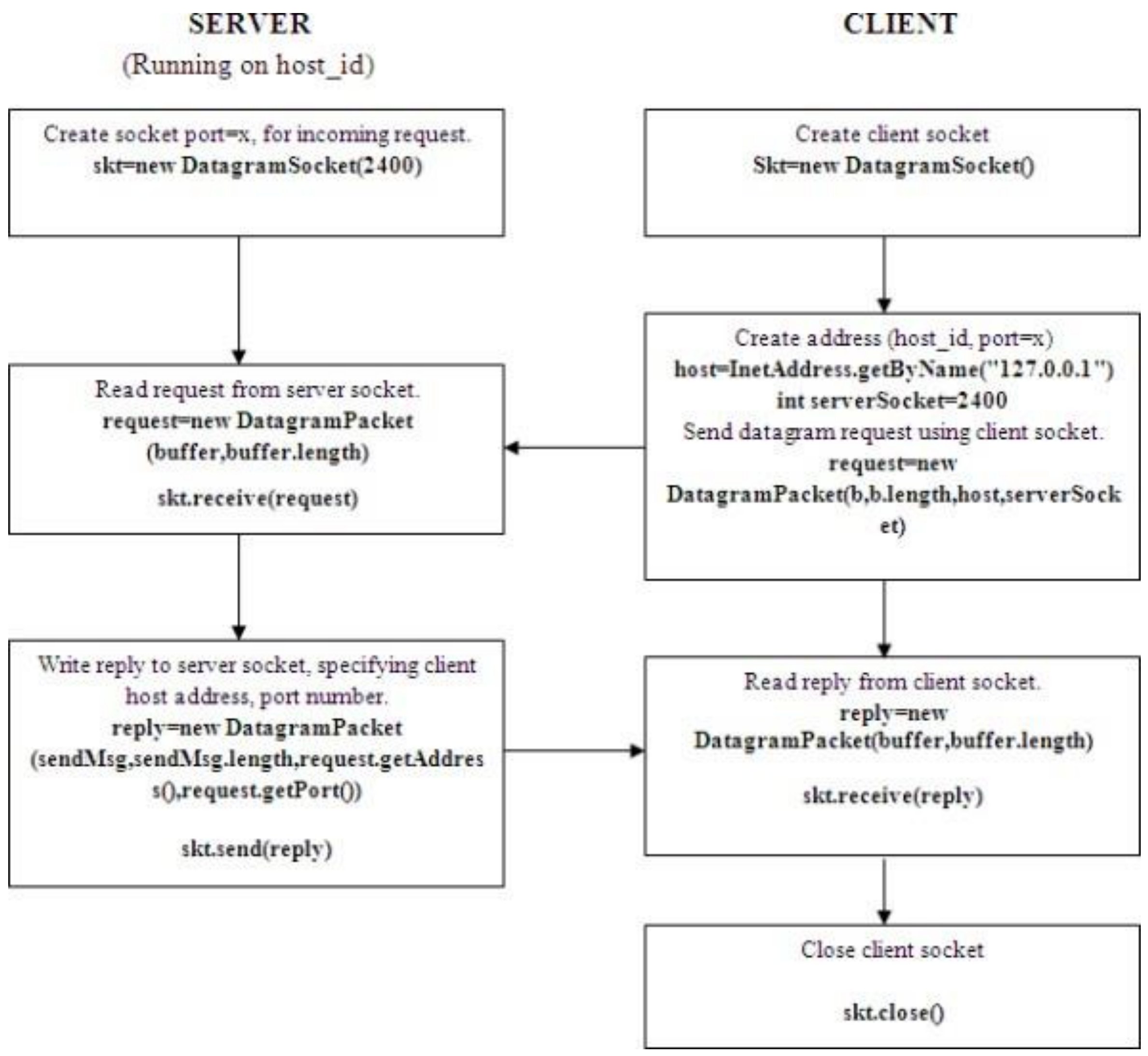
AT CLIENT SIDE

```
user@user-OptiPlex-3050:~/Desktop$ javac Client.java
user@user-OptiPlex-3050:~/Desktop$ java Client
Enter Server Address:
127.0.0.1
Send Get to start...
start
Receving file: flower.jpg
Saving as file: clientflower.jpg
File Size: 0 MB
Receving file..
Completed
```

Program- 8

Write a program on datagram socket for client/server to display the messages on client side, typed at the server side.

