RAISONI GROUP
a vision beyond

G. H. Raisoni College of Engineering & Management, Wagholi, Pune – 412 207

Department of Computer Engineering

D-19

Lab Manual (2021-22)

Class: B TECH Computer   Term: I

Cyber Security - (BCOL19402-A)

Faculty Name: Padmavati Sarode

Course Details

Course : Cyber Security Lab (BCOL19402-A)

Class: BTECH	**Division: A &B**
Internal Marks:	**External marks:**
Credits	**Pattern:**

COURSE OUTCOME	
CO1	Learn the basics of security
CO2	Apply the encryption and decryption algorithms.
CO3	Understand the Need of Intrusion Detection Systems and firewall.
CO4	Exemplify the threats posed to information security and the more common attacks associated with those threats.
CO5	Understand Block chain technology

List of Experiments

Sr. No.	Experiment List	CO Mapping	Software Required
1	Client A message is to be transmitted to Client B in secured using network resources Client A will encrypt message and send to Client B then Client B will decrypt message demonstrate the use of a Substitution Ciphers- caesar.	CO1	Eclipse
2	Alice and Bob authorized user exchange message through secure channel using Substitution Ciphers- monoalphabetic.	CO2	Eclipse
3	There are two authorized user wants to communicate in secured channel demonstrate the use of Transposition ciphers techniques	CO2	Eclipse
4	Transfer files between two devices like phone, PC etc. using symmetric key DES algorithm	CO2	Eclipse
5	Client A and Client B exchange message using network resources using symmetric key AES algorithm	CO4 ,C03	Eclipse
6	Two internet users Alice and Bob wish to have secure communication by exchanging key using Diffie -hellman key exchange	CO2 ,CO3	Eclipse
7	Client 1 wants to send a message to client 2, the public keys of the client2 are retrieved from the Whatsapp server, and this used to encrypt the message and send it to the client2. Client2 then decrypts the message with his own private key. Once a session has been established, clients exchange messages that are protected with a RSA algorithm	CO2 ,CO3	Eclipse
8	End-to-end encryption makes sure that a message that is sent is received only by the intended recipient and none other demonstrate use of MD5.	CO2, CO3	Eclipse
9	A message is to be transmitted using network resources from one machine to another calculate and demonstrate the use of a Hash value equivalent to SHA-1.	CO2,CO3	Eclipse
	Content Beyond Syllabus		
10	Write a program for advanced DES Algorithm	CO3	Eclipse

Assignment No 1

Problem Statement:

Client A message is to be transmitted to Client B in secured using network resources Client A will encrypt message and send to Client B then Client B will decrypt message demonstrate the use of a Substitution Ciphers- Caesar.

Theory:

Caesar Cipher

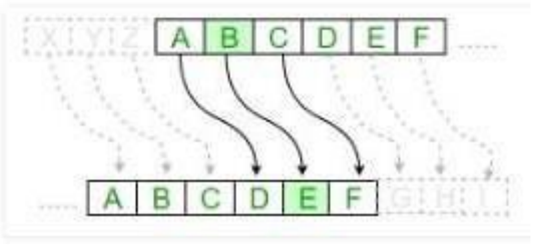
The Caesar Cipher technique is one of the earliest and simplest method of encryption technique. It‟s simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter some fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.
Thus to cipher a given text we need an integer value, known as shift which indicates the number of position each letter of the text has been moved down.
The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as.

$$E_n(x) = (x + n) mod\ 26$$

(Encryption Phase with shift n)

$$D_n(x) = (x - n) mod\ 26$$

(Decryption Phase with shift n)



Input:
1. A String of lower case letters, called Text.
2. An Integer between 0-25 denoting the required shift.

Procedure:
- Traverse the given text one character at a time.
- For each character, transform the given character as per the rule, depending on whether we‟re encrypting or decrypting the text.
- Return the new string generated.

- **How to decrypt?**
  We can either write another function decrypt similar to encrypt, that‟ll apply the given shift in the opposite direction to decrypt the original text. However we can use the cyclic property of the cipher under modulo , hence we can simply observe
- Cipher(n) = De-cipher(26-n)
- Hence, we can use the same function to decrypt, instead we‟ll modify the shift value such that shift = 26-shift.

Code:

Encryption

```
  import java.util.Scanner;

   public class CaesarCipherJava {

  public static void main(String...s){

   String message, encryptedMessage = ""; int

key;

         char ch;

         Scanner sc = new Scanner(System.in);

         System.out.println("Enter a message: ");

         message = sc.nextLine();

         System.out.println("Enter key: ");

         key = sc.nextInt();

         for(int i = 0; i < message.length();

                 ++i){ ch = message.charAt(i);

                 if(ch >= 'a' && ch <=

                 'z'){ ch = (char)(ch + key);

                 if(ch > 'z'){

           ch = (char)(ch - 'z' + 'a' - 1);

         }
```

```
                                encryptedMessage += ch;

                  }

                  else if(ch >= 'A' && ch <=

                      'Z'){ ch = (char)(ch + key);


                      if(ch > 'Z'){

                          ch = (char)(ch - 'Z' + 'A' - 1);

                      }

                                 encryptedMessage += ch;

                  }

                  else {

                      encryptedMessage += ch;

                  }

                      }

                      System.out.println("Encrypted Message = " + encryptedMessage);

          }

}
```

Output

```
Enter a message:
Azk nMQ ls
Enter key:
3
Encrypted Message = Dcn qPT ov
```

Decryption

```
import java.util.Scanner;

public class CaesarCipherJava {
```

```
                public static void main(String...s){
                String message, decryptedMessage = "";
                int key;
                char ch;
                Scanner sc = new Scanner(System.in);
                System.out.println("Enter a message: ");
                message = sc.nextLine();
                System.out.println("Enter key: ");
                key = sc.nextInt();
                for(int i = 0; i < message.length();
                        ++i){ ch = message.charAt(i);
                        if(ch >= 'a' && ch <= 'z'){
                ch = (char)(ch - key);
                if(ch < 'a'){
                ch = (char)(ch + 'z' - 'a' + 1);
                }
                decryptedMessage += ch;
        }
        else if(ch >= 'A' && ch <=
                'Z'){ ch = (char)(ch -
                key);

                if(ch < 'A'){
                ch = (char)(ch + 'Z' - 'A' + 1);
                }
                decryptedMessage += ch;
        }
        else {
                decryptedMessage += ch;
        }
                }
                System.out.println("Decrypted Message = " + decryptedMessage);
        }
}
```

Output

```
Enter a message:
abz gpQ
Enter key:
2
Decrypted Message = yzx enO
```

Conclusion:

This way we have implemented, Caesar Cipher program that receives a Text and Shift value and returns the encrypted text.

