DELHI TECHNOLOGICAL UNIVERSITY

DEPT. OF COMPUTER SCIENCE

INFORMATION AND NETWORK SECURITY LAB (CO - 405)

Submitted To:-

Dr. Aruna Bhatt

(Associate Professor, COE Department)

Submitted By:-

Umang Ahuja

A4 - G3

(2K16/CO/337)

4th Year

Index

S. No.	Name of Practical	Dated	Signature
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Experiment - 1

Aim

To implement Caesar - Cipher encryption and decryption.

Description

The Caesar Cipher, also known as a shift cipher, is one of the oldest and simplest forms of encrypting a message. It is a type of substitution cipher where each letter in the original message is replaced with a letter corresponding to a certain number of letters shifted up or down in the alphabet.

For example with a shift of 3, A would be replaced by D, B would become E, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

$$c = E(x) = (x + k)\, mod 26$$
(Encryption with k shifts)

$$p = D(x) = (x - k)\, mod 26$$
(Decryption with k shifts)

Algorithm

Algorithm for Caesar Cipher:
Input:

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

Procedure:

1. Traverse the given text character by character .
2. For each character, transform the given character as per the rule, depending on encryption or decryption.
3. Return the new string generated.

Code

```
#include <iostream>
#include <cstring>
using namespace std;

string encrypt(string message, int key)
{
   string cipher;

   for(int i=0;i<message.size();i++)
```

```
    {
        if(message[i] == ' ')
            cipher += ' ';

        else if(isupper(message[i]))
            cipher += (message[i] - 'A' + key )%26 + 'A';

        else
            cipher += (message[i] - 'a' + key )%26 + 'a';
    }
    return cipher;
}

string decrypt(string cipher, int key)
{
    string message;

    for(int i=0;i<cipher.size();i++)
    {
        if(cipher[i] == ' ')
            message += ' ';

        else if(isupper(cipher[i]))
            message += ( (cipher[i] - 'A') - key + 26)%26 + 'A';

        else
            message += ( (cipher[i]- 'a') - key + 26)%26 + 'a';
    }
    return message;
}

int main()
{
    string message;
    int key;

    cout << "Enter the message : ";
    getline(cin, message);

    cout << "Enter the key : ";
    cin >> key;

    string cipher = encrypt(message, key);
```

```
    cout << "\nEncrypted Message : " << cipher << endl;

    string plaintext = decrypt(cipher,key);
    cout << "Decrypted Message : " << plaintext << endl;

    return 0;
}
```

Output

```
lab $ ./a.out
Enter the message : Hello World
Enter the key : 4

Encrypted Message : Lipps Asvph
Decrypted Message : Hello World
```

Discussion

To send a message it is necessary for both parties to have the ‘key’ so that sender can encrypt and receiver can decrypt the message. Here key is the number of shifts by which each character is shifted.

One of the shortcoming of Caesar cipher is that it can be broken even to ciphertext-only attack since the number of keys are only 25.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using Caesar cipher which is one of the simplest and earliest method. But due to the simplicity of the method, the message can be broken by means of cipher text attack in which the attacker can find the key used in the algorithm and further decrypt all the incoming messages.

Another cryptanalytic approach is to calculate the frequency distribution of the letters in the cipher text. This consists of counting how many times each letter appears. Natural English text has a very distinct distribution that can be used help crack codes. This means that the letter e is the most common, and appears almost 13% of the time, whereas z appears far less than 1 percent of time. Since application of the Caesar cipher does not change these letter frequencies, it merely shifts them along a bit. A cryptanalyst just has to find the shift that causes the ciphertext frequencies to match up closely with the natural English frequencies, then decrypt the text using that shift. This method can be used to easily break Caesar ciphers by hand.

Experiment - 2

Aim

To implement autokey cipher encryption and decryption.

Description

The Autokey is polyalphabetic substitution cipher that incorporates the message into the key. Key starts with a relatively-short keyword, the primer, and appends the message to it.

More popular autokeys use a tabula recta, a square with 26 copies of the alphabet, the first line starting with 'A', the next line starting with 'B' etc. To encrypt a plaintext, the row with the first letter of the message and the column with the first letter of the key are located. The letter in which the row and the column cross is the ciphertext letter.

Expressed mathematically, the encryption of the message at letter *i*, is equal to the alphabetic value of *i* in the plaintext plus the alphabetic value of the corresponding *i* in the key.

$$E_K(M_i) = (M_i + K_i) \, mod 26$$

Decryption is the same process reversed, subtracting the key instead of adding to arrive back at the original, plaintext value.

$$D_K(C_i) = (C_i - K_i) \, mod 26$$

Algorithm

Input:

1. A String of lower case letters, called text.
2. A key formed by appending a short keyword and the original text message.