A datagram socket is the one for sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.


SERVER
(Running on host_id)
CLIENT
Create socket port=x, for incoming request.
skt=new DatagramSocket(2400)
Create client socket
Skt=new DatagramSocket()
Read request from server socket.
request=new DatagramPacket (buffer,buffer.length)
skt.receive(request)
Create address (host_id, port=x)
host=InetAddress.getByName("127.0.0.1")
int serverSocket=2400
Send datagram request using client socket.
request=new DatagramPacket(b,b.length,host,serverSocket)
Write reply to server socket, specifying client host address, port number.
reply=new DatagramPacket (sendMsg,sendMsg.length,request.getAddress(),request.getPort())
skt.send(reply)
Read reply from client socket.
reply=new DatagramPacket(buffer,buffer.length)
skt.receive(reply)
Close client socket
skt.close()

SourceCode:

UDP CLIENT

```
import java.io.*;
import java.net.*;
public class UDPC
{
public static void main(String[] args)
{
DatagramSocket skt;
try
{
skt=new DatagramSocket();
String msg= "text message ";
byte[] b = msg.getBytes();
InetAddress host=InetAddress.getByName("127.0.0.1");
int serverSocket=6788;
DatagramPacket request =new DatagramPacket (b,b.length,host,serverSocket);
skt.send(request);
byte[] buffer =new byte[1000];
DatagramPacket reply= new DatagramPacket(buffer,buffer.length);
skt.receive(reply);
System.out.println("client received:" +new String(reply.getData()));
skt.close();
}
catch(Exception ex)
{
}
}
}
```

UDP SERVER

```
import java.io.*;
import java.net.*;
public class UDPS
{
public static void main(String[] args)
{
DatagramSocket skt=null;
try
{
skt=new DatagramSocket(6788);
byte[] buffer = new byte[1000];
while(true)
{
DatagramPacket request = new DatagramPacket(buffer,buffer.length);
skt.receive(request);
String[] message = (new String(request.getData())).split(" ");
byte[] sendMsg= (message[1]+ " server processed").getBytes();
DatagramPacket reply = new
DatagramPacket(sendMsg,sendMsg.length,request.getAddress(),request.getPort());
skt.send(reply);
}
}
catch(Exception ex)
{
}
}
}
```

OUTPUT

SERVER SIDE

```
ser@user-OptiPlex-3050:~$ cd Desktop
user@user-OptiPlex-3050:~/Desktop$ javac UDPS.java
user@user-OptiPlex-3050:~/Desktop$ java UDPS
```

CLIENT SIDE

```
user@user-OptiPlex-3050:~/Desktop$ javac UDPC.java
user@user-OptiPlex-3050:~/Desktop$ java UDPC
client received:message server processed
```

Program- 9

RSA is an example of public key cryptography. It was developed by Rivest, Shamir and Adelman. The RSA algorithm can be used for both public key encryption and digital signatures. Its security is based on the difficulty of factoring large integers.

The RSA algorithm's efficiency requires a fast method for performing the modular exponentiation operation. A less efficient, conventional method includes raising anumber (the input) to a power (the secret or public key of the algorithm, denoted e and d, respectively) and taking the remainder of the division with N. A straight-forward implementation performs these two steps of the operation sequentially: first, raise it to the power and second, apply modulo. The RSA algorithm comprises of three steps, which are depicted below:

Key Generation Algorithm

1. Generate two large random primes, p and q, of approximately equal size such that their product n = p*q
2. Compute n = p*q and Euler's totient function (φ) phi(n) = (p-1)(q-1).
3. Choose an integer e, 1 < e < phi, such that gcd(e, phi) = 1.
4. Compute the secret exponent d, 1 < d < phi, such that e*d ≡ 1 (mod phi).
5. The public key is (e, n) and the private key is (d, n). The values of p, q, and phi should also be kept secret.

Encryption

Sender A does the following:-

1. Using the public key (e,n)
2. Represents the plaintext message as a positive integer M
3. Computes the cipher text C = M^e mod n.
4. Sends the cipher text C to B (Receiver).