Assignment No 2

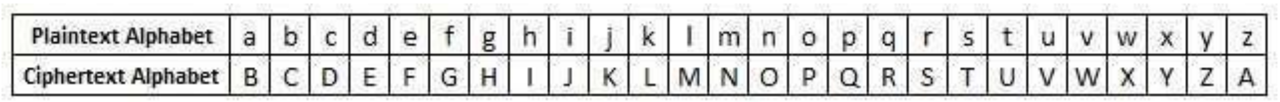
Problem Statement:

Alice and Bob authorized user exchange message through secure channel using Substitution Ciphers- monoalphabetic

Theory:

A monoalphabetic substitution cipher, also known as a simple substitution cipher, relies on a fixed replacement structure. That is, the substitution is fixed for each letter of the alphabet. Thus, if "a" is encrypted to "R", then every time we see the letter "a" in the plaintext, we replace it with the letter "R" in the ciphertext.

A simple example is where each letter is encrypted as the next letter in the alphabet: "a simple message" becomes "B TJNQMF NFTTBHF". In general, when performing a simple substitution manually, it is easiest to generate the ciphertext alphabet first, and encrypt by comparing this to the plaintext alphabet. The table below shows how one might choose to, and we will, lay them out for this example.



Plaintext Alphabet	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Ciphertext Alphabet	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A

The ciphertext alphabet for the cipher where you replace each letter by the next letter in the alphabet

There are many different monoalphabetic substitution ciphers, in fact infinitely many, as each letter can be encrypted to any symbol, not just another letter.

Code:

```
import java.util.Scanner;

 public class MonoalphabeticCipher

{

    public static char p[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',

            'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',

            'w', 'x', 'y', 'z' };

    public static char ch[] = { 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',

            'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'Z', 'X', 'C',

            'V', 'B', 'N', 'M' };
```

```
public static String doEncryption(String s)
{
    char c[] = new char[(s.length())];
    for (int i = 0; i < s.length(); i++)
    {
        for (int j = 0; j < 26; j++)
        {
            if (p[j] == s.charAt(i))
            {
                c[i] = ch[j];
                break;
            }
        }
    }
    return (new String(c));
}
 public static String doDecryption(String s)
{
    char p1[] = new char[(s.length())];
    for (int i = 0; i < s.length(); i++)
    {
        for (int j = 0; j < 26; j++)
        {
            if (ch[j] == s.charAt(i))
            {
                p1[i] = p[j];
```

```
                    break;
                }
            }
        }
        return (new String(p1));
    }
     public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the message: ");
        String en = doEncryption(sc.next().toLowerCase());
        System.out.println("Encrypted message: " + en);
        System.out.println("Decrypted message: " + doDecryption(en));
        sc.close();
    }
}
```

Output:

```
$ javac MonoalphabeticCipher.java
$ java MonoalphabeticCipher
 Enter the message:
Sanfoundry
Encrypted message: LQFYGXFRKN
Decrypted message: sanfoundry
```

Conclusion:

This way we have implemented monoalphabetic ciphers for encryption and decryption.

Assignment No 3

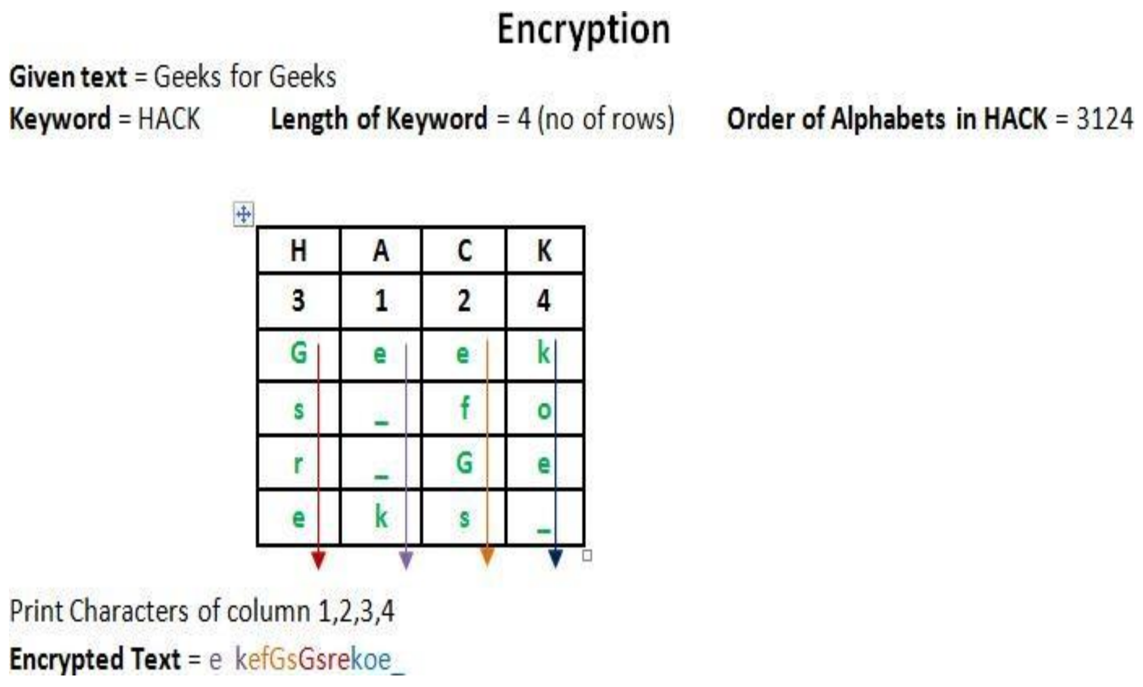
Problem Statement: There are two authorized user wants to communicate in secured channel demonstrate the use of Transposition ciphers techniques

Theory:

Encryption

In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

1. The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.
2. Width of the rows and the permutation of the columns are usually defined by a keyword.
3. For example, the word HACK is of length 4 (so the rows are of length 4), and the permutation is defined by the alphabetical order of the letters in the keyword. In this case, the order would be “3 1 2 4”.
4. Any spare spaces are filled with nulls or left blank or placed by a character (Example: _).
5. Finally, the message is read off in columns, in the order specified by the keyword.



Encryption

Given text = Geeks for Geeks

Keyword = HACK **Length of Keyword** = 4 (no of rows) **Order of Alphabets in HACK** = 3124

H	A	C	K
3	1	2	4
G	e	e	k
s	_	f	o
r	_	G	e
e	k	s	_

Print Characters of column 1,2,3,4

Encrypted Text = e kefGsGsrekoe_

Decryption

1. To decipher it, the recipient has to work out the column lengths by dividing the message length by the key length.
2. Then, write the message out in columns again, then re-order the columns by reforming the key word.

