LIST OF EXPERIMENTS

S. No	Name of Experiment	Page No.	Course Outcome
1	a)Caesar Cipher b) Playfair Cipher c)Hill Cipher d) Vigenere Cipher e) Rail fence – row & Column Transfromation		
2	Implement the MDS algorithm		
3	Apply DES algorithm for practical application		
4	Apply AES algorithm for practical applications		
5	Implement RSA Algorithm using HTML and JavaScript		
6	Implement the diffie-Hellman Key exchange algorithm for a given problem.		
7	Calculate the message digest of a text using the SHA-1 algorithm		
8	Implement the SIGNATURE SCHEME- Digital Signature-Standard.		
9	Demonstrate instruction detection system(ids) using any tool eg. Snort or any other s/w.		
10	Automated attack and penetration tools Exploring N-Stalker, a Vulnerability assessment Tool		

Experiment 01(a)

AIM: Caeser Cipher

THEORY: The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a 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.

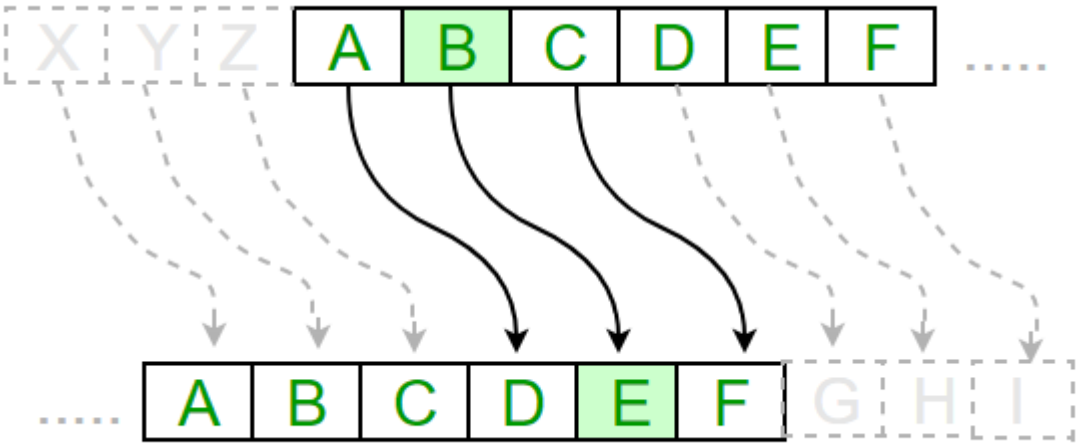
Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down. The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as.

$$En(x)=(x+y) \bmod 26$$

(Encryption Phase with shift n)

$$Dn(x)=(x-n) \bmod 26$$

(Decryption Phase with shift n)



Examples :

Text : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Shift: 23
Cipher: XYZABCDEFGHIJKLMNOPQRSTUVW

Text : ATTACKATONCE
Shift: 4
Cipher: EXXEGOEXSRGI

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:

- 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. A program that receives a Text (string) and Shift value(integer) and returns the encrypted text.

```
// A C++ program to illustrate Caesar Cipher Technique
#include <iostream>
using namespace std;

// This function receives text and shift and
// returns the encrypted text
string encrypt(string text, int s)
{
	string result = "";

	// traverse text
	for (int i=0;i<text.length();i++)
	{
		// apply transformation to each character
		// Encrypt Uppercase letters
		if (isupper(text[i]))
			result += char(int(text[i]+s-65)%26 +65);

	// Encrypt Lowercase letters
	else
		result += char(int(text[i]+s-97)%26 +97);
	}
```

```
        // Return the resulting string
        return result;
}

// Driver program to test the above function
int main()
{
        string text="ATTACKATONCE";
        int s = 4;
        cout << "Text : " << text;
        cout << "\nShift: " << s;
        cout << "\nCipher: " << encrypt(text, s);
        return 0;
}
```

Output
Text : ATTACKATONCE
Shift: 4
Cipher: EXXEGOEXSRGI
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 (Refer to this for a sample run in C++).

Experiment 01(b)

AIM: Playfair Cipher

THEORY: The **Playfair cipher** was the first practical digraph substitution cipher. The scheme was invented in **1854** by **Charles Wheatstone** but was named after Lord Playfair who promoted the use of the cipher. In playfair cipher unlike traditional cipher we encrypt a pair of alphabets(digraphs) instead of a single alphabet.
It was used for tactical purposes by British forces in the Second Boer War and in World War I and for the same purpose by the Australians during World War II. This was because Playfair is reasonably fast to use and requires no special equipment.

