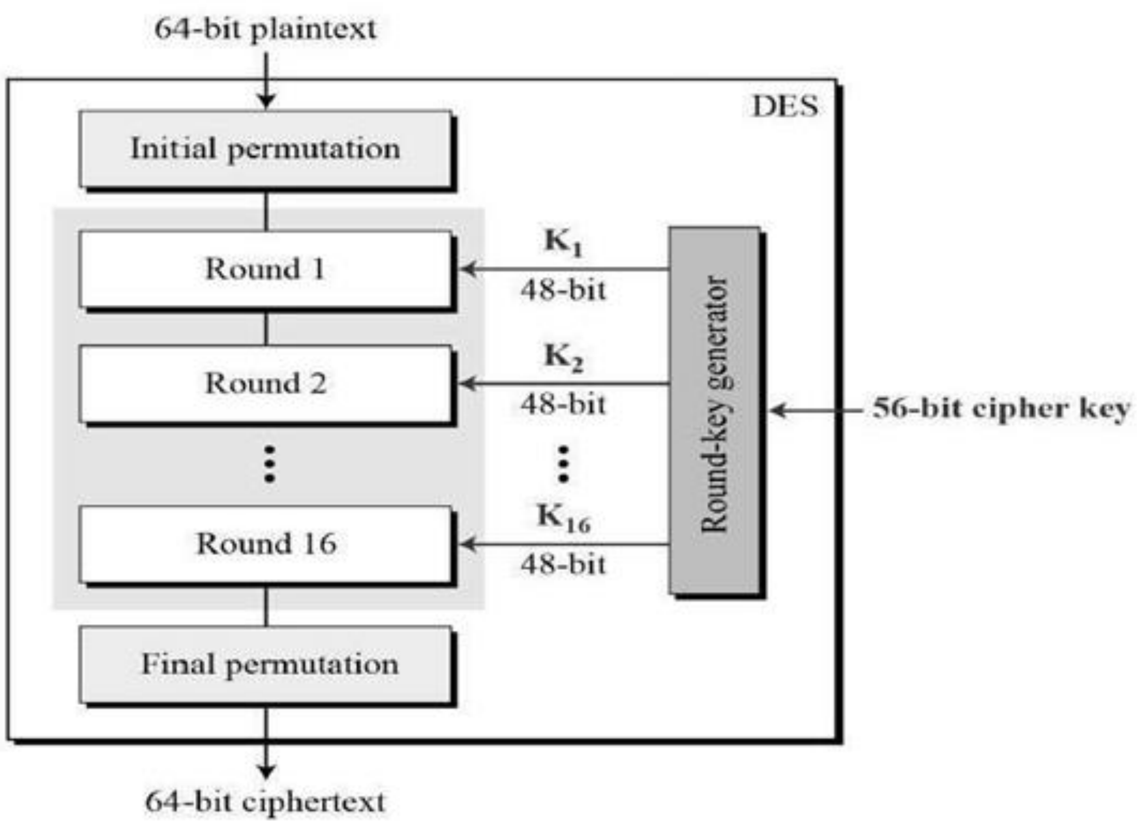
EX.NO:2(a) **DATA ENCRYPTION STANDARD (DES) ALGORITHM**
DATE:

AIM:

To develop a program to implement Data Encryption Standard for encryption and decryption.

ALGORITHM DESCRIPTION:

- The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).
- DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit.
- Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only).
- General Structure of DES is depicted in the following illustration



