Java Programs

Java is a general-purpose computer programming language that is simple, concurrent, class-based, object-oriented language. The compiled Java code can run on all platforms that support Java without the need for recompilation hence Java is called as "write once, run anywhere" (WORA).The Java compiled intermediate output called “byte-code” that can run on any Java virtual machine (JVM) regardless of computer architecture. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

In Linux operating system Java libraries are preinstalled. It’s very easy and convenient to compile and run Java programs in Linux environment. To compile and run Java Program is a two-step process:

1. Compile Java Program from Command Prompt

   [root@host ~]# javac Filename.java

The Java compiler (Javac) compiles java program and generates a byte-code with the same file name and .class extension.

2. Run Java program from Command Prompt

   [root@host ~]# java Filename

The java interpreter (Java) runs the byte-code and gives the respective output. It is important to note that in above command we have omitted the .class suffix of the byte- code (Filename.class).

Experiment No:3 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 connections are checked with 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, he basic mathematical process is always the same:

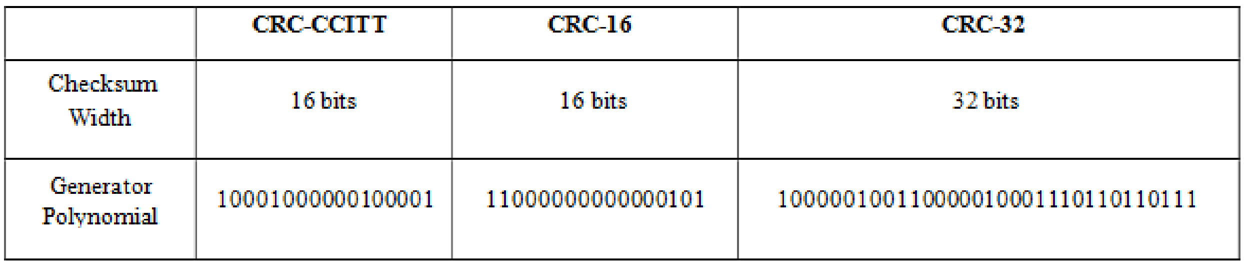
• The message bits are appended with *c* zero bits; this *augmented message* is the dividend

• 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

```
import java.io.*;

    class Crc
    {
        public static void main(String args[]) throws IOException
        {
                BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
                int[ ] data;
                int[ ]div;
                int[ ]divisor;
                int[ ]rem;
                int[ ] crc;
                int data_bits, divisor_bits, tot_length;

                System.out.println("Enter number of data bits : ");
                data_bits=Integer.parseInt(br.readLine());
                data=new int[data_bits];

                System.out.println("Enter data bits : ");


                  for(int i=0; i< data_bits; i++)

                        data[i]=Integer.parseInt(br.readLine());
                        System.out.println("Enter number of bits in divisor : ");
                        divisor_bits=Integer.parseInt(br.readLine());
                        divisor=new int[divisor_bits];
                        System.out.println("Enter Divisor bits : ");
                for(int i=0; i<divisor_bits; i++)
                        divisor[i]=Integer.parseInt(br.readLine());


                /* System.out.print("Data bits are : ");
for(int i=0; i< data_bits; i++)

                       System.out.print(data[i]);
                       System.out.println();

                       System.out.print("divisor bits are : ");
               for(int i=0; i< divisor_bits; i++)
                       System.out.print(divisor[i]);
                       System.out.println();


               */ tot_length=data_bits+divisor_bits-1;


               div=new int[tot_length];
               rem=new int[tot_length];
               crc=new int[tot_length];
               /*------------------ CRC GENERATION-----------------------*/
               for(int i=0;i<data.length;i++)
                       div[i]=data[i];


               System.out.print("Dividend (after appending 0's) are : ");
               for(int i=0; i< div.length; i++)
```

```
                System.out.print(div[i]);
                System.out.println();

            for(int j=0; j<div.length; j++){
                rem[j] = div[j];
        }
        rem=divide(div, divisor, rem);

        for(int i=0;i<div.length;i++) //append dividend and ramainder
        {
                    crc[i]=(div[i]^rem[i]);
        }

                System.out.println();

                System.out.println("CRC code");



              for(int i=0;i<crc.length;i++)
              System.out.print(crc[i]);
        /*-------------------ERROR DETECTION---------------------*/
           System.out.println();
           System.out.println("Enter CRC code of "+tot_length+" bits : ");
           for(int i=0; i<crc.length; i++)
                crc[i]=Integer.parseInt(br.readLine());


        /* System.out.print("crc bits are : ");
        for(int i=0; i< crc.length; i++)
           System.out.print(crc[i]);
           System.out.println();
          */
        for(int j=0; j<crc.length; j++){

rem[j] = crc[j];
}

rem=divide(crc, divisor, rem);

for(int i=0; i< rem.length; i++)
{
if(rem[i]!=0)
{
System.out.println("Error");
break;
}
if(i==rem.length-1)
System.out.println("No Error");
}
```