Encryption Technique
For the encryption process let us consider the following example:

The Playfair Cipher Encryption Algorithm:
The Algorithm consists of 2 steps:

Generate the key Square(5×5):
1.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.
For example:
PlainText: "instruments"
After Split: 'in' 'st' 'ru' 'me' 'nt' 'sz'
1. Pair cannot be made with same letter. Break the letter in single and add a bogus letter to the previous letter.
Plain Text: “hello”
After Split: ‘he’ ‘lx’ ‘lo’
Here ‘x’ is the bogus letter.
2. If the letter is standing alone in the process of pairing, then add an extra bogus letter with the alone letter
Plain Text: “helloe”

AfterSplit: ‘he’ ‘lx’ ‘lo’ ‘ez’
Here ‘z’ is the bogus letter.

Rules for Encryption:

- **If both the letters are in the same column**: Take the letter below each one (going back to the top if at the bottom).
  For example:

  Diagraph: "me"
  Encrypted Text: cl
  Encryption:
  m -> c
  e -> l

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

- **If both the letters are in the same row**: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).
  For example:

  Diagraph: "st"
  Encrypted Text: tl
  Encryption:
  s -> t
  t -> l

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

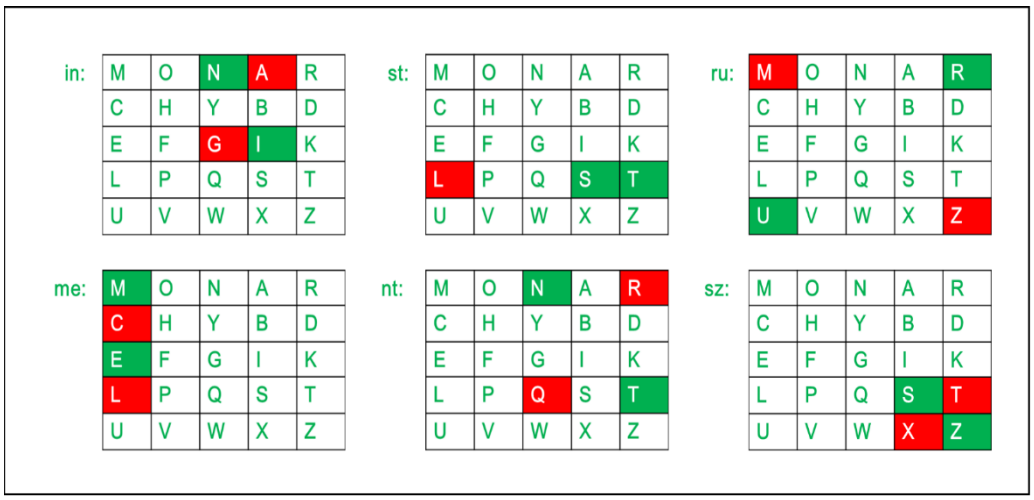
- **If neither of the above rules is true**: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.
  For example:

Diagraph: "nt"
Encrypted Text: rq
Encryption:
n -> r
t -> q

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

For Examples

Plain Text: "instrumentsz"
Encrypted Text: gatlmzclrqtx
Encryption:
i -> g
n -> a
s -> t
t -> l
r -> m
u -> z
m -> c
e -> l
n -> r
t -> q
s -> t
z -> x