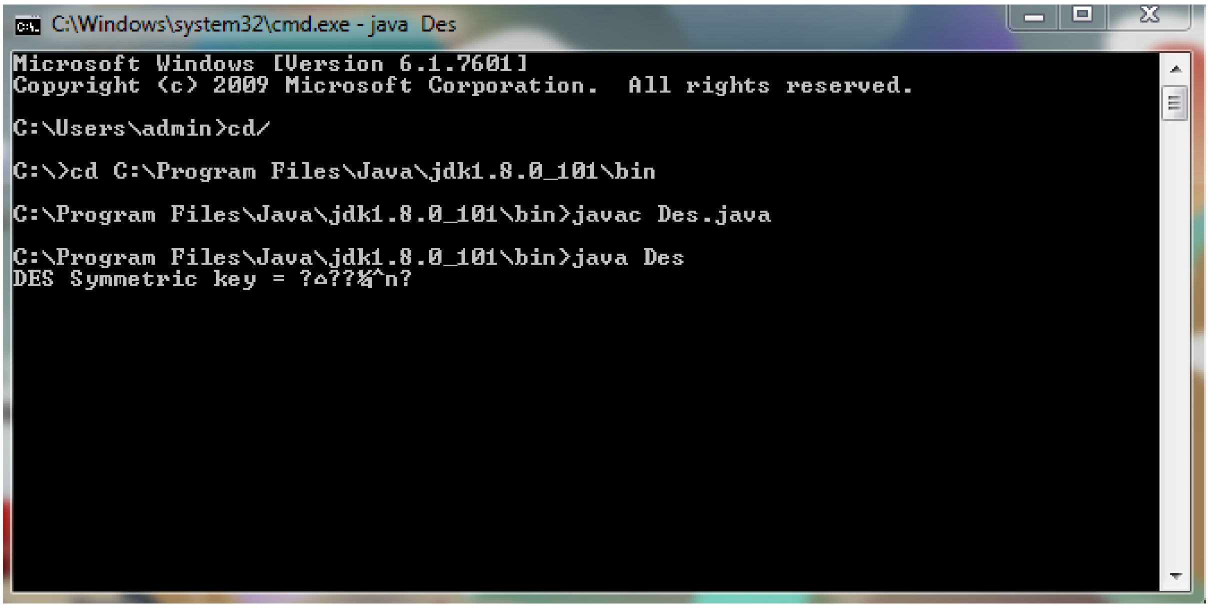
DES ALGORITHM

PROGRAM

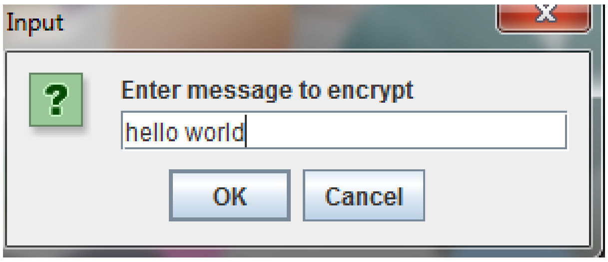
```
import javax.swing.*;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Random ;
class Des {
byte[] skey = new byte[1000];
String skeyString;
static byte[] raw;
String inputMessage,encryptedData,decryptedMessage;
public Des() {
try {
generateSymmetricKey();
inputMessage=JOptionPane.showInputDialog(null,"Enter message to encrypt");
byte[] ibyte = inputMessage.getBytes();
byte[] ebyte=encrypt(raw, ibyte);
String encryptedData = new String(ebyte);
System.out.println("Encrypted message "+encryptedData);
JOptionPane.showMessageDialog(null,"Encrypted Data "+"\n"+encryptedData);
byte[] dbyte= decrypt(raw,ebyte);
String decryptedMessage = new String(dbyte);
System.out.println("Decrypted message "+decryptedMessage);
JOptionPane.showMessageDialog(null,"Decrypted Data "+"\n"+decryptedMessage);
} catch(Exception e) {
System.out.println(e);
}}
void generateSymmetricKey() {
try {
Random r = new Random();
int num = r.nextInt(10000);
```

```
String knum = String.valueOf(num);
byte[] knumb = knum.getBytes();
skey=getRawKey(knumb);      //to get the key
skeyString = new String(skey);
System.out.println("DES Symmetric key = "+skeyString);
} catch(Exception e) {
System.out.println(e);
}}
private static byte[] getRawKey(byte[] seed) throws Exception {
KeyGenerator kgen = KeyGenerator.getInstance("DES");      //generates the key
SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
sr.setSeed(seed);
kgen.init(56, sr);
SecretKey skey = kgen.generateKey();
raw = skey.getEncoded();
return raw;
}
private static byte[] encrypt(byte[] raw, byte[] clear) throws Exception {
SecretKeySpec skeySpec = new SecretKeySpec(raw, "DES");
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
byte[] encrypted = cipher.doFinal(clear);
return encrypted;
}
private static byte[] decrypt(byte[] raw, byte[] encrypted) throws Exception {
SecretKeySpec skeySpec = new SecretKeySpec(raw, "DES");
Cipher cipher = Cipher.getInstance("DES");
cipher.init(Cipher.DECRYPT_MODE, skeySpec);
byte[] decrypted = cipher.doFinal(encrypted);
return decrypted;
}
public static void main(String args[]) {
Des des = new Des();
}
}
```