Procedure:

1. Traverse the given text character by character .
2. For each character, transform the given character as per the rule, depending on encryption or decryption.
3. Return the new string generated.

Code

```
#include <iostream>
#include <cstring>

using namespace std;
```

```
string generate_key(string key, string message)
{
   for(int i=0;i<key.size();i++)
   {
      key[i] = toupper(key[i]);
   }

   int i=0;
   while(key.size() < message.size())
   {
      key.push_back(toupper(message[i++]));
   }

   return key;
}

string encrypt(string message, string key)
{
   string cipher;

   for(int i=0;i<message.size();i++)
   {
      if(isupper(message[i]))
         cipher += (message[i] - 'A' + key[i] - 'A' )%26 + 'A';

      else
         cipher += (message[i] - 'a' + key[i] - 'A' )%26 + 'a';
   }

   return cipher;
}

string decrypt(string cipher, string key)
{
   string message;

   for(int i=0;i<cipher.size();i++)
   {
      if(isupper(cipher[i]))
         message += (cipher[i] - 'A' - (key[i] - 'A') + 26 )%26 + 'A';

      else
         message += (cipher[i] - 'a' - (key[i] - 'A') + 26 )%26 + 'a';
```

```
    }

    return message;
}

int main()
{
    string message;
    string key;

    cout << "Enter the message : ";
    cin >> message;

    cout << "Enter the key : ";
    cin >> key;

    key = generate_key(key, message);

    string cipher = encrypt(message, key);
    cout << "\nEncrypted Message : " << cipher << endl;

    string plaintext = decrypt(cipher,key);
    cout << "Decrypted Message : " << plaintext << endl;

    return 0;
}
```

Output

```
lab $ ./a.out
Enter the message : Attack
Enter the key : back

Encrypted Message : Btvkcd
Decrypted Message : Attack
```

Discussion

Autokey ciphers are somewhat more secure than polyalphabetic ciphers that use fixed keys since the key does not repeat within a single message.

A key weakness of the system, however, is that the plaintext is part of the key. That means that the key will likely contain common words at various points. The key can be attacked by using a dictionary of common words, bigrams, trigrams etc. and by attempting the decryption of the message by moving that word through the key until potentially-readable text appears.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using Autokey cipher which appends plaintext message onto the given keyword to form the encryption and decryption key.

The idea behind this cipher is to disguise the plaintext letter frequency to interfere with a straightforward application of frequency analysis. For instance, if P is the most frequent letter in a ciphertext whose plaintext is in English, one might suspect that P corresponds to E since E is the most frequently used letter in English. However, by using the Autokey cipher, E can be enciphered as different ciphertext letters at different points in the message, which defeats simple frequency analysis.

Thus it is more secure than Caesar Cipher.

Experiment - 3

Aim

To implement Playfair Cipher encryption and decryption.

Description

The Playfair Cipher is a manual symmetric encryption cipher invented in 1854 by Charles Wheatstone, however its name and popularity came from the endorsement of Lord Playfair.

The Playfair cipher encrypts pairs of letters (digraphs), instead of single letters as is the case with simpler substitution ciphers such as the Caesar Cipher. Frequency analysis is still possible on the Playfair cipher, however it would be against 600 possible pairs of letters instead of 26 different possible letters. For this reason the Playfair cipher is much more secure than older substitution ciphers.

Algorithm

1. Generate the key Square(5×5):

- The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
- The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.

2. Algorithm to encrypt the plain text:

The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.

Rules for Encryption:

i) If the letters appear on the same row of your table, replace them with the letters to their immediate right respectively (wrapping around to the left side of the row if a letter in the original pair was on the right side of the row).

ii) If the letters appear on the same column of your table, replace them with the letters immediately below respectively (wrapping around to the top side of the column if a letter in the original pair was on the bottom side of the column).

iii) If the letters are not on the same row or column, replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair.