Below is an implementation of Playfair Cipher in C:

```
// C++ program to implement Playfair Cipher

#include <bits/stdc++.h>
using namespace std;
#define SIZE 30

// Function to convert the string to lowercase
void toLowerCase(char plain[], int ps)
{
	int i;
	for (i = 0; i < ps; i++) {
		if (plain[i] > 64 && plain[i] < 91)
			plain[i] += 32;
```

```
        }
}

// Function to remove all spaces in a string
int removeSpaces(char* plain, int ps)
{
        int i, count = 0;
        for (i = 0; i < ps; i++)
                if (plain[i] != ' ')
                        plain[count++] = plain[i];
        plain[count] = '\0';
        return count;
}

// Function to generate the 5x5 key square
void generateKeyTable(char key[], int ks, char keyT[5][5])
{
        int i, j, k, flag = 0;

        // a 26 character hashmap
        // to store count of the alphabet
        int dicty[26] = { 0 };
        for (i = 0; i < ks; i++) {
                if (key[i] != 'j')
                        dicty[key[i] - 97] = 2;
        }

        dicty['j' - 97] = 1;

        i = 0;
        j = 0;

        for (k = 0; k < ks; k++) {
                if (dicty[key[k] - 97] == 2) {
                        dicty[key[k] - 97] -= 1;
                        keyT[i][j] = key[k];
                        j++;
                        if (j == 5) {
                                i++;
                                j = 0;
                        }
                }
        }
```

```
        for (k = 0; k < 26; k++) {
                if (dicty[k] == 0) {
                        keyT[i][j] = (char)(k + 97);
                        j++;
                        if (j == 5) {
                                i++;
                                j = 0;
                        }
                }
        }
}

// Function to search for the characters of a digraph
// in the key square and return their position
void search(char keyT[5][5], char a, char b, int arr[])
{
        int i, j;

        if (a == 'j')
                a = 'i';
        else if (b == 'j')
                b = 'i';

        for (i = 0; i < 5; i++) {

                for (j = 0; j < 5; j++) {

                        if (keyT[i][j] == a) {
                                arr[0] = i;
                                arr[1] = j;
                        }
                        else if (keyT[i][j] == b) {
                                arr[2] = i;
                                arr[3] = j;
                        }
                }
        }
}

// Function to find the modulus with 5
int mod5(int a) { return (a % 5); }
```

```
// Function to make the plain text length to be even
int prepare(char str[], int ptrs)
{
	if (ptrs % 2 != 0) {
		str[ptrs++] = 'z';
		str[ptrs] = '\0';
	}
	return ptrs;
}

// Function for performing the encryption
void encrypt(char str[], char keyT[5][5], int ps)
{
	int i, a[4];

	for (i = 0; i < ps; i += 2) {

		search(keyT, str[i], str[i + 1], a);

		if (a[0] == a[2]) {
			str[i] = keyT[a[0]][mod5(a[1] + 1)];
			str[i + 1] = keyT[a[0]][mod5(a[3] + 1)];
		}
		else if (a[1] == a[3]) {
			str[i] = keyT[mod5(a[0] + 1)][a[1]];
			str[i + 1] = keyT[mod5(a[2] + 1)][a[1]];
		}
		else {
			str[i] = keyT[a[0]][a[3]];
			str[i + 1] = keyT[a[2]][a[1]];
		}
	}
}

// Function to encrypt using Playfair Cipher
void encryptByPlayfairCipher(char str[], char key[])
{
	char ps, ks, keyT[5][5];

	// Key
	ks = strlen(key);
	ks = removeSpaces(key, ks);
	toLowerCase(key, ks);
```

```
	// Plaintext
	ps = strlen(str);
	toLowerCase(str, ps);
	ps = removeSpaces(str, ps);

	ps = prepare(str, ps);

	generateKeyTable(key, ks, keyT);

	encrypt(str, keyT, ps);
}

// Driver code
int main()
{
	char str[SIZE], key[SIZE];

	// Key to be encrypted
	strcpy(key, "Monarchy");
	cout << "Key text: " << key << "\n";

	// Plaintext to be encrypted
	strcpy(str, "instruments");
	cout << "Plain text: " << str << "\n";

	// encrypt using Playfair Cipher
	encryptByPlayfairCipher(str, key);

	cout << "Cipher text: " << str << "\n";

	return 0;
}
```

Experiment 01©

AIM: Hill Cipher

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

Examples:

Input : Plaintext: ACT

Key: GYBNQKURP

Output : Ciphertext: POH

Input : Plaintext: GFG

Key: HILLMAGIC

Output : Ciphertext: SWK

Encryption:

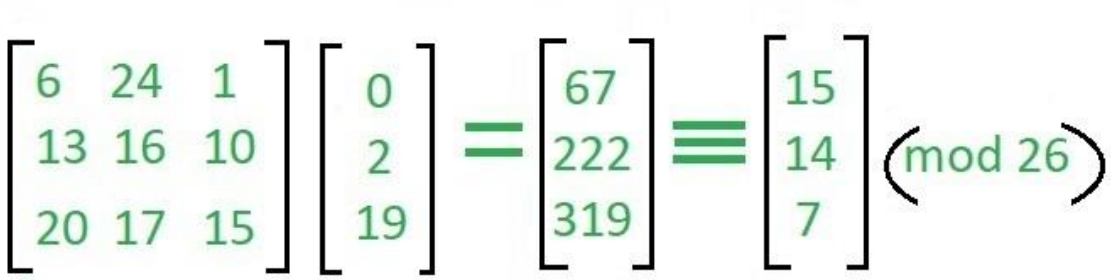
We have to encrypt the message ‘ACT’ (n=3).The key is ‘GYBNQKURP’ which can be written as the nxn matrix:

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

The message ‘ACT’ is written as vector:

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

The enciphered vector is given as:

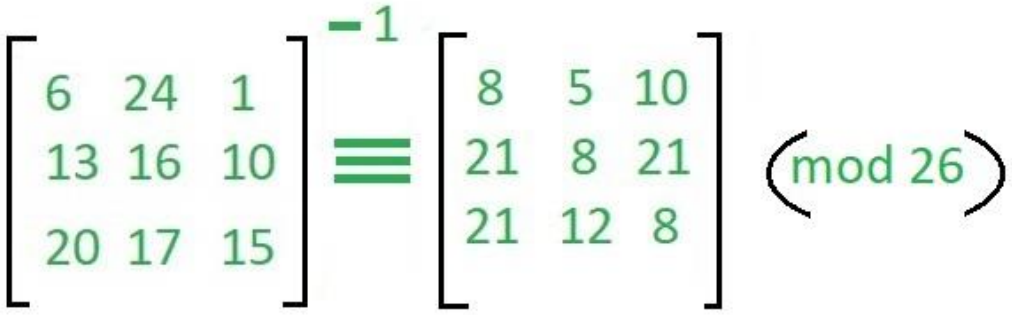


$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} = \begin{bmatrix} 67 \\ 222 \\ 319 \end{bmatrix} \equiv \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \pmod{26}$$

which corresponds to ciphertext of ‘POH’

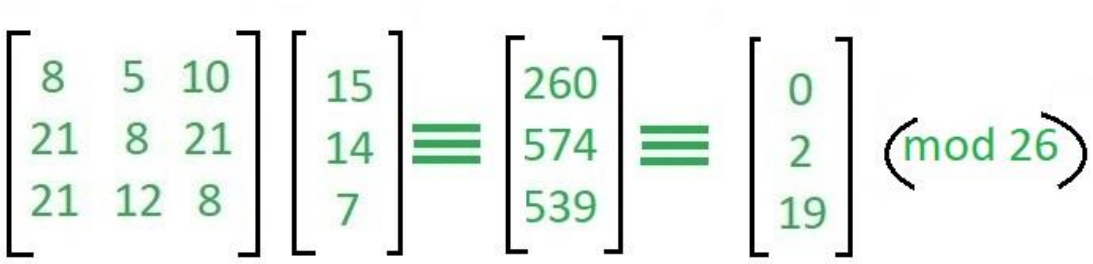
Decryption

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’:



which gives us back ‘ACT’.
Assume that all the alphabets are in upper case.
Below is the implementation of the above idea for n=3.

```
// C++ code to implement Hill Cipher
#include <iostream>
using namespace std;

// Following function generates the
// key matrix for the key string
void getKeyMatrix(string key, int keyMatrix[][3])
{
        int k = 0;
        for (int i = 0; i < 3; i++)
        {
                for (int j = 0; j < 3; j++)
                {
                        keyMatrix[i][j] = (key[k]) % 65;
                        k++;
                }
        }
}

// Following function encrypts the message
void encrypt(int cipherMatrix[][1],
```

```
                int keyMatrix[][3],
                int messageVector[][1])
{
    int x, i, j;
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 1; j++)
        {
            cipherMatrix[i][j] = 0;

            for (x = 0; x < 3; x++)
            {
                cipherMatrix[i][j] +=
                    keyMatrix[i][x] * messageVector[x][j];
            }

            cipherMatrix[i][j] = cipherMatrix[i][j] % 26;
        }
    }
}

// Function to implement Hill Cipher
void HillCipher(string message, string key)
{
    // Get key matrix from the key string
    int keyMatrix[3][3];
    getKeyMatrix(key, keyMatrix);

    int messageVector[3][1];
```

```
        // Generate vector for the message
        for (int i = 0; i < 3; i++)
                messageVector[i][0] = (message[i]) % 65;

        int cipherMatrix[3][1];

        // Following function generates
        // the encrypted vector
        encrypt(cipherMatrix, keyMatrix, messageVector);

        string CipherText;

        // Generate the encrypted text from
        // the encrypted vector
        for (int i = 0; i < 3; i++)
                CipherText += cipherMatrix[i][0] + 65;

        // Finally print the ciphertext
        cout << " Ciphertext:" << CipherText;
}

// Driver function for above code
int main()
{
        // Get the message to be encrypted
        string message = "ACT";

        // Get the key
```

```
    string key = "GYBNQKURP";

    HillCipher(message, key);

    return 0;
}
```

Output:

Ciphertext: POH

In a similar way you can write the code for decrypting the encrypted message by following the steps explained above.

Experiment 01(d)

AIM: Vigennere Cipher

THEORY: Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the *Vigenère square or Vigenère 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.

Example:
Input : Plaintext : GEEKSFORGEEKS

Keyword : AYUSH

Output : Ciphertext : GCYCZFMLYLEIM

For generating key, the given keyword is repeated

in a circular manner until it matches the length of

the plain text.