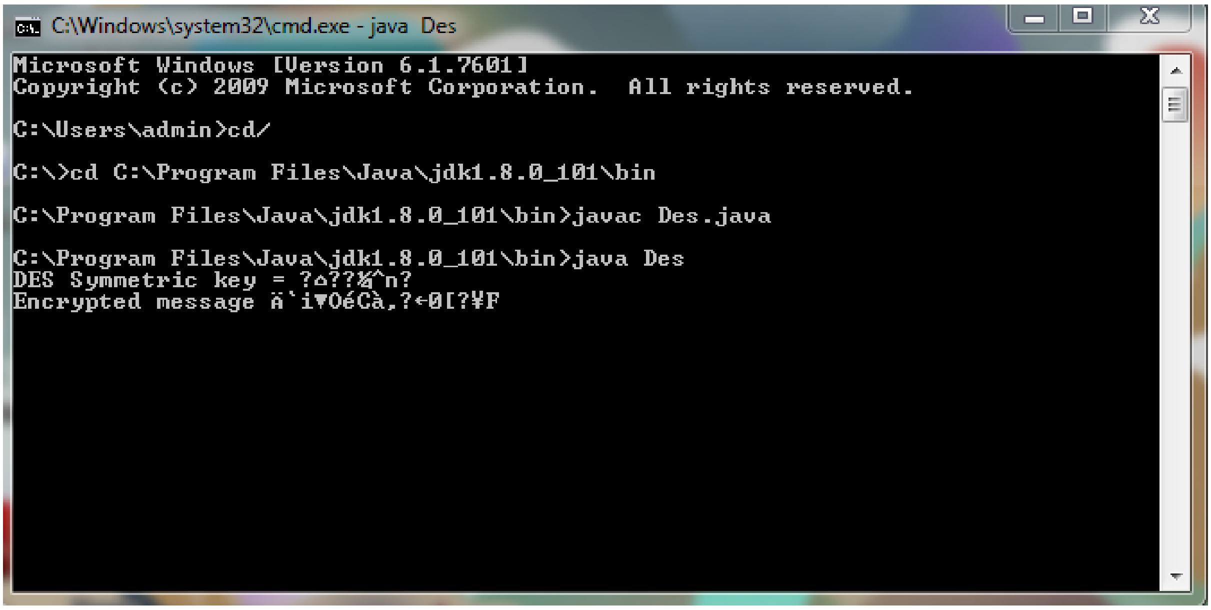
OUTPUT:

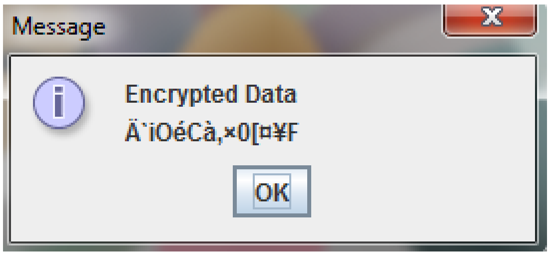


Input String to encrypt:

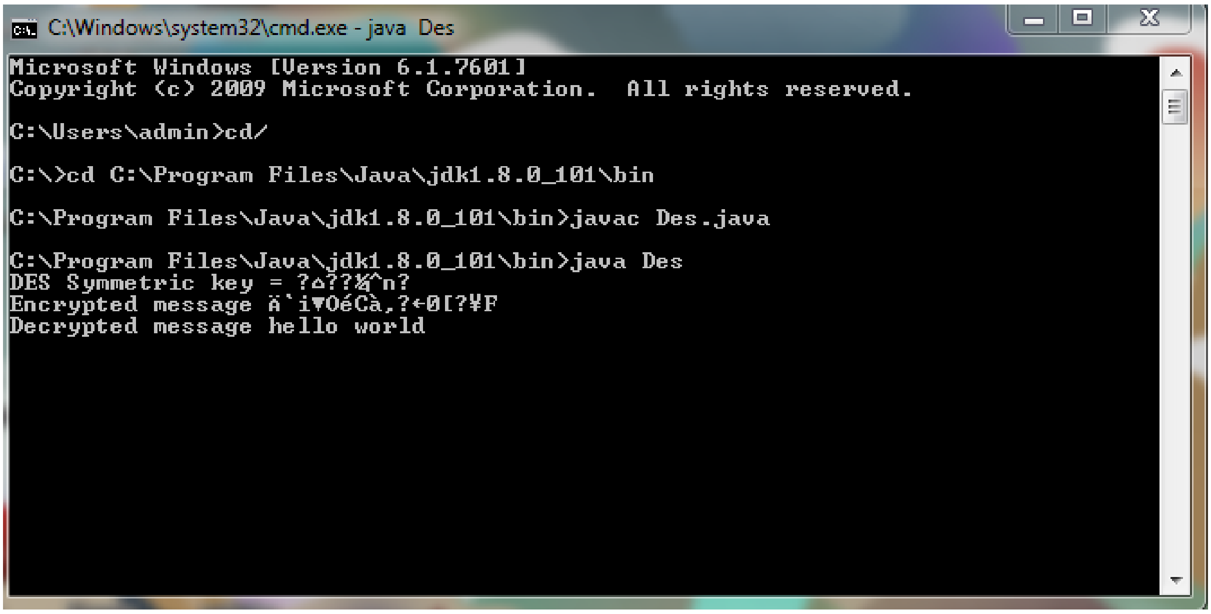


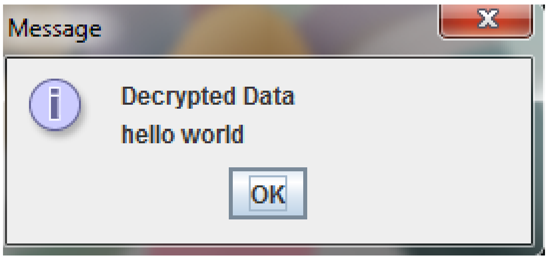
Encrypted data:





Decrypted data:





RESULT:

Thus the java program to implement DES Algorithm is executed and the output is verified.

EX.NO.:2(b) **RSA ALGORITHM**
DATE:

AIM:

To develop a program to implement RSA algorithm for encryption and decryption.

INTRODUCTION:

RSA cryptosystem is one the initial system. It remains most employed cryptosystem even today. The system was invented by three scholars **Ron Rivest, Adi Shamir,** and **Len Adleman** and hence, it is termed as RSA cryptosystem. The two aspects of the RSA cryptosystem, firstly generation of key pair and secondly encryption-decryption algorithms

ALGORITHM DESCRIPTION:

Generation of RSA Key Pair

- Each person or a party who desires to participate in communication using encryption needs to generate a pair of keys, namely public key and private key.
- The process followed in the generation of keys is described below −
- Generate the RSA modulus (n)
    - Select two large primes, p and q.
    - Calculate n=p*q. For strong unbreakable encryption, let n be a large number, typically a minimum of 512 bits.
- Find Derived Number (e)
    - Number e must be greater than 1 and less than (p − 1)(q − 1).
    - There must be no common factor for e and (p − 1)(q − 1) except for 1. In other words two numbers e and (p – 1)(q – 1) are coprime.
- Form the public key
    - The pair of numbers (n, e) form the RSA public key and is made public. Interestingly, though n is part of the public key, difficulty in factorizing a large prime number ensures that attacker cannot find in finite time the two primes (p & q) used to obtain n. This is strength of RSA.
- Generate the private key
    - Private Key d is calculated from p, q, and e. For given n and e, there is unique number d.
    - Number d is the inverse of e modulo (p - 1)(q – 1). This means that d is the number less than (p - 1)(q - 1) such that when multiplied by e, it is equal to 1 modulo (p - 1)(q - 1).
- This relationship is written mathematically as follows ed = 1 mod (p − 1)(q − 1)
- The Extended Euclidean Algorithm takes p, q, and e as input and gives d as output.

IMPLEMENTATION OF RSA ALGORITHM

PROGRAM

```
import java.io.DataInputStream;
import java.io.IOException;
import java.math.BigInteger;
import java.util.Random;
public class RSA
{
    private BigInteger p;
    private BigInteger q;
    private BigInteger N;
    private BigInteger phi;
    private BigInteger e;
    private BigInteger d;
    private int bitlength = 1024;
    private Random r;
    public RSA()
    {
        r = new Random();
        p = BigInteger.probablePrime(bitlength, r);
        q = BigInteger.probablePrime(bitlength, r);
        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);
        }
        d = e.modInverse(phi);
    }
   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("Decrypting Bytes: " + bytesToString(decrypted));
        System.out.println("Decrypted 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

```
Output - RSA (run)

run:
Enter the plain text:
RSA_Algorithm
Encrypting String: RSA_Algorithm
String in Bytes: 8283659565108103111114105116104109
Decrypting Bytes: 8283659565108103111114105116104109
Decrypted String: RSA_Algorithm
BUILD SUCCESSFUL (total time: 7 seconds)
```

RESULT:

Thus the java program to implement RSA Algorithm is executed and the output is verified.

EX.NO.:2(c) **DIFFIEE HELLMAN KEY EXCHANGE ALGORITHM**
DATE:

AIM:

Develop a program to implement Diffie Hellman Key Exchange Algorithm for encryption and Decryption.

ALGORITHM DESCRIPTION:

Diffie–Hellman key exchange (D–H) is a specific method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

ALGORITHM

Global Public Elements:

Let q be a prime number and α is a primitive root of q.

1. User A Key Generation:

Select private XA where XA < q

Calculate public YA where $YA = \alpha^{XA} \bmod q$

2. User B Key Generation:

Select private XB where XB < q

Calculate public YB where $YB = \alpha^{XB} \bmod q$

3. Calculation of Secret Key by User A

$K = (YB)^{XA} \bmod q$

4. Calculation of Secret Key by User B:

$K = (YA)^{XB} \bmod q$

DIFFIEE-HELLMANN KEY EXCHANGE

PROGRAM

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
public class DeffieHellman {
public static void main(String[]args)throws IOException
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter prime number:");
BigInteger p=new BigInteger(br.readLine());
System.out.print("Enter primitive root of "+p+":");
BigInteger g=new BigInteger(br.readLine());
System.out.println("Enter value for x less than "+p+":");
BigInteger x=new BigInteger(br.readLine());
BigInteger R1=g.modPow(x,p);
System.out.println("R1="+R1);
System.out.print("Enter value for y less than "+p+":");
BigInteger y=new BigInteger(br.readLine());
BigInteger R2=g.modPow(y,p);
System.out.println("R2="+R2);
BigInteger k1=R2.modPow(x,p);
System.out.println("Key calculated at Alice's side:"+k1);
BigInteger k2=R1.modPow(y,p);
System.out.println("Key calculated at Bob's side:"+k2);
System.out.println("deffie hellman secret key Encryption has Taken");
}
}
```

OUTPUT:

Output - JavaApplication4 (run) #2

```
run:
Enter prime number:
11
Enter primitive root of 11:7
Enter value for x less than 11:
7
R1=6
Enter value for y less than 11:9
R2=8
Key calculated at Alice's side:2
Key calculated at Bob's side:2
deffie hellman secret key Encryption has Taken
BUILD SUCCESSFUL (total time: 22 seconds)
```

RESULT:

Thus the java program to implement Diffiee-Hellmann Algorithm is executed and the output is verified.