Code:

```
import  java.util.*;
class simpleColumnar{
    public static void main(String
    sap[]){ Scanner sc = new

    System.out.print("\nEnter plaintext(enter in lower case): ");
    String message = sc.next();
    System.out.print("\nEnter key in numbers: ");
    String key = sc.next();

    /* columnCount would keep track of columns */
    int columnCount = key.length();

    /* rowCount will keep of track of rows...no of rows = (plaintextlength + keylength) /
keylength */
    int rowCount = (message.length()+columnCount)/columnCount;

    /*plainText and cipherText would be array containing ASCII values for respective alphabets
*/
    int plainText[][] = new int[rowCount][columnCount];
    int cipherText[][] = new int[rowCount][columnCount];

    /*Encryption Process*/
    System.out.print("\n-----Encryption---- \n");
    cipherText = encrypt(plainText, cipherText, message, rowCount, columnCount, key);

    // prepare final string
    String ct = "";
    for(int i=0; i<columnCount; i++)
    {
        for(int j=0; j<rowCount; j++)
        {
            if(cipherText[j][i] == 0)
                ct = ct + 'x';
            else{
                ct = ct + (char)cipherText[j][i];
            }
        }
    }
    System.out.print("\nCipher Text: " + ct);

    /*Decryption Process*/
    System.out.print("\n\n\n-----Decryption---- \n");

    plainText = decrypt(plainText, cipherText, ct, rowCount, columnCount, key);

    // prepare final string
    String pt = "";
    for(int i=0; i<rowCount; i++)
    {
        for(int j=0; j<columnCount; j++)
        {
            if(plainText[i][j] == 0)
                pt = pt + "";
            else{
                 pt = pt + (char)plainText[i][j];
            }
        }
    }
    System.out.print("\nPlain Text: " + pt);

    System.out.println();
    }

    static int[][] encrypt(int plainText[][], int cipherText[][], String message, int rowCount,
int                                                                         columnCount, String key){
        int i,j;
        int k=0;

        /* here array would be filled row by row  */
        for(i=0; i<rowCount; i++)
        {
            for(j=0; j<columnCount; j++)
            {
                /* terminating condition...as string length can be smaller than 2-D array */
                if(k < message.length())
                {
                    /* respective ASCII characters would be placed */
                    plainText[i][j] = (int)message.charAt(k);
                    k++;
                }
                else
                {
                    break;
                }
            }
        }

        /* here array would be filled according to the key column by column */
        for(i=0; i<columnCount; i++)
        {
            /* currentCol would have current column number i.e. to be read...as there would be
ASCII value stored in key so we would subtract it by 48 so that we can get the original
number...and -1 would be subtract as array position starts from 0*/
```

```
 int currentCol= ( (int)key.charAt(i) - 48 ) -1;
                for(j=0; j<rowCount; j++)
                {
                      cipherText[j][i] = plainText[j][currentCol];
                }

          }

          System.out.print("Cipher Array(read column by column): \n");
          for(i=0;i<rowCount;i++){
                for(j=0;j<columnCount;j++){
                      System.out.print((char)cipherText[i][j]+"\t");
                }
                System.out.println();
          }

          return cipherText;
     }

     static int[][] decrypt(int plainText[][], int cipherText[][], String message, int rowCount,
int                                                                                                              columnCount, String key){
            int i,j;
            int k=0;

            for(i=0; i<columnCount; i++)
            {
                  int currentCol= ( (int)key.charAt(i) - 48 ) -1;
                  for(j=0; j<rowCount; j++)
                  {
                        plainText[j][currentCol] = cipherText[j][i];
                  }
            }

            System.out.print("Plain Array(read row by row): \n");
            for(i=0;i<rowCount;i++){
                 for(j=0;j<columnCount;j++){
                        System.out.print((char)plainText[i][j]+"\t");
                 }
                 System.out.println();
            }

           return plainText;
      }
}
```

Output:

```
Enter plaintext(enter in lower case): networksecurity

Enter key in numbers: 31452

-----Encryption-----
Cipher Array(read column by column):
t	n	w	o	e
s	r	e	c	k
i	u	t	y	r

Cipher Text: tsixnruxwetxocyxekrx


-----Decryption-----
Plain Array(read row by row):
n	e	t	w	o
r	k	s	e	c
u	r	i	t	y

Plain Text: networksecurity
```

Conclusion:

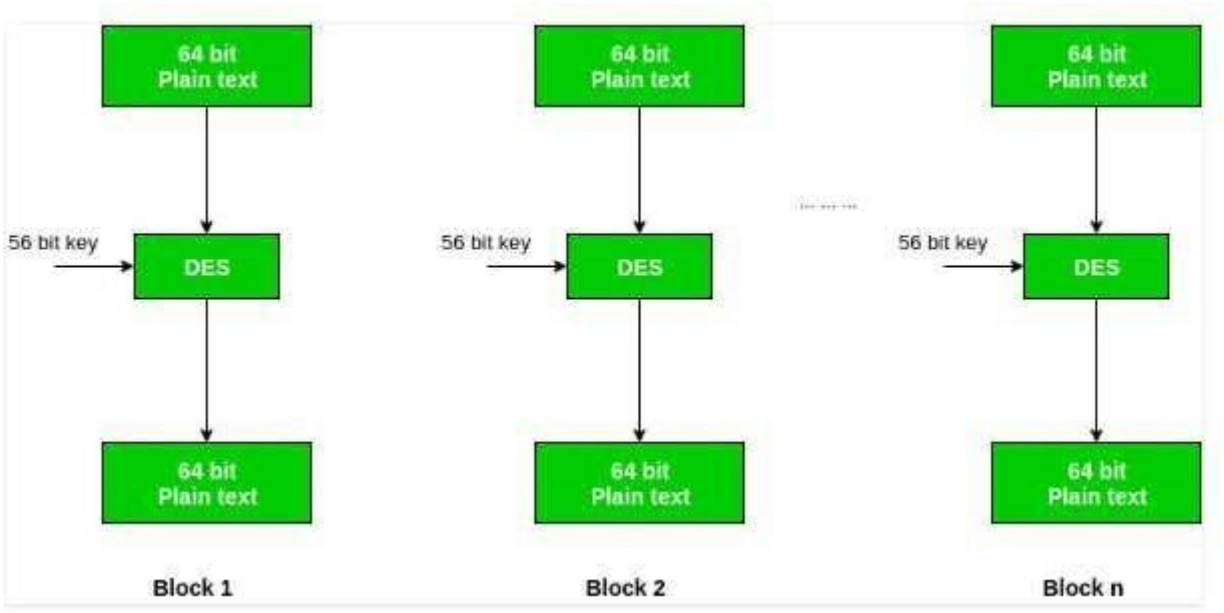
This way we have implemented transposition ciphers for encryption and decryption.