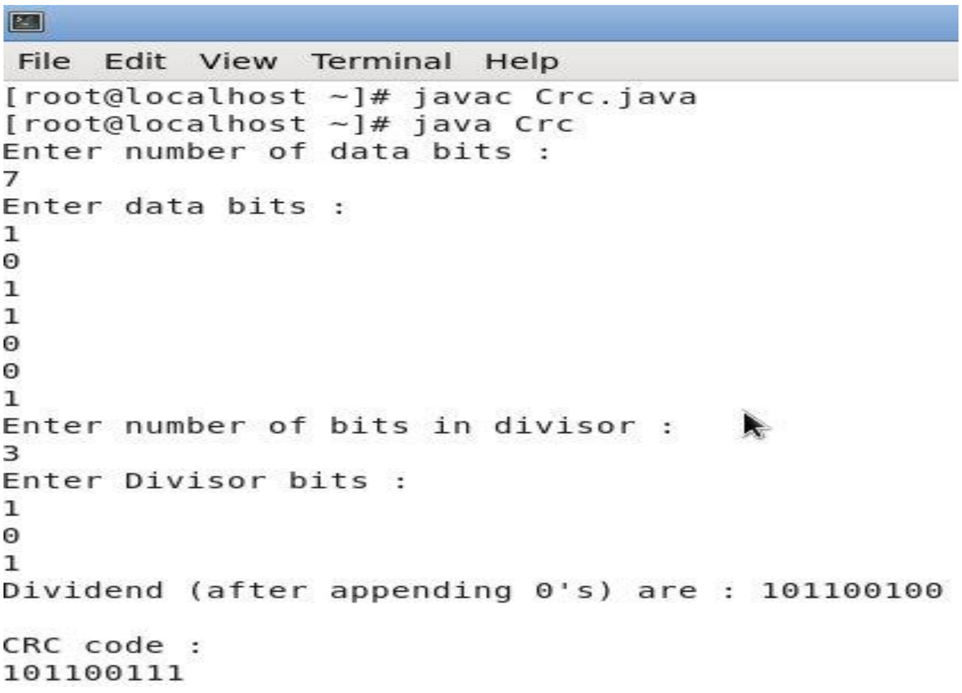
```
System.out.println("THANK YOU.... :)");
}

static int[] divide(int div[],int divisor[], int rem[])
{
int cur=0;
while(true)
{
for(int i=0;i<divisor.length;i++)
rem[cur+i] =(rem[cur+i]^divisor[i]);
while(rem[cur] = = 0 && cur! = rem.length-1)
cur++;

if((rem.length-cur)<divisor.length)
break;
}
return rem;
}
}
```

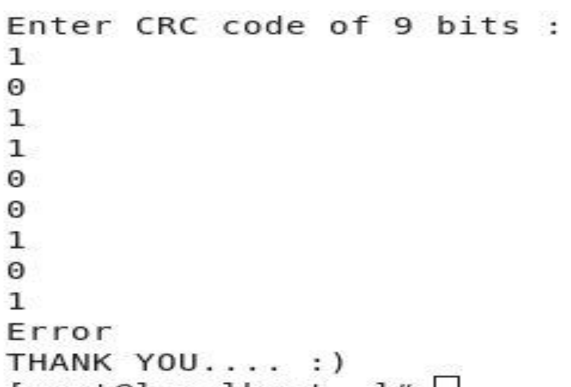
Output:

[root@localhost ~]# vi Crc.java



```
[root@localhost ~]# javac Crc.java
[root@localhost ~]# java Crc
Enter number of data bits :
7
Enter data bits :
1
0
1
1
0
0
1
Enter number of bits in divisor :
3
Enter Divisor bits :
1
0
1
Dividend (after appending 0's) are : 101100100

CRC code :
101100111
```



```
Enter CRC code of 9 bits :
1
0
1
1
0
0
1
0
1
Error
THANK YOU.... :)
```

Experiment No:5 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

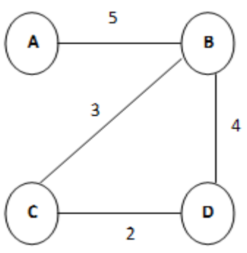
Source code:

```
import java.util.Scanner;
public class BellmanFord
```

```
{
private int D[];
private int 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++)
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();
}
}
```

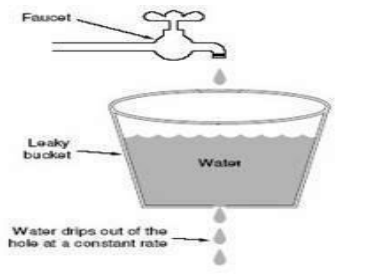
Input graph:



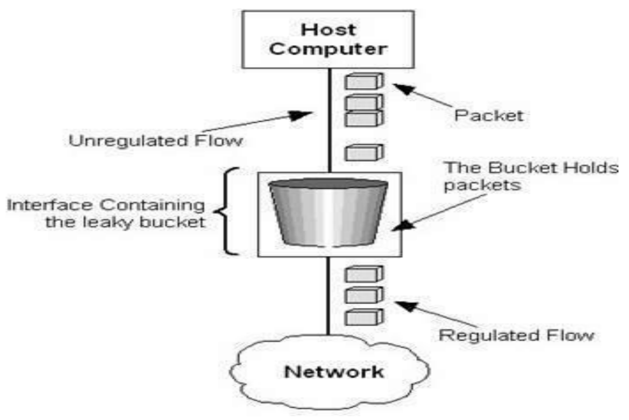
Experiment No:7. Write a program for congestion control using leaky bucket algorithm.

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.Scanner;
public class LeakyBucket {
public static int min(int a, int b)
{
        if(a<b)
                return a;
        else
```

```
                return b;
        }
        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
            int cap, oprt, nsec, cont, i=0,dr=0, x, res;
            int [] inp = new int [25];
            int ch;
                System.out.println("\n\nLEAKY BUCKET ALGORITHM\n");
                System.out.println("\nEnter bucket size :   ");
                cap = sc.nextInt();
                System.out.println("\nEnter output rate (no..of pkts/sec)  :  ");
                oprt = sc.nextInt();

          do{
                System.out.println("\nEnter the no..of packets entering at second:"+(i+1));
                inp[i++] = sc.nextInt();
                System.out.println("Enter 1 to insert packets or 0 to quit  :   ");
                ch = sc.nextInt();
             }  while(ch==1);
              nsec=i;
          System.out.println("\nSecond : Packets recvd to  bucket : Packets sent : In bucket: Dropped\n");

                for(cont=i=0;  (cont>0) || (i<nsec) ; i++)
                {
                        System.out.print("    :"+(i+1));
                        System.out.print("     \t: "+inp[i]);
                        res = min(cont+inp[i],oprt);
                        System.out.print("     \t: "+res);
                        if((x=cont+inp[i]-oprt)>0)
                        {
                                if(x>cap)
                                {
                                        cont=cap;
                                        dr=x-cap;
                                }
                                else
                                {       cont=x;
                                        dr=0;
                                }
                        }
                        else
                        cont=0;
                        System.out.print("   \t\t:"+cont);
                System.out.print("   \t\t:"+dr+"\n");
        }
        }
        }
```

Output:

```
LEAKY BUCKET ALGORITHM

Enter bucket size :
3
Enter output rate (no..of pkts/sec)  :
2
Enter the no..of packets entering at second:1
6
Enter 1 to insert packets or 0 to quit  :
0

Second :   Packets sent :   Packets recvd :   In bucket:    Dropped
```

```
(1)         : 6         : 2          :3         1
(2)         : 0         : 2          :1         0
(3)         : 0         : 1          :0         0
```