EX.NO.:2(d) **MESSAGE DIGEST ALGORITHM (MD5)**
DATE:

AIM:

To develop a program to implement Message Digest Algorithm.

ALGORITHM DESCRIPTION:

- The MD5 **message-digest algorithm** is a widely used **cryptographic hash function** producing a 128-bit (16-byte) **hash value**, typically expressed in text format as a 32-digit hexadecimal number.

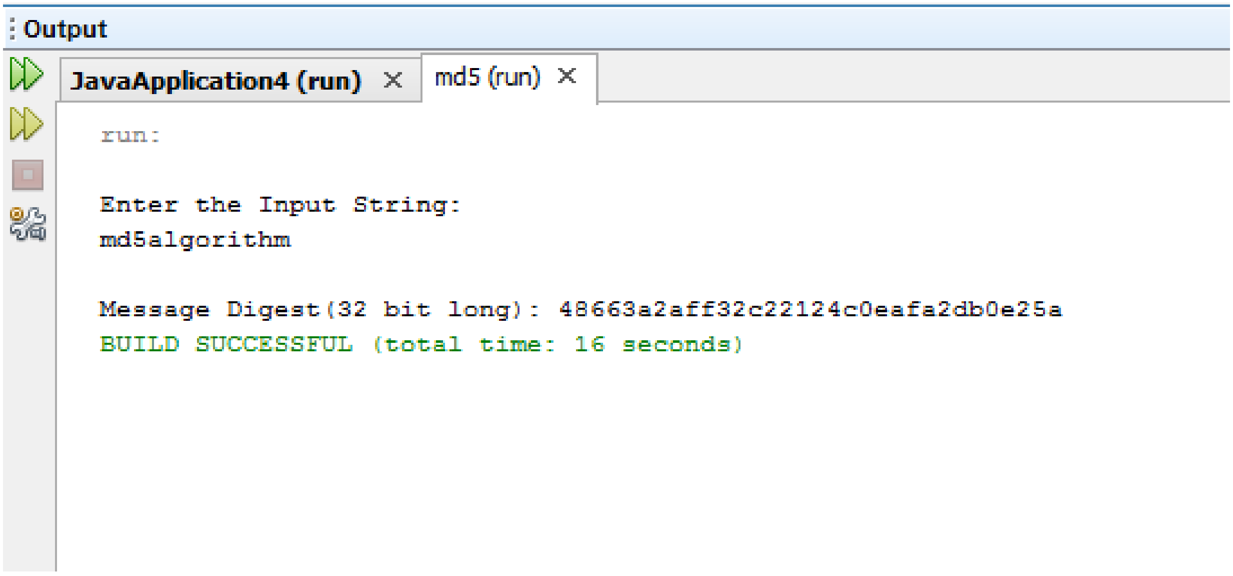
- MD5 has been utilized in a wide variety of cryptographic applications and is also commonly used to verify data integrity.