Assignment No 4

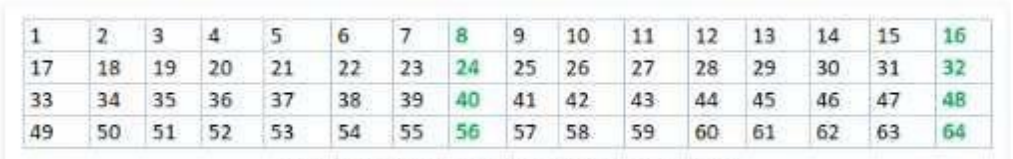
Problem Statement: Transfer files between two devices like phone, PC etc using symmetric key DES algorithm

Theory:

Data encryption standard (DES) has been found vulnerable against very powerful attacks and therefore, the popularity of DES has been found slightly on decline.
DES is a block cipher, and encrypts data in blocks of size of 64 bit each, means 64 bits of plain text goes as the input to DES, which produces 64 bits of cipher text. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is 56 bits. The basic idea is show in figure.



We have mention that DES uses a 56 bit key. Actually, the initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56 bit key. That is bit position 8, 16, 24, 32, 40, 48, 56 and 64 are discarded.



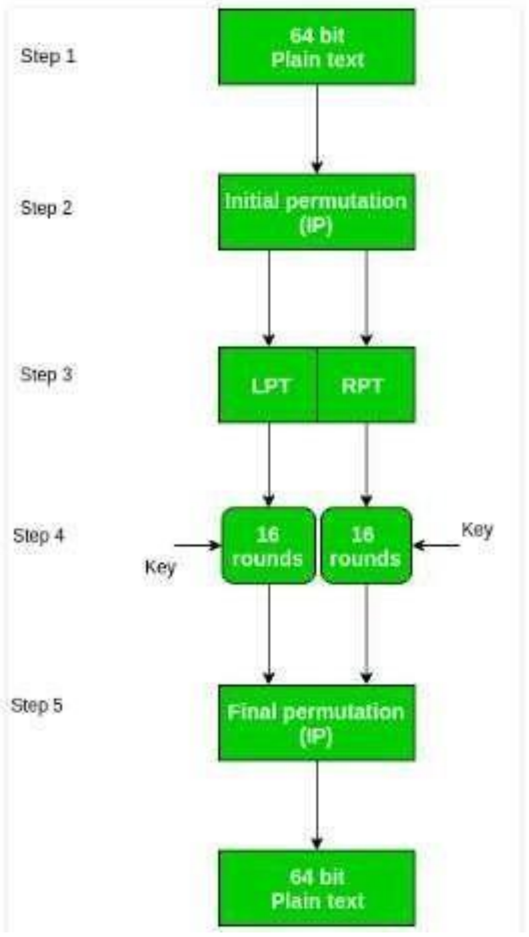
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure - discording of every 8th bit of original key

Thus, the discarding of every 8th bit of the key produces a 56-bit key from the original 64-bit key.

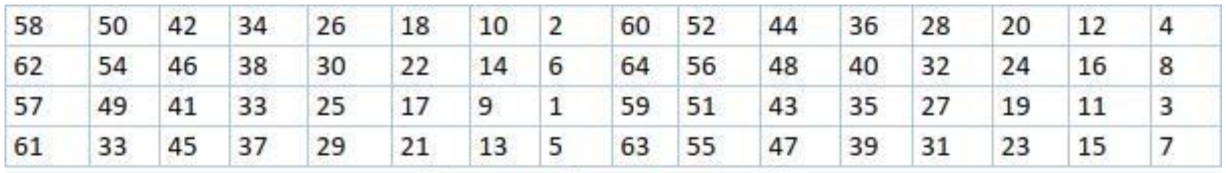
DES is based on the two fundamental attributes of cryptography: substitution (also called as confusion) and transposition (also called as diffusion). DES consists of 16 steps, each of which is called as a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

1. In the first step, the 64 bit plain text block is handed over to an initial Permutation (IP) function.
2. The initial permutation performed on plain text.
3. Next the initial permutation (IP) produces two halves of the permuted block; says Left Plain Text (LPT) and Right Plain Text (RPT).
4. Now each LPT and RPT to go through 16 rounds of encryption process.
5. In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
6. The result of this process produces 64 bit cipher text.



Initial Permutation (IP) –
As we have noted, the Initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as show in figure.
For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block and so on.

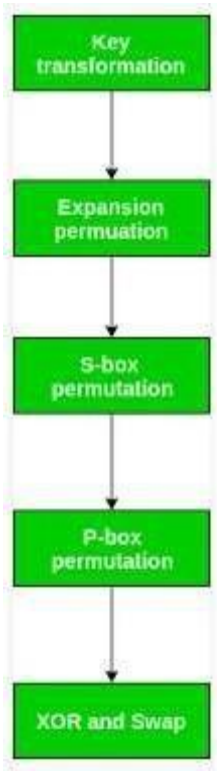
This is nothing but jugglery of bit positions of the original plain text block. the same rule applies for all the other bit positions which shows in the figure.



58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

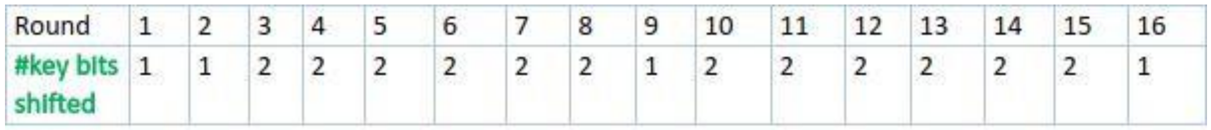
Figure - Initial permutation table

As we have noted after IP done, the resulting 64-bit permuted text block is divided into two half blocks. Each half block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad level steps outlined in figure.



Step-1: Key transformation –
We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called as key transformation. For this the 56 bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.

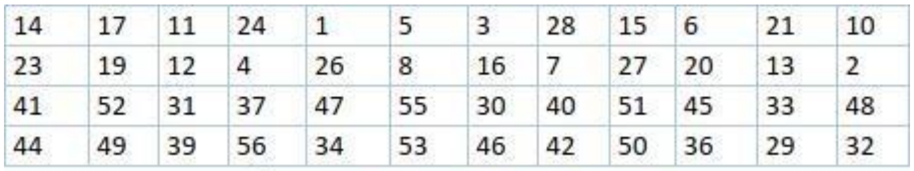
For example, if the round number 1, 2, 9 or 16 the shift is done by only position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is show in figure.



Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure - number of key bits shifted per round

After an appropriate shift, 48 of the 56 bit are selected. for selecting 48 of the 56 bits the table show in figure given below. For instance, after the shift, bit number 14 moves on the first position, bit number 17 moves on the second position and so on. If we observe the table carefully, we will realize that it contains only 48 bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as selection of a 48-bit sub set of the original 56-bit key it is called Compression Permutation.



14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure - compression permutation

Because of this compression permutation technique, a different subset of key bits is used in each round. That‟s make DES not easy to crack.

Step-2: Expansion Permutation –
Recall that after initial permutation, we had two 32-bit plain text areas called as Left Plain Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called as expansion permutation. This happens as the 32 bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4 bit block of the previous step is then expanded to a corresponding 6 bit block, i.e., per 4 bit block, 2 more bits are added.