Code

```
#include <iostream>
#include <string>
using namespace std;

string msg;
char n[5][5];

char getChar( int a, int b )
{
   return n[ (a + 5) % 5 ][ (b + 5) % 5 ];
}

bool getPos( char l, int &c, int &d )
{
   for( int x = 0; x < 5; x++ )
   for( int y = 0; y < 5; y++ )
   if( n[x][y] == l ) {
      c = x;
      d = y;
      return true;
   }
   return false;
}

void print()
{
   cout << "\n\nSolution:" << endl;
   string::iterator it = msg.begin();
   while( it != msg.end() ) {
      cout << *it;
      it++;
      cout << *it << " ";
      it++;
   }
}

void getText( string t, bool m, bool e ) {
   for( string::iterator it = t.begin(); it != t.end(); it++ ) {
      *it = toupper( *it );
      if( *it < 65 || *it > 90 )
      continue;
```

```
        if( *it == 'J' && m )
        *it = 'I';
        else if( *it == 'Q' && !m )
        continue;
        msg += *it;
    }
    if( e ) {
        string nmsg = ""; size_t len = msg.length();
        for( size_t x = 0; x < len; x += 2 ) {
        nmsg += msg[x];
        if( x + 1 < len ) {
            if( msg[x] == msg[x + 1] ) nmsg += 'X';
            nmsg += msg[x + 1];
        }
        }
        msg = nmsg;
    }
    if( msg.length() & 1 )
    msg += 'X';
}
void createEncoder( string key, bool m )
{
    key += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    string s= "";
    for(int i = 0 ; i < key.size(); i++)
    {
        key[i] = toupper(key[i]);
        if( key[i] < 65 || key[i] > 90 )
        continue;
        if( ( key[i] == 'J' && m ) || ( key[i] == 'Q' && !m ) )
        continue;
        if( s.find( key[i] ) == -1 )
        s += key[i];
    }
    copy( s.begin(), s.end(), &n[0][0] );

    cout << "\nKey table \n";
    for(int i=0;i<5;i++)
    {
        for(int j=0;j<5;j++)
            cout << n[i][j] << ' ';
        cout << endl;
    }
```

```
}

void play( int dir ) {
   int j,k,p,q;
   string nmsg;
   for( string::const_iterator it = msg.begin(); it != msg.end(); it++ ) {
      if( getPos( *it++, j, k ) )
      if( getPos( *it, p, q)) {
      //for same row
      if( j == p ) {
         nmsg += getChar( j, k + dir );
         nmsg += getChar( p, q + dir );
      }
      //for same column
      else if( k == q ) {
         nmsg += getChar( j + dir, k );
         nmsg += getChar( p + dir, q );
      }
      else {
         nmsg += getChar( j, q );
         nmsg += getChar( p, k );
      }
      }
   }
   msg = nmsg;
}

void play( string k, string t, bool m, bool e )
{
   createEncoder( k, m );
   getText( t, m, e );
   if( e )
      play( 1 );
   else
      play( -1 );
   print();
}

int main( )
{
   string k, i, msg;
   bool m, c;
```

```
    cout << "Encrpty or Decypt? ";
    getline( cin, i );

    c = ( i[0] == 'e' || i[0] == 'E' );

    cout << "Enter a key: ";
    getline( cin, k);

    cout << "I <-> J (Y/N): ";
    getline( cin, i );

    m = ( i[0] == 'y' || i[0] == 'Y' );
    cout << "Enter the message: ";
    getline( cin, msg );

    play( k, msg,m, c );
    return 0;
}
```

Output

Encryption

```
lab $ ./a.out
Encrpty or Decypt? Encrypt
Enter a key: playfair example
I <-> J (Y/N): Y
Enter the message: Hide the gold in the tree stump

Key table
P L A Y F
I R E X M
B C D G H
K N O Q S
T U V W Z


Solution:
BM OD ZB XD NA BE KU DM UI XM MO UV IF
```

Decryption

```
lab $ ./a.out
Encrpty or Decypt? Decrypt
Enter a key: playfair example
I <-> J (Y/N): Y
Enter the message: BM OD ZB XD NA BE KU DM UI XM MO UV IF

Key table
P L A Y F
I R E X M
B C D G H
K N O Q S
T U V W Z


Solution:
HI DE TH EG OL DI NT HE TR EX ES TU MP
```

Discussion

The process is fairly simple and quick with this approach which starts with creating a key table. The key table is a 5×5 grid of letters that will act as the key for encrypting your plaintext. Each of the 25 letters must be unique and one letter of the alphabet (usually J) is omitted from the table (as there are 25 spots and 26 letters in the alphabet). And with same key, the same key table is generated every time.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using Playfair cipher which work on diagrams rather than monogram.

Advantages:

1. It is significantly harder to break since the frequency analysis technique used to break simple substitution ciphers is difficult but still can be used on (25*25) = 625 digraphs rather than 25 monographs which is difficult.
2. Frequency analysis thus requires more cipher text to crack the encryption.

Disadvantages:

1. An interesting weakness is the fact that a digraph in the ciphertext (AB) and it's reverse (BA) will have corresponding plaintexts like UR and RU. That can easily be exploited with the aid of frequency analysis, if the language of the plaintext is known.
2. Another disadvantage is that Playfair cipher is a symmetric cipher thus same key is used for both encryption and decryption.