The keyword "AYUSH" generates the key "AYUSHAYUSHAYU"

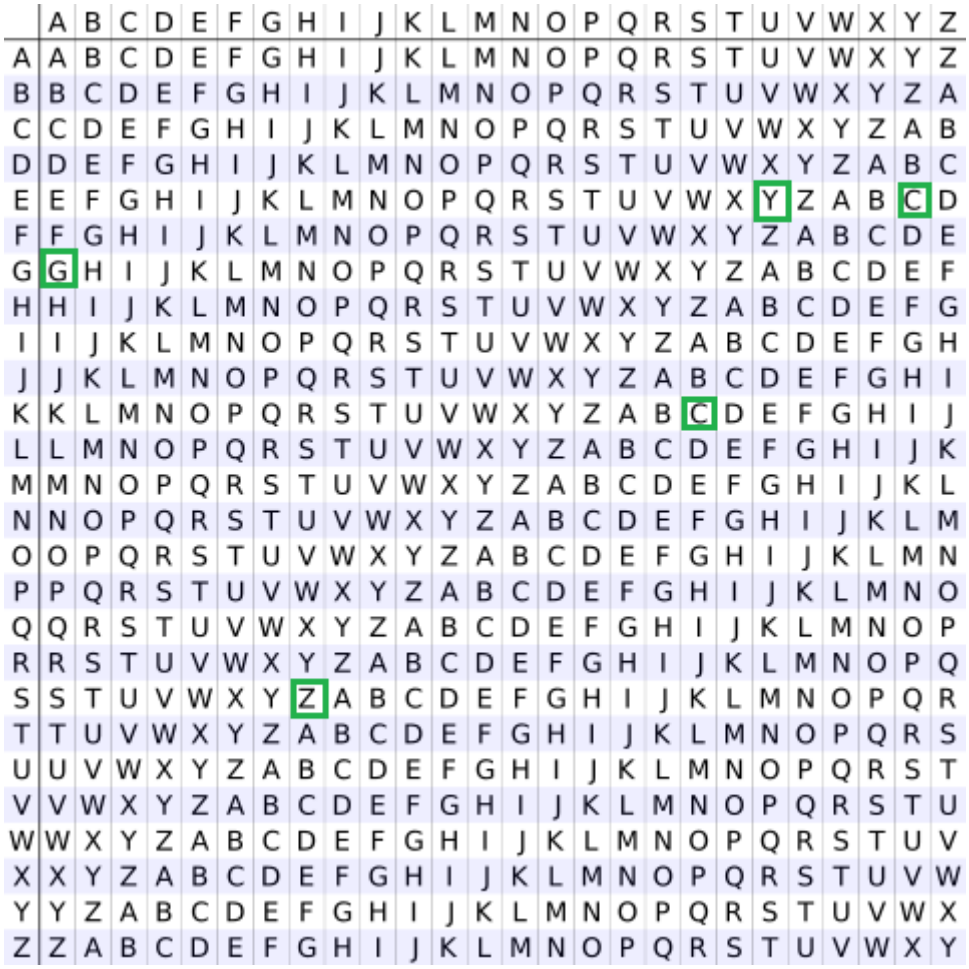
The plain text is then encrypted using the process

explained below.

Encryption:

The first letter of the plaintext, G is paired with A, the first letter of the key. So use row G and column A of the Vigenère square, namely G. Similarly, for the second letter of the plaintext, the second letter of the key is used, the letter at row E, and column Y is C. The rest of the plaintext is enciphered in a similar fashion.

Table to encrypt:



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

Decryption:

Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in this row, and then using the column's label as the plaintext. For example, in row A (from AYUSH), the ciphertext G appears in column G, which is the first plaintext letter. Next, we go to row Y (from AYUSH), locate the ciphertext C which is found in column E, thus E is the second plaintext letter.

A more **easy implementation** could be to visualize Vigenère algebraically by converting [A-Z] into numbers [0–25].

Encryption

The plaintext(P) and key(K) are added modulo 26.

$E_i = (P_i + K_i) \bmod 26$

Decryption

$D_i = (E_i - K_i + 26) \bmod 26$

Note:

D_i denotes the offset of the i-th character of the plaintext. Like offset of **A** is 0 and of **B** is 1 and so on.

```
// C++ code to implement Vigenere Cipher
#include<bits/stdc++.h>
using namespace std;

// This function generates the key in
// a cyclic manner until it's length isi'nt
// equal to the length of original text
string generateKey(string str, string key)
{
  int x = str.size();

  for (int i = 0; ; i++)
  {
        if (x == i)
              i = 0;
        if (key.size() == str.size())
              break;
        key.push_back(key[i]);
  }
  return key;
}

// This function returns the encrypted text
// generated with the help of the key
string cipherText(string str, string key)
{
  string cipher_text;

  for (int i = 0; i < str.size(); i++)
  {
        // converting in range 0-25
        char x = (str[i] + key[i]) %26;

        // convert into alphabets(ASCII)
        x += 'A';

        cipher_text.push_back(x);
  }
  return cipher_text;
```

```
}

// This function decrypts the encrypted text
// and returns the original text
string originalText(string cipher_text, string key)
{
  string orig_text;

  for (int i = 0 ; i < cipher_text.size(); i++)
  {
        // converting in range 0-25
        char x = (cipher_text[i] - key[i] + 26) %26;

        // convert into alphabets(ASCII)
        x += 'A';
        orig_text.push_back(x);
  }
  return orig_text;
}

// Driver program to test the above function
int main()
{
  string str = "GEEKSFORGEEKS";
  string keyword = "AYUSH";

  string key = generateKey(str, keyword);
  string cipher_text = cipherText(str, key);

  cout << "Ciphertext : "
        << cipher_text << "\n";

  cout << "Original/Decrypted Text : "
        << originalText(cipher_text, key);
  return 0;
}
```

Output:

```
Ciphertext : GCYCZFMLYLEIM
```

Experiment 01(e)

AIM: Rail fence-row & column transformation

THEORY: Given a plain-text message and a numeric key, cipher/de-cipher the given text using Rail Fence algorithm. The rail fence cipher (also called a zigzag cipher) is a form of transposition cipher. It derives its name from the way in which it is encoded.

Examples:

Encryption
Input : "GeeksforGeeks "
Key = 3
Output : GsGsekfrek eoe
Decryption
Input : GsGsekfrek eoe
Key = 3
Output : "GeeksforGeeks "

Encryption
Input : "defend the east wall"
Key = 3
Output : dnhaweedtees alf tl
Decryption
Input : dnhaweedtees alf tl
Key = 3
Output : defend the east wall

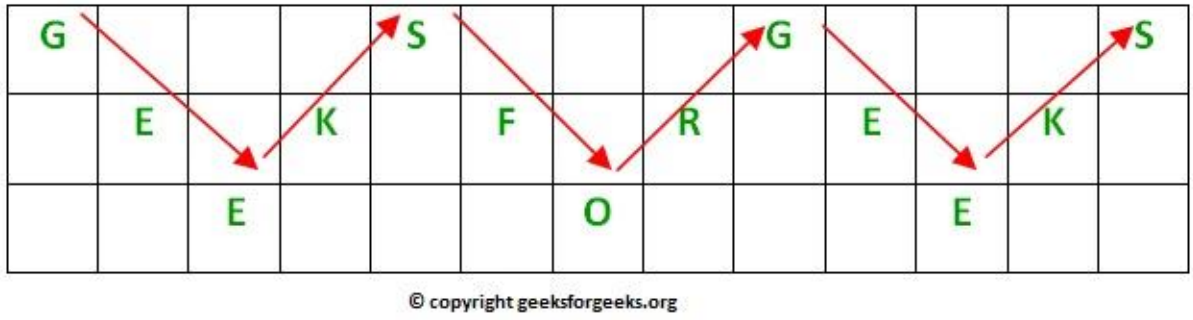
Encryption
Input : "attack at once"
Key = 2
Output : atc toctaka ne
Decryption
Input : "atc toctaka ne"
Key = 2
Output : attack at once

Encryption:

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

- In the rail fence cipher, the plain-text is written downwards and diagonally on successive rails of an imaginary fence.
- When we reach the bottom rail, we traverse upwards moving diagonally, after reaching the top rail, the direction is changed again. Thus the alphabets of the message are written in a zig-zag manner.
- After each alphabet has been written, the individual rows are combined to obtain the cipher-text.

  For example, if the message is “GeeksforGeeks” and the number of rails = 3 then cipher is prepared as:



Decryption:

As we’ve seen earlier, the number of columns in rail fence cipher remains equal to the length of plain-text message. And the key corresponds to the number of rails.

- Hence, rail matrix can be constructed accordingly. Once we’ve got the matrix we can figure-out the spots where texts should be placed (using the same way of moving diagonally up and down alternatively).
- Then, we fill the cipher-text row wise. After filling it, we traverse the matrix in zig-zag manner to obtain the original text.