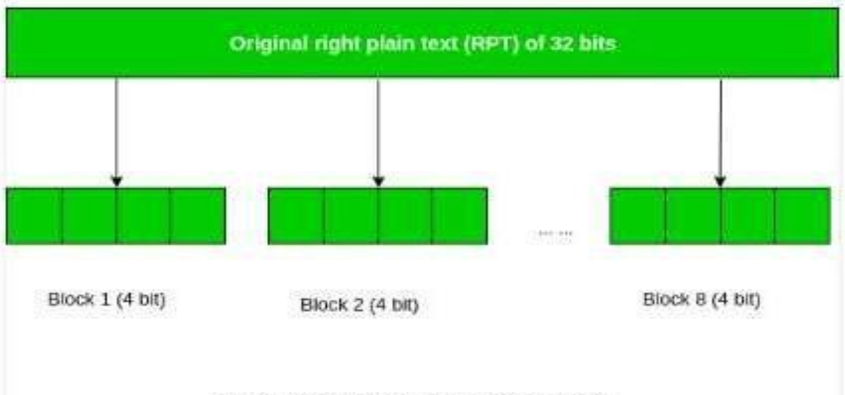


Figure - division of 32 bit RPT into 8 bit blocks

This process results into expansion as well as permutation of the input bit while creating output. Key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the 32-bit RPT to 48-bits. Now the 48-bit key is XOR with 48-bit RPT and resulting output is given to the next step, which is the S-Box substitution.

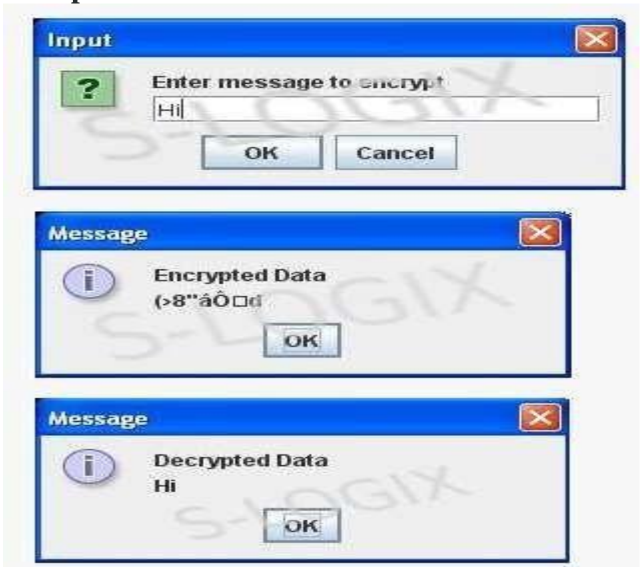
Code:

```
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);
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");
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
Enter message to encrypt
Hi
OK   Cancel

Message
Encrypted Data
(>8"âÔ□d
OK

Message
Decrypted Data
Hi
OK
```

Conclusion:

This way we have implemented DES algorithm.

Assignment No 5

Problem Statement: Client A and Client B exchange message using network resources using symmetric key AES algorithm

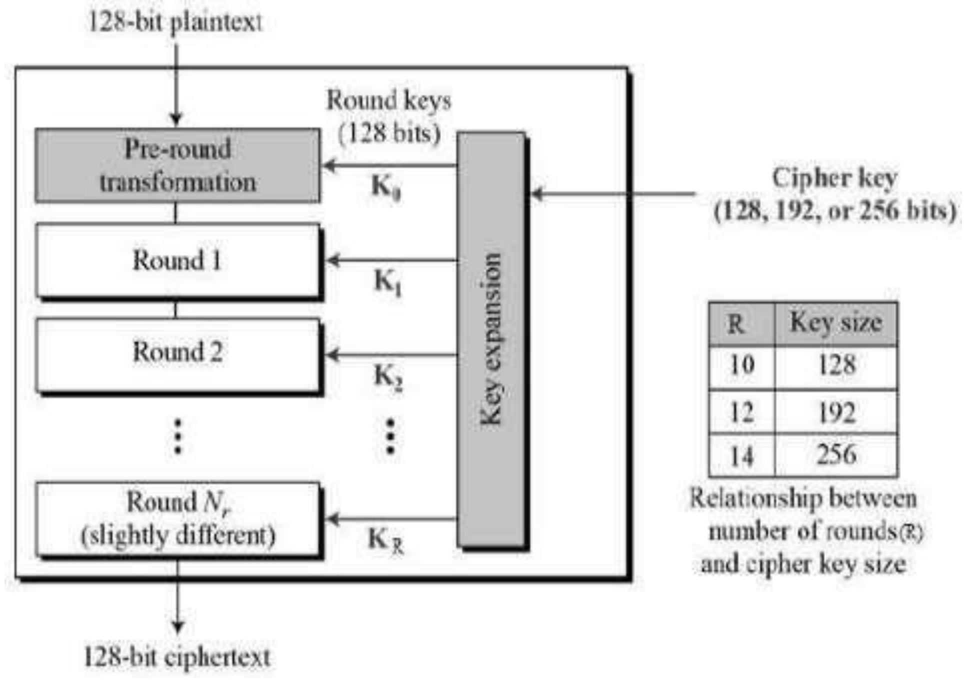
Theory:

AES is an iterative rather than Feistel cipher. It is based on „substitution–permutation network". It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix −

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.

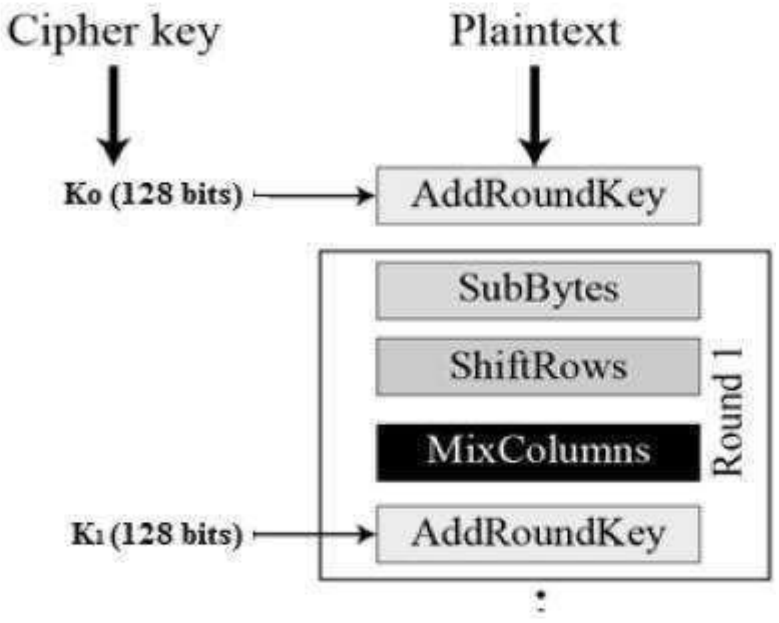
The schematic of AES structure is given in the following illustration



R	Key size
10	128
12	192
14	256

Encryption Process

Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes. The first round process is depicted below −



Byte Substitution (SubBytes)

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

Shiftrows

Each of the four rows of the matrix is shifted to the left. Any entries that „fall off" are re-inserted on the right side of row. Shift is carried out as follows −

- First row is not shifted.
- Second row is shifted one (byte) position to the left.
- Third row is shifted two positions to the left.
- Fourth row is shifted three positions to the left.
- The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

MixColumns

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.