Experiment - 4

Aim

To implement Vigenere Cipher encryption and decryption.

Description

The Vigenere cipher is probably the best-known example of a polyalphabetic cipher, though it is a simplified special case. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the Vigenere table.

The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

Expressed mathematically, the encryption of the message at letter *i*, is equal to the alphabetic value of *i* in the plaintext plus the alphabetic value of the corresponding *i* in the key.

$$E_K(M_i) = (M_i + K_i) \, mod 26$$

Decryption is the same process reversed, subtracting the key instead of adding to arrive back at the original, plaintext value.

$$D_K(C_i) = (C_i - K_i) \, mod 26$$

Algorithm

Input:

1. A String of lower case letters, called text.
2. A keyword to form key by repeating the keyword multiple times.

Procedure:

1. Traverse the given text character by character .
2. For each character, transform the given character as per the rule, depending on encryption or decryption.
3. Return the new string generated.

Code

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
string generate_key(string message, string key)
{
   for(int i=0;i<key.size();i++)
   {
      key[i] = toupper(key[i]);
   }

   int n = key.size();

   for (int i = 0; key.size() < message.size(); i=(i+1)%n)
   {
      key.push_back(key[i]);
   }
   return key;
}

string encrypt(string message, string key)
{
   string cipher;

   for(int i=0;i<message.size();i++)
   {
      if(isupper(message[i]))
         cipher += (message[i] - 'A' + key[i] - 'A' )%26 + 'A';

      else
         cipher += (message[i] - 'a' + key[i] - 'A' )%26 + 'a';
   }

   return cipher;
}

string decrypt(string cipher, string key)
{
   string message;

   for(int i=0;i<cipher.size();i++)
   {
      if(isupper(cipher[i]))
         message += (cipher[i] - 'A' - (key[i] - 'A') + 26 )%26 + 'A';

      else
         message += (cipher[i] - 'a' - (key[i] - 'A') + 26 )%26 + 'a';
```

```
    }

    return message;
}

int main()
{
    string message;
    string key;

    cout << "Enter the message : ";
    cin >> message;

    cout << "Enter the key : ";
    cin >> key;

    key = generate_key(message, key);

    string cipher = encrypt(message, key);
    cout << "\nEncrypted Message : " << cipher << endl;

    string plaintext = decrypt(cipher,key);
    cout << "Decrypted Message : " << plaintext << endl;

    return 0;
}
```

Output

```
lab $ ./a.out
Enter the message : Information
Enter the key : abc

Encrypted Message : Iohosoaukoo
Decrypted Message : Information
```

Discussion

The primary weakness of the Vigenère cipher is the repeating nature of its key. If a cryptanalyst correctly guesses the key's length, the cipher text can be treated as interwoven Caesar ciphers, which can easily be broken individually.

Here a given letter of the alphabet will not always enciphered by the same ciphertext letter, and, as a consequence, cannot be described by a single set of ciphertext alphabet corresponding to a single set of plaintext alphabet.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using Vigenere cipher which repeats the given keyword to form the encryption and decryption key.

The idea behind this cipher is to disguise the plaintext letter frequency to interfere with a straightforward application of frequency analysis. For instance, if P is the most frequent letter in a ciphertext whose plaintext is in English, one might suspect that P corresponds to E since E is the most frequently used letter in English. However, by using the Vigenere cipher, E can be enciphered as different ciphertext letters at different points in the message, which defeats simple frequency analysis.

Thus it is more secure than Monoalphabetic Ciphers.

Experiment - 5

Aim

To implement Hill cipher encryption and decryption.

Description

The Hill cipher is a polygraphic substitution cipher based on linear algebra. Invented by Lester S. Hill in 1929, it was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once.

Hill cipher is a polygraphic substitution cipher based on linear algebra. Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher.

To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26.

$$E(M) = (KM) \, mod 26$$

To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

$$D(C) = (K^{-1}C) \, mod 26$$

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

Algorithm

We will explain the algorithm with the help of an example.

Encryption:
We have to encrypt the message ‘ACT’ (n=3).The key is ‘GYBNQKURP’ which can be written as the nxn matrix:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}$$

The message ‘ACT’ is written as vector:

$$\begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix}$$

The enciphered vector is given as:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} = \begin{bmatrix} 67 \\ 222 \\ 319 \end{bmatrix} \equiv \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \pmod{26}$$

which corresponds to ciphertext of ‘POH’

Decryption:
To decrypt the message, we turn the ciphertext back into a vector, then simply multiply by the inverse matrix of the key matrix (IFKVIVVMI in letters).The inverse of the matrix used in the previous example is:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}^{-1} \equiv \begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \pmod{26}$$

For the previous Ciphertext ‘POH’:

$$\begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \equiv \begin{bmatrix} 260 \\ 574 \\ 539 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} \pmod{26}$$

which gives us back ‘ACT’.

Code