Decryption

Recipient B does the following:-

1. Uses his private key (d, n) to compute M = C^d mod n.
2. Extracts the plaintext from the integer representative m.

Source Code:
```
import java.io.DataInputStream;
import java.io.IOException;
import java.math.BigInteger;
import java.util.Random;
public class RSA
{
private BigInteger p,q,N,phi,e,d;
private int bitlength=1024;
private Random r;
public RSA()
{
r=new Random();
p=BigInteger.probablePrime(bitlength,r);
q=BigInteger.probablePrime(bitlength,r);
System.out.println("Prime number p is"+p);
System.out.println("prime number q is"+q);
N=p.multiply(q);
phi=p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
e=BigInteger.probablePrime(bitlength/2,r);
while(phi.gcd(e).compareTo(BigInteger.ONE)>0&&e.compareTo(phi)<0)
{
e.add(BigInteger.ONE);
}
System.out.println("Public key is"+e);
d=e.modInverse(phi);
System.out.println("Private key is"+d);
}
public RSA(BigInteger e,BigInteger d,BigInteger N)
{
this.e=e;
this.d=d;
this.N=N;
}
public static void main(String[] args)throws IOException
{
RSA rsa=new RSA();
DataInputStream in=new DataInputStream(System.in);
String testString;
System.out.println("Enter the plain text:");
testString=in.readLine();
System.out.println("Encrypting string:"+testString);
System.out.println("string in bytes:"+bytesToString(testString.getBytes()));
byte[] encrypted=rsa.encrypt(testString.getBytes());
byte[] decrypted=rsa.decrypt(encrypted);
System.out.println("Dcrypting Bytes:"+bytesToString(decrypted));
System.out.println("Dcrypted string:"+new String(decrypted));
}
private static String bytesToString(byte[] encrypted)
{
String test=" ";
for(byte b:encrypted)
{
test+=Byte.toString(b);
}
return test;
}
public byte[]encrypt(byte[]message)
{
return(new BigInteger(message)).modPow(e,N).toByteArray();
}
```

```
public byte[]decrypt(byte[]message)

{

return(new BigInteger(message)).modPow(d,N).toByteArray();

}

}
```

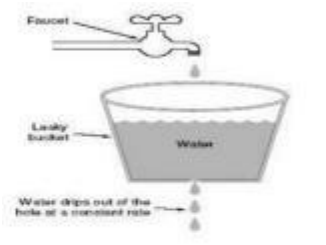
OUTPUT

```
user@user-OptiPlex-3050:~/Desktop$ javac RSA.java
Note: RSA.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
user@user-OptiPlex-3050:~/Desktop$ java RSA
Prime number p
is94340415559667604196396997194355500620617261892363799333186130581555030559022258412648913166938447749
396172942836509622095538500548337654390771331149455795514743327288673267997616067107077271234272699473
316258433539818381316108677253948520717066079742130872191391026876978938301595297841729450104777515744
329
prime number q
is16295987633597507022079035792651362708653034086451179412791395159018434039686574419084445457091309019
958011356682077762167116800690512861094823685729412493004985480695151418364557357441519382087492354812
303281204443293129284873661565419734726397689991056674514937464946198124476058951341663906940428877098
9747
Public key
is12535317231931615097535292679185522142398124359148578631491867555660541594908142243803236928383564409
0912254854167494838496767313072925642546688763477901 13
Private key
is13447068803454147250016548099154994288001205002052029380889366442542448009766902483706511629378778814
278990034496738434310061277380078992204031368701787218848258622194596869433341569244363873935384095105
592616511001433759869739164863728336417128555891622497173940668066194911692005738411199165928696303164
523779310027829195066723798832917749431105127613599172760980717753017355228063125668506068598724533589
104327931657316929758777163359133663584946773520040126746860463171317362804721343414191734927496021690
915795912693044571397754154768310721320751218537689010905006735039126790700979397990051206085125951747
130833
Enter the plain text:
Hello everyone
Encrypting string:Hello everyone
string in bytes: 72101108108111321011181011141211111110101
Dcrypting Bytes: 72101108108111321011181011141211111110101
Dcrypted string:Hello everyone
```