Implementation:

Let cipher-text = “GsGsekfrek eoe” , and Key = 3

Number of columns in matrix = len(cipher-text) = 13

Number of rows = key = 3

Hence original matrix will be of 3*13 , now marking places with text as ‘*’ we get

```
*   *   *   *
 * * * * * * *
  *   *   *
```

_ _ * _ _ _ * _ _ _ * _

Below is a program to encrypt/decrypt the message using the above algorithm.

```
// C++ program to illustrate Rail Fence Cipher
// Encryption and Decryption
#include <bits/stdc++.h>
using namespace std;

// function to encrypt a message
string encryptRailFence(string text, int key)
{
	// create the matrix to cipher plain text
	// key = rows , length(text) = columns
	char rail[key][(text.length())];

	// filling the rail matrix to distinguish filled
	// spaces from blank ones
	for (int i=0; i < key; i++)
		for (int j = 0; j < text.length(); j++)
			rail[i][j] = '\n';

	// to find the direction
	bool dir_down = false;
	int row = 0, col = 0;

	for (int i=0; i < text.length(); i++)
	{
		// check the direction of flow
		// reverse the direction if we've just
		// filled the top or bottom rail
		if (row == 0 || row == key-1)
			dir_down = !dir_down;

		// fill the corresponding alphabet
		rail[row][col++] = text[i];

		// find the next row using direction flag
		dir_down?row++ : row--;
	}

	//now we can construct the cipher using the rail matrix
```

```
	string result;
	for (int i=0; i < key; i++)
		for (int j=0; j < text.length(); j++)
			if (rail[i][j]!='\n')
				result.push_back(rail[i][j]);

	return result;
}

// This function receives cipher-text and key
// and returns the original text after decryption
string decryptRailFence(string cipher, int key)
{
	// create the matrix to cipher plain text
	// key = rows , length(text) = columns
	char rail[key][cipher.length()];

	// filling the rail matrix to distinguish filled
	// spaces from blank ones
	for (int i=0; i < key; i++)
		for (int j=0; j < cipher.length(); j++)
			rail[i][j] = '\n';

	// to find the direction
	bool dir_down;

	int row = 0, col = 0;

	// mark the places with '*'
	for (int i=0; i < cipher.length(); i++)
	{
		// check the direction of flow
		if (row == 0)
			dir_down = true;
		if (row == key-1)
			dir_down = false;

		// place the marker
		rail[row][col++] = '*';

		// find the next row using direction flag
		dir_down?row++ : row--;
	}
```

```
        // now we can construct the fill the rail matrix
        int index = 0;
        for (int i=0; i<key; i++)
                for (int j=0; j<cipher.length(); j++)
                        if (rail[i][j] == '*' && index<cipher.length())
                                rail[i][j] = cipher[index++];


        // now read the matrix in zig-zag manner to construct
        // the resultant text
        string result;

        row = 0, col = 0;
        for (int i=0; i< cipher.length(); i++)
        {
                // check the direction of flow
                if (row == 0)
                        dir_down = true;
                if (row == key-1)
                        dir_down = false;

                // place the marker
                if (rail[row][col] != '*')
                        result.push_back(rail[row][col++]);

                // find the next row using direction flag
                dir_down?row++: row--;
        }
        return result;
}

//driver program to check the above functions
int main()
{
        cout << encryptRailFence("attack at once", 2) << endl;
        cout << encryptRailFence("GeeksforGeeks ", 3) << endl;
        cout << encryptRailFence("defend the east wall", 3) << endl;

        //Now decryption of the same cipher-text
        cout << decryptRailFence("GsGsekfrek eoe",3) << endl;
        cout << decryptRailFence("atc toctaka ne",2) << endl;
        cout << decryptRailFence("dnhaweedtees alf tl",3) << endl;
```

```
        return 0;
}
```

Output:

```
atc toctaka ne
GsGsekfrek eoe
dnhaweedtees alf  tl
GeeksforGeeks
attack at once
delendfthe east wal
```

Experiment 2

AIM: Implement the MDS algorithm

THEORY: **Multidimensional scaling (MDS)** is a way to reduce the dimensionality of data to visualize it. We basically want to project our (likely highly dimensional) data into a lower dimensional space and preserve the distances between points. The first time that I learned about MDS was by way of STATS 202 with John Taylor, although to be honest, I got through the class by watching old lectures by David Mease on youtube. Thank goodness for youtube! If we have some highly complex data that we project into some lower N dimensions, we will assign each point from our data a coordinate in this lower dimensional space, and the idea is that these N dimensional coordinates are ordered based on their ability to capture variance in the data. Since we can only visualize things in 2D, this is why it is common to assess your MDS based on plotting the first and second dimension of the output. It's not that the remaining N-2 axis don't capture meaningful information, however they capture progressively less information.

If you look at the output of an MDS algorithm, which will be points in 2D or 3D space, the distances represent similarity. So very close points = very similar, and points farther away from one another = less similar. MDS can also be useful for assessing correlation matrices, since a correlation is just another metric of similarity between two things.