```
#include<iostream>
#include<cmath>
using namespace std;
```

```
float message[3][1], encrypt[3][1], decrypt[3][1], a[3][3], inv[3][3], c[3][3];

void getKeyMessage() {
   int i, j;
   char msg[3];
   char key_char;
   cout<<"Enter 3x3 invertible matrix for key (uppercase) : ";
   for(i = 0; i < 3; i++)
      for(j = 0; j < 3; j++) {

         cin>>key_char;
         a[i][j] = key_char-65;
         c[i][j] = a[i][j];
      }
   cout<<"\nEnter a 3 lowercase letter string : ";
   cin>>msg;
   for(i = 0; i < 3; i++)
      message[i][0] = msg[i] - 97;
}

void inverse() {
   int i, j, k;
   float p, q;
   for(i = 0; i < 3; i++)
      for(j = 0; j < 3; j++) {
         if(i == j)
            inv[i][j]=1;
         else
            inv[i][j]=0;
      }
   for(k = 0; k < 3; k++) {
      for(i = 0; i < 3; i++) {
         p = c[i][k];
         q = c[k][k];
         for(j = 0; j < 3; j++) {
            if(i != k) {
               c[i][j] = c[i][j]*q - p*c[k][j];
               inv[i][j] = inv[i][j]*q - p*inv[k][j];
            }
         }
      }
   }
   for(i = 0; i < 3; i++)
```

```
        for(j = 0; j < 3; j++)
            inv[i][j] = inv[i][j] / c[i][i];

    cout<<"\n\nInverse Matrix is:\n";
    for(i = 0; i < 3; i++) {
        for(j = 0; j < 3; j++)
            cout<<inv[i][j]<<" ";
        cout<<"\n";
    }
}

void encryption() {
    int i, j, k;
    for(i = 0; i < 3; i++)
        for(j = 0; j < 1; j++)
            for(k = 0; k < 3; k++)
                encrypt[i][j] = encrypt[i][j] + a[i][k] * message[k][j];
    cout<<"\nEncrypted string is: ";
    for(i = 0; i < 3; i++)
        cout<<(char)(fmod(encrypt[i][0], 26) + 97);
}

void decryption() {
    int i, j, k;
    inverse();
    for(i = 0; i < 3; i++)
        for(j = 0; j < 1; j++)
            for(k = 0; k < 3; k++)
                decrypt[i][j] = decrypt[i][j] + inv[i][k] * encrypt[k][j];
    cout<<"\nDecrypted string is: ";
    for(i = 0; i < 3; i++)
        cout<<(char)(fmod(decrypt[i][0], 26) + 97);
    cout<<"\n";
}

int main() {
    getKeyMessage();
    encryption();
    decryption();
    return 0;
}
```

Output

```
lab $ ./a.out
Enter 3x3 invertible matrix for key (uppercase) : GYBNQKURP

Enter a 3 lowercase letter string : cat

Encrypted string is: fin

Inverse Matrix is:
0.15873 -0.777778 0.507937
0.0113379 0.15873 -0.106576
-0.22449 0.857143 -0.489796