Addroundkey

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

Decryption Process

The process of decryption of an AES ciphertext is similar to the encryption process in the reverse order. Each round consists of the four processes conducted in the reverse order −

- Add round key
- Mix columns
- Shift rows
- Byte substitution

Since sub-processes in each round are in reverse manner, unlike for a Feistel Cipher, the encryption and decryption algorithms needs to be separately implemented, although they are very closely related.

Code:

```
import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.Base64;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class AES {

    private static SecretKeySpec secretKey;
    private static byte[] key;

    public static void setKey(String myKey)
    {
        MessageDigest sha = null;
        try {
            key = myKey.getBytes("UTF-8");
            sha = MessageDigest.getInstance("SHA-1");
            key = sha.digest(key);
            key = Arrays.copyOf(key, 16);
            secretKey = new SecretKeySpec(key, "AES");
        }
        catch (NoSuchAlgorithmException e)
            { e.printStackTrace();
        }
        catch (UnsupportedEncodingException e)
            { e.printStackTrace();
        }
    }

    public static String encrypt(String strToEncrypt, String secret)
    {
        try
        {
            setKey(secret);
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF
-8")));
        }
        catch (Exception e)
        {
            System.out.println("Error while encrypting: " + e.toString());
        }
        return null;
    }

    public static String decrypt(String strToDecrypt, String secret)
    {
        try
        {
            setKey(secret);
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));
        }
        catch (Exception e)
        {
            System.out.println("Error while decrypting: " + e.toString());
        }
        return null;
    }
}
public static void main(String[] args)
{
        final String secretKey = "ssshhhhhhhhhhh!!!!";

        String originalString = "howtodoinjava.com";
        String encryptedString = AES.encrypt(originalString, secretKey) ;
        String decryptedString = AES.decrypt(encryptedString, secretKey) ;

        System.out.println(originalString);
        System.out.println(encryptedString);
        System.out.println(decryptedString);
    }
```

Output:

```
howtodoinjava.com
Tg2Nn7wUZOQ6Xc+1lenkZTQ9ZDf9a2/RBRiqJBCIX6o=
howtodoinjava.com
```

Conclusion:

This way we have implemented AES algorithm.

Assignment No 6

Problem Statement: Two internet users Alice and Bob wish to have secure communication by exchanging key using Diffie hellman key exchange

Theory:

Diffie-Hellman key exchange, also called exponential key exchange, is a method of digital encryption that uses numbers raised to specific powers to produce decryption keys on the basis of components that are never directly transmitted, making the task of a would-be code breaker mathematically overwhelming.

To implement Diffie-Hellman, the two end users Alice and Bob, while communicating over a channel they know to be private, mutually agree on positive whole numbers *p* and *q*, such that *p* is a prime number and *q* is a generator of *p*. The generator *q* is a number that, when raised to positive whole-number powers less than *p*, never produces the same result for any two such whole numbers. The value of *p* may be large but the value of *q* is usually small.

The most serious limitation of Diffie-Hellman in its basic or "pure" form is the lack of authentication. Communications using Diffie-Hellman all by itself are vulnerable to man in the middle attacks. Ideally, Diffie-Hellman should be used in conjunction with a recognized authentication method such as digital signatures to verify the identities of the users over the public communications medium. Diffie-Hellman is well suited for use in data communication but is less often used for data stored or archived over long periods of time.

Steps in the algorithm:

1 Alice and Bob agree on a prime number p and a base g.

2 Alice chooses a secret number a, and sends Bob (g^ a mod p).

3 Bob chooses a secret number b, and sends Alice (g ^b mod p).

4 Alice computes ((g^ b mod p)^ a mod p).

5 Bob computes ((g ^a mod p) ^b mod p).

Both Alice and Bob can use this number as their key. Notice tha t p and g need not be protected.

1 Alice and Bob agree on p = 23 and g = 5.

2 Alice chooses a = 6 and sends 5 6 mod 23 = 8.

3 Bob chooses b = 15 and sends 515 mod 23 = 19.

4 Alice computes 19 6 mod 23 = 2.

5 Bob computes 815 mod 23 = 2.

Then 2 is the shared secret.

Clearly, much larger values of a, b, and p are required. An eavesdropper cannot discover this value even if she knows p and g and can obtain each of the messages.

Code:

```
import java.io.*;
import java.math.BigInteger;
class Diffie
{
    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:

```
Enter prime number:
11
Enter primitive root of 11:7
Enter value for x less than 11:
3
R1=2
Enter value for y less than 11:6
R2=4
Key calculated at Alice's side:9
Key calculated at Bob's side:9
deffie hellman secret key Encryption has Taken
```

Conclusion:

This way we have implemented Diffie hellman key exchange algorithm.

Assignment No 7

Problem Statement: client1 wants to send a message to client 2, the public keys of the client2 are retrieved from the Whatsapp server, and this used to encrypt the message and send it to the client2. Client2 then decrypts the message with his own private key. Once a session has been established, clients exchange messages that are protected with a RSA algorithm

Theory:

RSA algorithm is asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. **Public Key** and **Private Key.** As the name describes that the Public Key is given to everyone and Private key is kept private.

An example of asymmetric cryptography :

1. A client (for example browser) sends its public key to the server and requests for some data.
2. The server encrypts the data using client‟s public key and sends the encrypted data.
3. Client receives this data and decrypts it.

>> **Generating Public Key :**

Select two prime no's. Suppose **P = 53 and Q = 59**.
Now First part of the Public key : **n = P*Q = 3127**.
We also need a small exponent say **e** :
But e Must be
An integer.
Not be a factor of n.
1 < e < Φ(n) [Φ(n) is discussed below],
Let us now consider it to be equal to 3.

Our Public Key is made of n and e
>> **Generating Private Key :**
We need to calculate Φ(n) :
Such that **Φ(n) = (P-1)(Q-1)**
so, Φ(n) = 3016

Now calculate Private Key, **d** :
d = (k*Φ(n) + 1) / e for some integer k
For k = 2, value of d is 2011.
Now we are ready with our – Public Key (n = 3127 and e = 3) and Private Key(d = 2011)

Now we will encrypt **"HI"** :
Convert letters to numbers : H = 8 and I = 9
Thus Encrypted Data c = 89e mod n.
Thus our Encrypted Data comes out to be 1394
Now we will decrypt 1394 :

Decrypted Data = cd mod n.

Thus our Encrypted Data comes out to be 89
8 = H and I = 9 i.e. "HI".

Code:

```
import java.io.*;
import java.math.*;
class rsa
{
public static void main(String args[])throws IOException
{
int q,p,n,pn,publickey=0,d=0,msg;
double cipher,ptext;
int check,check1;
DataInputStream in=new DataInputStream(System.in);
System.out.println(“ENTER NO”);
p=Integer.parseInt(in.readLine());
q=Integer.parseInt(in.readLine());
check=prime(p);
check1=prime(q);
if(check!=1||check1!=1)
{
System.exit(0);
}
n=p*q;
pn=(p-1)*(q-1);
for(int e=2;e<pn;e++)
{
if(gcd(e,pn)==1)
{
publickey=e;
System.out.println(“PUBLIC KEY :”+e);
break;
}
}
for(int i=0;i<pn;i++)
{
d=i;
if(((d*publickey)%pn)==1)
break;
}
System.out.println(“PRIVATE KEY :”+d);
System.out.println(“ENTER MESSAGE “);
msg=Integer.parseInt(in.readLine());
cipher=Math.pow(msg,publickey);
cipher=cipher%n;
System.out.println("ENCRYPTED :"+cipher);
ptext=Math.pow(cipher,d);
ptext=ptext%n;
System.out.println("DECRYPTED :"+ptext);
}
static int prime(int a)
{
int flag=0;
for(int i=2;i<a;i++)
{
if(a%i==0)
{
System.out.println(a+" is not a Prime Number");
flag = 1;
return 0;
}
}
if(flag==0)
return 1;
return 1;
}
static int gcd(int number1, int number2)
{
if(number2 ==
0){ return
number1;
}
return gcd(number2, number1%number2);
}
}
```

Output:

```
ENTER NO
3
11
PUBLIC KEY :3
PRIVATE KEY :7
ENTER MESSAGE
20
ENCRYPTED :14.0
DECRYPTED :20.0
```

Conclusion:

This way we have implemented RSA algorithm.

Assignment No 8

Problem Statement: End-to-end encryption makes sure that a message that is sent is received only by the intended recipient and none other demonstrate use of MD5.

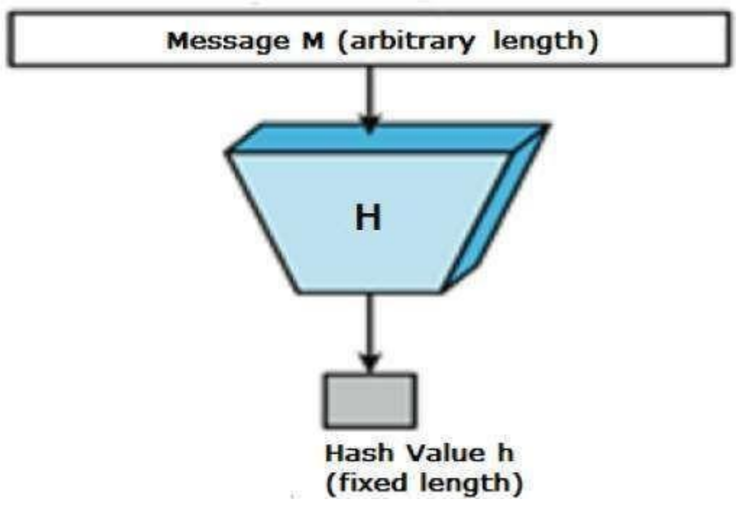
Theory:

The **MD5 message-digest algorithm** is a widely used hash function producing a 128-bit hash value. Although MD5 was initially designed to be used as a cryptographic hash function, it has been found to suffer from extensive vulnerabilities. It can still be used as a checksum to verify data integrity, but only against unintentional corruption. It remains suitable for other non-cryptographic purposes, for example for determining the partition for a particular key in a partitioned database.

Hash functions are extremely useful and appear in almost all information security applications.

A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.

Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function −



Features of Hash Functions

The typical features of hash functions are −

- **Fixed Length Output (Hash Value)**
    - Hash function coverts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.

    - In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.
    - Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.
    - Hash function with n bit output is referred to as an **n-bit hash function**. Popular hash functions generate values between 160 and 512 bits.

- **Efficiency of Operation**
    - Generally for any hash function h with input x, computation of h(x) is a fast operation.
    - Computationally hash functions are much faster than a symmetric encryption.

Message Digest (MD)

MD5 was most popular and widely used hash function for quite some years.

- The MD family comprises of hash functions MD2, MD4, MD5 and MD6. It was adopted as Internet Standard RFC 1321. It is a 128-bit hash function.
- MD5 digests have been widely used in the software world to provide assurance about integrity of transferred file. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.
- In 2004, collisions were found in MD5. An analytical attack was reported to be successful only in an hour by using computer cluster. This collision attack resulted in compromised MD5 and hence it is no longer recommended for use.

Code:

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

// Java program to calculate MD5 hash value
public class MD5 {
    public static String getMd5(String input)
    {
        try {
            // Static getInstance method is called with hashing MD5
            MessageDigest md = MessageDigest.getInstance("MD5");

            // digest() method is called to calculate message digest
            // of an input digest() return array of byte
            byte[] messageDigest = md.digest(input.getBytes());

            // Convert byte array into signum representation
            BigInteger no = new BigInteger(1, messageDigest);

            // Convert message digest into hex value
            String hashtext = no.toString(16);
            while (hashtext.length() < 32)
                { hashtext = "0" + hashtext;
            }
            return hashtext;
        }

        // For specifying wrong message digest algorithms
        catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    // Driver code
    public static void main(String args[]) throws NoSuchAlgorithmException
    {
        String s = "GeeksForGeeks";
        System.out.println("Your HashCode Generated by MD5 is: " + getMd5(s));
    }
}
```

Output:

```
Your HashCode Generated by MD5 is: e39b9c178b2c9be4e99b141d956c6ff6
```

Conclusion:

This way we have implemented MD5 algorithm.

Assignment No 9

Problem Statement: A message is to be transmitted using network resources from one machine to another calculate and demonstrate the use of a Hash value equivalent to SHA-1.

Theory:

SHA-1 or Secure Hash Algorithm 1 is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value. This hash value is known as a message digest. This message digest is usually then rendered as a hexadecimal number which is 40 digits long. It is a U.S. Federal Information Processing Standard and was designed by the United States National Security Agency.

SHA-1 is now considered insecure since 2005. Major tech giants browsers like Microsoft, Google, Apple and Mozilla have stopped accepting SHA-1 SSL certificates by 2017.