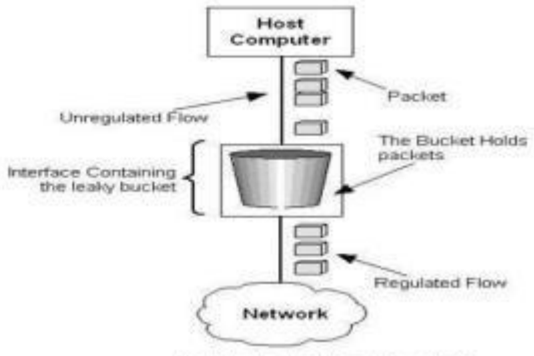
Program- 10

Write a program for congestion control using leaky bucket algorithm.

The main concept of the leaky bucket algorithm is that the output data flow remains constant despite the variant input traffic, such as the water flow in a bucket with a small hole at the bottom. In case the bucket contains water (or packets) then the output flow follows a constant rate, while if the bucket is full any additional load will be lost because of spillover. In a similar way if the bucket is empty the output will be zero. From network perspective, leaky bucket consists of a finite queue (bucket) where all the incoming packets are stored in case there is space in the queue, otherwise the packets are discarded. In order to regulate the output flow, leaky bucket transmits one packet from the queue in a fixed time (e.g. at every clock tick). In the following figure we can notice the main rationale of leaky bucket algorithm, for both the two approaches (e.g. leaky bucket with water (a) and with packets (b)).



(a) A leaky bucket with water



(b) A leaky bucket with packets

While leaky bucket eliminates completely bursty traffic by regulating the incoming data flow its main drawback is that it drops packets if the bucket is full. Also, it doesn't take into account the idle process of the sender which means that if the host doesn't transmit data for some time the bucket becomes empty without permitting the transmission of any packet.

Source Code

```
import java.util.*;
public class leaky
{
static int min(int x,int y)
{
if(x<y)
return x;
else
return y;
}
public static void main(String[] args)
{ int drop=0,mini,nsec,cap,count=0,i,process;
int inp[]=new int[25];
Scanner sc=new Scanner(System.in);
System.out.println("Enter The Bucket Size\n");
cap= sc.nextInt();
System.out.println("Enter The Operation Rate\n");
process= sc.nextInt();
System.out.println("Enter The No. Of Seconds You Want To Stimulate\n");
nsec=sc.nextInt();
for(i=0;i<nsec;i++)
{
System.out.println("Enter The Size Of The Packet Entering At 1 sec:");
inp[i] = sc.nextInt();
}
System.out.println("\nSecond | Packet Recieved | Packet Sent | Packet Left |Packet Dropped|\n");
System.out.println("------------------------------------------------\n");
for(i=0;i<nsec;i++)
{ count+=inp[i];
if(count>cap)
{ drop=count-cap;
count=cap;
}
System.out.print(i+1);
System.out.print("\t\t"+inp[i]);
mini=min(count,process);
System.out.print("\t\t"+mini);
count=count-mini;
System.out.print("\t\t"+count);
System.out.print("\t\t"+drop);
drop=0;
System.out.println();
}
for(;count!=0;i++)
{
if(count>cap)
{
drop=count-cap;
count=cap;
}
System.out.print(i+1);
System.out.print("\t\t0");
mini=min(count,process);
System.out.print("\t\t"+mini);
count=count-mini;
System.out.print("\t\t"+count);
System.out.print("\t\t"+drop);
System.out.println();
}
}
}
```

OUTPUT

```
user@user-OptiPlex-3050:~/Desktop$ javac leaky.java
user@user-OptiPlex-3050:~/Desktop$ java leaky
Enter The Bucket Size

5
Enter The Operation Rate

2
Enter The No. Of Seconds You Want To Stimulate

3
Enter The Size Of The Packet Entering At 1 sec:
2
Enter The Size Of The Packet Entering At 1 sec:
3
Enter The Size Of The Packet Entering At 1 sec:
5

Second | Packet Recieved | Packet Sent | Packet Left |Packet Dropped|

--------------------------------------------------------------------------------

1       2       2       0       0
2       3       2       1       0
3       5       2       3       1
4       0       2       1       0
5       0       1       0       0
```