Decrypted string is: cat
```

Discussion

The Hill Cipher uses an area of mathematics called Linear Algebra, and in particular requires the user to have an elementary understanding of matrices. It also make use of Modulo Arithmetic (like the Affine Cipher). Because of this, the cipher has a significantly more mathematical nature than some of the others. However, it is this nature that allows it to act (relatively) easily on larger blocks of letters.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using Hill cipher which uses matrix multiplication to encrypt and decrypt the message.

The basic Hill cipher is vulnerable to a known-plaintext attack because it is completely linear. An opponent who intercepts plaintext/ciphertext character pairs can set up a linear system which can (usually) be easily solved; if it happens that this system is indeterminate, it is only necessary to add a few more plaintext/ciphertext pairs. Calculating this solution by standard linear algebra algorithms then takes very little time.

Experiment - 6

Aim

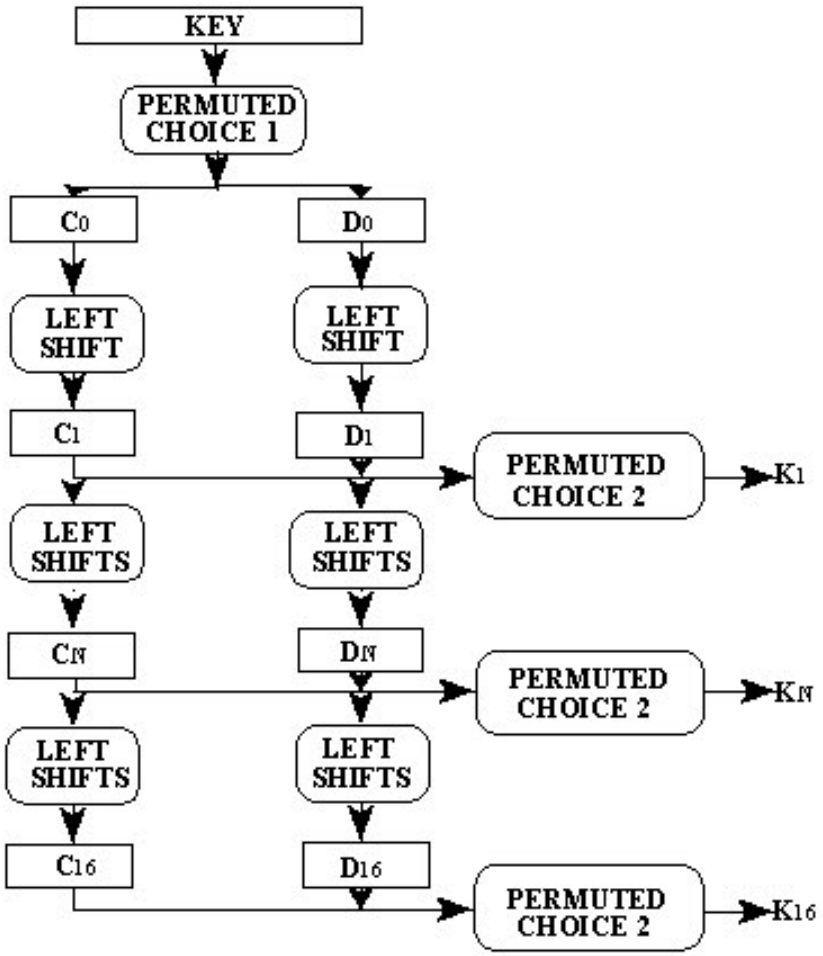
To implement DES subkey generation

Description

The DES (Data Encryption Standard) algorithm is the most widely used encryption algorithm in the world. DES is a *block cipher*--meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. DES operates on the 64-bit blocks using *key* sizes of 56-bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used. A single 64 bit key is used to generate 48 bit round keys by permutation and circular shifts.

Algorithm

1. The 64-bit key is permuted to produce 56 bit subkey using Permuted Choice 1 table.
2. The output is split into two halves C_0 and D_0.
3. For each round from 1 to 16 perform :-
    1. Circular left shift on both individual halves (C_{i-1} and D_{i-1}.) according to round number from shift table to get C_i and D_i.
    2. Combine the result (56 bit) and use Permuted Choice 2 table to get 48 bit round key K_i.



Code

```
#include <iostream>
#include <iomanip>
#include <cstring>

using namespace std;

string permute(string k, int* arr, int n){
   string per="";
   for(int i=0; i<n ; i++)
      per+= k[arr[i]-1];
   return per;
}

string shift_left(string k, int shifts){
   string s="";
   for(int i=0; i<shifts; i++){
      for(int j=1; j<28; j++){
         s+= k[j];
      }
      s+= k[0];
      k= s;
      s="";
   }
   return k;
}

void print_key(string key, int width){
   for(int i=1;i<=key.size();i++)
   {
      cout << key[i-1];
      if(i%width==0) cout << ' ';
   }
   cout << endl;
}

int main(){
   string key;
   cout<<"Enter key(in binary): ";
   cin>>key;

   cout << "\nOriginal Key      : ";
```

```
print_key(key, 8);

int keyp[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
};

key= permute(key, keyp, 56);
cout << "Permuted Choice 1 : ";
print_key(key, 7);

int shift_table[16]=
{ 1, 1, 2, 2,
   2, 2, 2, 2,
   1, 2, 2, 2,
   2, 2, 2, 1
};

int key_comp[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
};

string left= key.substr(0, 28);
string right= key.substr(28, 28);

for(int i=0; i<16; i++){

   left= shift_left(left, shift_table[i]);
   right= shift_left(right, shift_table[i]);
```

```
        string combine= left + right;

        string RoundKey= permute(combine, key_comp, 48);
        cout << "Round " << setw(2) << i+1 << " Key      : ";
        print_key(RoundKey, 6);
    }
    return 0;
}
```

Output