To calculate cryptographic hashing value in Java, **MessageDigest Class** is used, under the package **java.security**.
MessagDigest Class provides following cryptographic hash function to find hash value of a text as follows:

- MD2
- MD5
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

These algorithms are initialized in static method called **getInstance()**. After selecting the algorithm the message digest value is calculated and the results are returned as a byte array. BigInteger class is used, to convert the resultant byte array into its signum representation. This representation is then converted into a hexadecimal format to get the expected MessageDigest.

Secure Hash Function (SHA)

Family of SHA comprise of four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from same family, there are structurally different.

- The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0.

- SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security.

- In 2005, a method was found for uncovering collisions for SHA-1 within practical time frame making long-term employability of SHA-1 doubtful.
- SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending up on number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function.
- Though SHA-2 is a strong hash function. Though significantly different, its basic design is still follows design of SHA-1. Hence, NIST called for new competitive hash function designs.
- In October 2012, the NIST chose the Keccak algorithm as the new SHA-3 standard. Keccak offers many benefits, such as efficient performance and good resistance for attacks.

Code:

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class GFG {
    public static String encryptThisString(String input)
    {
        try {
// getInstance() method is called with algorithm SHA-1
MessageDigest md = MessageDigest.getInstance("SHA-1");

// digest() method is called
// to calculate message digest of the input string
// returned as array of byte
byte[] messageDigest = md.digest(input.getBytes());

// Convert byte array into signum representation
BigInteger no = new BigInteger(1, messageDigest);

// Convert message digest into hex value
String hashtext = no.toString(16);

// Add preceding 0s to make it 32 bit
while (hashtext.length() < 32) {
    hashtext = "0" + hashtext;
}

// return the HashText
```

```
            return hashtext;
        }

        // For specifying wrong message
        digest algorithms catch
        (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    // Driver code
    public static void main(String args[]) throws
                                    NoSuchAlgorithmException
    {

        System.out.println("HashCode Generated by SHA-1 for: ");

        String s1 = "GeeksForGeeks";
        System.out.println("\n" + s1 + " : " + encryptThisString(s1));

        String s2 = "hello world";
        System.out.println("\n" + s2 + " : " + encryptThisString(s2));
    }
}
```

Output:

```
HashCode Generated by SHA-1 for:

GeeksForGeeks :

addf120b430021c36c232c99ef8d926aea2acd6b hello

world :

2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

Conclusion:

This way we have implemented SHA-1 algorithm.

Assignment No 10

Problem Statement: Write a program for Advanced DES (3DES) Algorithm.

Theory:

In 2002, AES (Advanced Encryption Standard) replaced the DES encryption algorithm as the accepted standard. Later in 1995, the advanced version of the DES algorithm was introduced that is known as **Triple DES** (3DES or TDES). Officially, it is known as **Triple Data Encryption Algorithm** (TDEA or 3DEA).

TDEA is also a symmetric-key block cipher algorithm that uses the DES cipher algorithm thrice to each data block. Its block size is **64-bits** and key sizes are 168, 112, and 56-bits, respectively for the keys 1, 2, and 3. It also uses the DES equivalent rounds i.e. 48. It means 16 rounds for each key.

It encrypts the data using the first key (k1), decrypts the data by using the second key (k2) and again encrypts the data by using the third key (k3). Another variant of the algorithm uses only two keys k1 and k3. Where both the keys k1 and k3 are the same. It is used still but considered as a legacy algorithm.

Program:

```
int pc1[56] = {
  57,49,41,33,25,17,9,
  1,58,50,42,34,26,18,
  10,2,59,51,43,35,27,
  19,11,3,60,52,44,36,
  63,55,47,39,31,23,15,
  7,62,54,46,38,30,22,
  14,6,61,53,45,37,29,
  21,13,5,28,20,12,4
;


int pc2[48] = {
  14,17,11,24,1,5,
  3,28,15,6,21,10,
  23,19,12,4,26,8,
  16,7,27,20,13,2,
  41,52,31,37,47,55,
  30,40,51,45,33,48,
  44,49,39,56,34,53,
  46,42,50,36,29,32
};
//Java classes that are mandatory to import for encryption and decryption process
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.AlgorithmParameterSpec;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
public class DesProgram
{
//creating an instance of the Cipher class for encryption
private static Cipher encrypt;
//creating an instance of the Cipher class for decryption
private static Cipher decrypt;
//initializing vector
private static final byte[] initialization_vector = { 22, 33, 11, 44, 55, 99, 66, 77 };
//main() method
public static void main(String[] args)
{
```

```
//path of the file that we want to encrypt
String textFile = "C:/Users/Anubhav/Desktop/DemoData.txt";
//path of the encrypted file that we get as output
String encryptedData = "C:/Users/Anubhav/Desktop/encrypteddata.txt";
//path of the decrypted file that we get as output
String decryptedData = "C:/Users/Anubhav/Desktop/decrypteddata.txt";
try
{
//generating keys by using the KeyGenerator class
SecretKey scrtkey = KeyGenerator.getInstance("DES").generateKey();
AlgorithmParameterSpec aps = new IvParameterSpec(initialization_vector);
//setting encryption mode
encrypt = Cipher.getInstance("DES/CBC/PKCS5Padding");
ncrypt.init(Cipher.ENCRYPT_MODE, scrtkey, aps);
setting decryption mode
decrypt = Cipher.getInstance("DES/CBC/PKCS5Padding");
decrypt.init(Cipher.DECRYPT_MODE, scrtkey, aps);
//calling encrypt() method to encrypt the file
encryption(new FileInputStream(textFile), new FileOutputStream(encryptedData));
//calling decrypt() method to decrypt the file
decryption(new FileInputStream(encryptedData), new FileOutputStream(decrypted
Data));
//prints the stetment if the program runs successfully
System.out.println("The encrypted and decrypted files have been created successfull
y.");
}
//catching multiple exceptions by using the | (or) operator in a single catch block
catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException
 | InvalidAlgorithmParameterException | IOException e)
{
//prints the message (if any) related to exceptions
e.printStackTrace();
}
```

```
}
//method for encryption
private static void encryption(InputStream input, OutputStream output)
throws IOException
{
output = new CipherOutputStream(output, encrypt);
//calling the writeBytes() method to write the encrypted bytes to the file
writeBytes(input, output);
}
//method for decryption
private static void decryption(InputStream input, OutputStream output)
throws IOException
{
input = new CipherInputStream(input, decrypt);
//calling the writeBytes() method to write the decrypted bytes to the file
writeBytes(input, output);
}
//method for writting bytes to the files
private static void writeBytes(InputStream input, OutputStream output)
throws IOException
{
byte[] writeBuffer = new byte[512];
int readBytes = 0;
while ((readBytes = input.read(writeBuffer)) >= 0)
{
output.write(writeBuffer, 0, readBytes);
}
//closing the output stream
output.close();
//closing the input stream
input.close();
}
}
```

Conclusion:

This way we have implemented 3DES algorithm