IMPLEMENTATION OF MD5 ALGORITHM

PROGRAM

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;
public class Md5 {
public static String getMD5(String input) {
try{
MessageDigest md = MessageDigest.getInstance("MD5");
byte[] messageDigest = md.digest(input.getBytes());
BigInteger number = new BigInteger(1, messageDigest);
String hashtext = number.toString(16);
while(hashtext.length() < 32) {
hashtext = "0"+ hashtext;
}
return hashtext;
}
catch(NoSuchAlgorithmException e)
{
throw new RuntimeException(e);
}
}
public static void main(String[] args) throws NoSuchAlgorithmException
{
Scanner s=new Scanner(System.in);
System.out.println("\nEnter the Input String: ");
String str=s.nextLine();
System.out.println("\nMessage Digest(32 bit long): "+getMD5(str));
}
}
```

OUTPUT:


Output
JavaApplication4 (run)
md5 (run)
run:
Enter the Input String:
md5algorithm
Message Digest(32 bit long): 48663a2aff32c22124c0eafa2db0e25a
BUILD SUCCESSFUL (total time: 16 seconds)

RESULT:

Thus the java program to implement MD5 Algorithm is executed and the output is verified.

EX.NO: 2e) **SECURE HASH FUNCTION (SHA-1)**
DATE:

AIM:

To develop a program to implement Secure Hash Algorithm (SHA-1)

ALGORITHM DESCRIPTION:

Secured Hash Algorithm-1 (SHA-1):

Step 1: Append Padding Bits….

Message is "padded" with a 1 and as many 0's as necessary to bring the message length to 64 bits less than an even multiple of 512.

Step 2: Append Length....

64 bits are appended to the end of the padded message. These bits hold the binary format of 64 bits indicating the length of the original message.

Step 3: Prepare Processing Functions….

SHA1 requires 80 processing functions defined as:

f(t;B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)
f(t;B,C,D) = B XOR C XOR D (20 <= t <= 39)
f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t<=59)
f(t;B,C,D) = B XOR C XOR D (60 <= t <= 79)

Step 4: Prepare Processing Constants....

SHA1 requires 80 processing constant words defined as:

K(t) = 0x5A827999 (0 <= t <= 19)
K(t) = 0x6ED9EBA1 (20 <= t <= 39)
K(t) = 0x8F1BBCDC (40 <= t <= 59)
K(t) = 0xCA62C1D6 (60 <= t <= 79)

Step 5: Initialize Buffers….

SHA1 requires 160 bits or 5 buffers of words (32 bits):

H0 = 0x67452301
H1 = 0xEFCDAB89
H2 = 0x98BADCFE
H3 = 0x10325476
H4 = 0xC3D2E1F0

Step 6: Processing Message in 512-bit blocks (L blocks in total message)….

This is the main task of SHA1 algorithm which loops through the padded and appended message in 512-bit blocks.

Input and predefined functions: M[1, 2, ..., L]: Blocks of the padded and appended message f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): 80 Processing Functions K(0), K(1), ..., K(79): 80 Processing Constant Words

H0, H1, H2, H3, H4, H5: 5 Word buffers with initial values

Step 7: Pseudo Code….

- For loop on k = 1 to L

  (W(0),W(1),...,W(15)) = M[k] */* Divide M[k] into 16 words */* For t = 16 to 79 do:
- W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)) <<< 1 A = H0, B = H1, C = H2, D = H3, E = H4

  For t = 0 to 79 do:
- TEMP = A<<<5 + f(t;B,C,D) + E + W(t) + K(t) E = D, D = C,

  C = B<<<30, B = A, A = TEMP End of for loop

H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E End of for loop

Step 8: Output:

H0, H1, H2, H3, H4, H5: Word buffers with final message digest

IMPLEMENTATION OF SHA-1

PROGRAM

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;
public class Sha1 {
public static void main(String[] args) throws NoSuchAlgorithmException
{
Scanner s=new Scanner(System.in);
System.out.println("\nEnter the Input String: ");
String str=s.nextLine();
System.out.println("\nMessage Digest(40 bits long): "+sha1(str));
}
static String sha1(String input) throws NoSuchAlgorithmException
{
MessageDigest mDigest = MessageDigest.getInstance("SHA1");
byte[] result = mDigest.digest(input.getBytes());
StringBuilder sb = new StringBuilder();
for(int i = 0; i < result.length; i++) {
sb.append(Integer.toString((result[i] & 0xff) + 0x100, 16).substring(1));
}return sb.toString();
}
}
```

OUTPUT:

```
JavaApplication4 (run) ×   Debugger Console ×   sha1 (run) ×

run:

Enter the Input String:
sha1algorithm

Message Digest(40 bits long): b7864943d7227ffe11d32b1b7375faad89745252
BUILD SUCCESSFUL (total time: 16 seconds)
```

RESULT:

Thus the java program to implement SHA – 1 is executed and the output is verified.