```
lab $ ./a.out
Enter key(in binary): 0001001100110100010101110111100110011011101111001101111111110001

Original Key      : 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001
Permuted Choice 1 : 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111
Round  1 Key      : 000110 110000 001011 101111 111111 000111 000001 110010
Round  2 Key      : 011110 011010 111011 011001 110110 111100 100111 100101
Round  3 Key      : 010101 011111 110010 001010 010000 101100 111110 011001
Round  4 Key      : 011100 101010 110111 010110 110110 110011 010100 011101
Round  5 Key      : 011111 001110 110000 000111 111010 110101 001110 101000
Round  6 Key      : 011000 111010 010100 111110 010100 000111 101100 101111
Round  7 Key      : 111011 001000 010010 110111 111101 100001 100010 111100
Round  8 Key      : 111101 111000 101000 111010 110000 010011 101111 111011
Round  9 Key      : 111000 001101 101111 101011 111011 011110 011110 000001
Round 10 Key      : 101100 011111 001101 000111 101110 100100 011001 001111
Round 11 Key      : 001000 010101 111111 010011 110111 101101 001110 000110
Round 12 Key      : 011101 010111 000111 110101 100101 000110 011111 101001
Round 13 Key      : 100101 111100 010111 010001 111110 101011 101001 000001
Round 14 Key      : 010111 110100 001110 110111 111100 101110 011100 111010
Round 15 Key      : 101111 111001 000110 001101 001111 010011 111100 001010
Round 16 Key      : 110010 110011 110110 001011 000011 100001 011111 110101
```

Discussion

The DES algorithm turns a 64-bit message block M into a 64-bit cipher block C. If each 64-bit block is encrypted individually, then the mode of encryption is called Electronic Code Book (ECB) mode. There are two other modes of DES encryption, namely Chain Block Coding (CBC) and Cipher Feedback (CFB), which make each cipher block dependent on all the previous messages blocks through an initial XOR operation.

Finding and Learnings

In this experiment we learnt how to generate subkeys for each round of DES. A single 64 bit key is passed through 2 permutations to generate 48 bit round keys which are used to encrypt and decrypt the message. Encryption is performed using each subkeys for respective rounds while decryption is performed in the same way but reversing the order of subkeys.

With advent of faster computing, DES with 56 bit key is prone to brute force attack. Thus Double and Triple DES were introduced to increased the key size twice and thrice which make brute force attack infeasible.

Experiment - 7

Aim

To implement Diffie-Hellman key exchange program

Description

The Diffie-Hellmann key exchange is a secure method for exchanging cryptographic keys. The key exchange was invented by Whitfield Diffie and Martin Hellmann in 1976 as the first practical method for establishing a shared secret code over an open communications channel. The general idea of the Diffie-Hellmann key exchange involves two parties exchanging numbers and doing simple calculations in order to get a common number which serves as the secret key.

Algorithm

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo p, where p is prime, and g is a primitive root modulo p. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to p–1.

1. The two users Alice and Bob mutually agree on using prime number p and generator g (g is primitive root of p).
2. Alice chooses a secret key **a** and generate public key $\mathbf{A = g^a \bmod p}$
3. Bob chooses a secret key **b** and generate public key $\mathbf{B = g^b \bmod p}$
4. Both share the public keys over communication channel
5. Alice produce secret key $\mathbf{K_a = B^a \bmod p}$
6. Bob produce secret key $\mathbf{K_b = A^b \bmod p}$

Code

```
#include <iostream>
#include <cmath>

using namespace std;

long long power(long long x, long long p, long long n)
{
   if(p==1) return x;

   long long temp = power(x,p/2,n);

   if(p&1)
      return (temp%n * temp%n * x)%n;
   else
```

```
        return (temp%n * temp%n)%n;
}

int main()
{
    long long p,g;

    cout << "Enter prime number p : ";
    cin >> p;

    cout << "Enter the value of primitive root (g) of p : ";
    cin >> g;

    long long a,b,A,B;

    cout << "\nEnter private key of A : ";
    cin >> a;

    cout << "Enter private key of B : ";
    cin >> b;

    A = power(g,a,p);

    cout << "\nPublic key of A : " << A << endl;

    B = power(g,b,p);

    cout << "Public key of B : " << B << endl;

    long long ka, kb;

    ka = power(B,a,p);
    kb = power(A,b,p);

    cout << "\nSecret key of A : " << ka << endl;
    cout << "Secret key of B : " << kb << endl;

    return 0;
}
```

Output

```
Enter prime number p : 11
Enter the value of primitive root (g) of p : 2

Enter private key of A : 8
Enter private key of B : 4

Public key of A : 3
Public key of B : 5

Secret key of A : 4
Secret key of B : 4
```

Discussion

Diffie-Hellman key exchange is considered secure, despite the fact that attackers can intercept the values p, g, A, and B. On a mathematical level, the Diffie-Hellman key exchange relies on one-way functions as the basis for its security. These are calculations which are simple to do one way, but much more difficult to calculate in reverse.

Finding and Learnings

In this experiment we learnt how to exchange secret key using Diffie-Hellman key exchange algorithm. It is a secure algorithm which allows two parties to share secret key over insecure channel. 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.

Experiment - 8

Aim

To implement RSA encryption and decryption.

Description

RSA (Rivest–Shamir–Adleman) is an algorithm used by modern computers to encrypt and decrypt messages. It is a public key asymmetric algorithm which uses two different keys, one for encryption and one for decryption. In this algorithm, one key is kept private while the other can be shared with anyone to transmit the messages.

Encryption is performed by using public key which is available to all the users whereas the encrypted message can only be decrypted by the owner of the corresponding private key.

Algorithm

Algorithm for RSA :

1. Choose two different large random prime numbers p and q
2. Calculate $n = pq$
   n is the modulus for the public key and the private keys
3. Calculate the totient: $\phi(n) = (p-1)(q-1)$
4. Choose an integer e such that $1 < e < \phi(n)$, and e is co-prime to $\phi(n)$ i.e. e and $\phi(n)$ share no factors other than 1; gcd(e, $\phi(n)$) = 1
5. e is released as the public key
6. Compute d to satisfy the congruence relation $dk \equiv 1(mod(\phi(n))$
7. d is kept as the private key

Code

```
#include <iostream>
#include <cmath>

using namespace std;

long long gcd(long long a, long long b)
{
   if(b==0) return a;

   return gcd(b, a%b);
}

long long power(long long x, long long p, long long n)
```

```
{
    if(p==1) return x;

    long long temp = power(x,p/2,n);

    if(p&1)
        return (temp%n * temp%n * x)%n;
    else
        return (temp%n * temp%n)%n;
}

int main()
{
    long long p,q;

    long long message;

    cout << "Enter two prime numbers : ";
    cin >> p >> q;

    cout << "Enter the value of message : ";
    cin >> message;

    long long n = p * q;
    long long phi = (p-1) * (q-1);

    long long e = 2;
    while(e < phi)
    {
        if(gcd(e,phi)==1)
            break;

        else
            e++;
    }

    long long k = 1;
    long long d = (1 + k*phi)/e;

    while(d < phi)
    {
        if( (1+k*phi)%e == 0)
        {
```

```
            d = (1+k*phi)/e;
            break;
        }

        else
            k++;
    }

    long long c = power(message, e, n);

    cout << "\nOriginal Message : " << message << endl;

    cout << "\nPublic key (e,n) : (" << e << ", " << n << ")" << endl;

    cout << "Private key (d,n) : (" << d << ", " << n << ")" << endl;

    cout << "\nEncrypted Message : " << c << endl;

    long long m = power(c, d, n);

    cout << "Decrypted Message : " << m << endl;

    return 0;
}
```

Output

```
lab $ ./a.out
Enter two prime numbers : 7 11
Enter the value of message : 5

Original Message : 5

Public key (e,n) : (7, 77)
Private key (d,n) : (43, 77)

Encrypted Message : 47
Decrypted Message : 5
```

Discussion

The algorithm is based on the fact that finding the factors of a large composite number is difficult: when the integers are prime numbers, the problem is called prime factorization. RSA security relies on the computational difficulty of factoring large integers. As computing power increases and more efficient factoring algorithms are discovered, the ability to factor larger and larger numbers also increases.

RSA can be used for more than just encrypting data. Its properties also make it a useful system for confirming that a message has been sent by the entity who claims to have sent it, as well as proving that a message hasn't been altered or tampered with.

Finding and Learnings

In this experiment we learnt how to encrypt and decrypt a message using RSA algorithm which uses mathematical operations to perform encryption and decryption. It is simple in nature but still very strong despite its simplicity. Encryption strength is directly tied to key size, and doubling key length can deliver an exponential increase in strength, although it does impair performance. RSA keys are typically 1024- or 2048-bits long, but experts believe that 1024-bit keys are no longer fully secure against all attacks. One important factor is the size of the key. The larger the number of bits in a key (essentially how long the key is), the more difficult it is to crack through attacks such as brute-forcing and factoring.