How does MDS Work

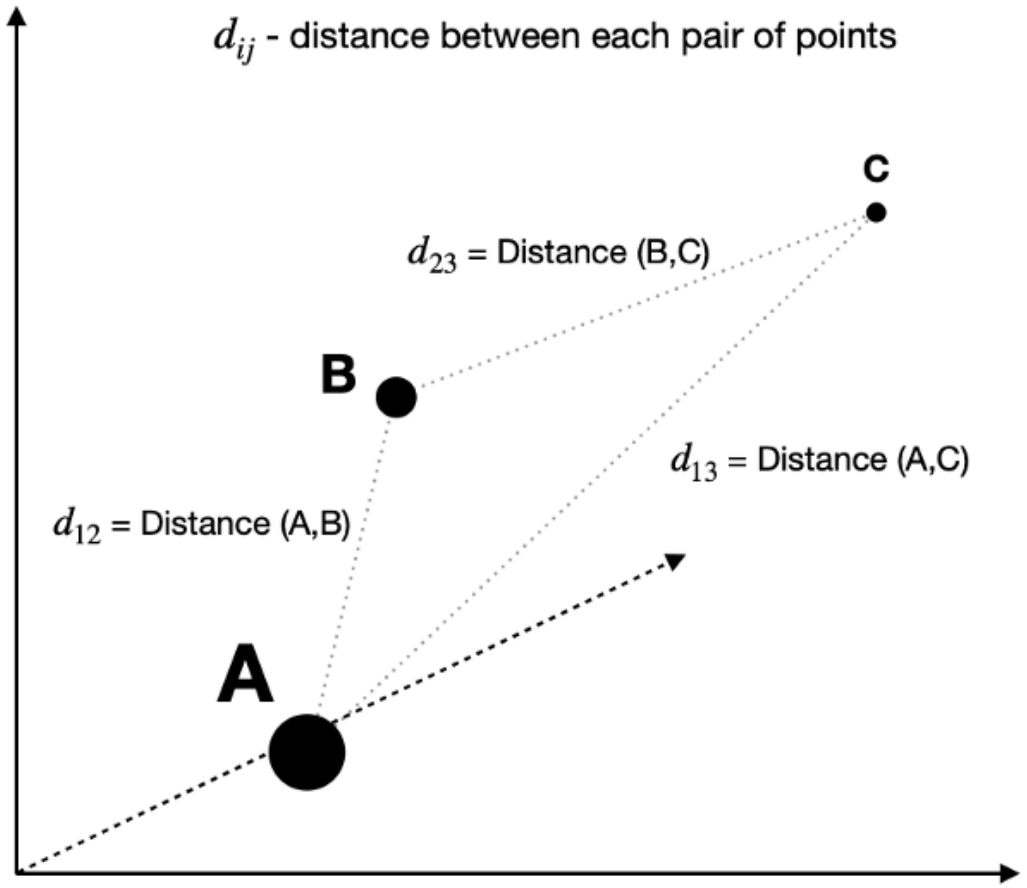
Let's implement very simple MDS so that we (and when I say we, I mean I!) know what is going on. The input to the MDS algorithm is our proximity matrix. There are two kinds of classical MDS that we could use:

- Classical (metric) MDS is for data that has metric properties, like actual distances from a map or calculated from a vector
- Nonmetric MDS is for more ordinal data (such as human-provided similarity ratings) for which we can say a 1 is more similar than a 2, but there is no defined (metric) distance between the values of 1 and 2.

For this post, we will walk through the algorithm for classical MDS, with metric data of course! I'm going to use one of Matlab's random datasets that is called "cities" and is made up of different ratings for a large set of US cities.

Steps used by metric MDS algorithm

Step 1 — The algorithm calculates distances between each pair of points, as illustrated below.



Step 2 — With the original distances known, the algorithm attempts to solve the optimization problem by finding a set of coordinates in a lower-dimensional that minimizes the value of Stress.

$$Stress_D(x_1, x_2, \ldots, x_N) = \sqrt{\sum_{i \neq j = 1,\ldots,N} (d_{ij} - ||x_i - x_j||)^2}$$

The goal of the algorithm is to minimize the value of stress.

Where $x_1, \ldots, x_N$ are data points with their new set of coordinates in lower dimensional space.

d_{ij} is the actual distance we have calculated between the two corresponding data points in their original dimensional space.

$||x_i - x_j||$ is the distance between the two corresponding data points in their lower dimensional space.

The closer the value of $||x_i - x_j||$ is to d_{ij} the smaller will be the value of stress.

Multiple approaches can be used to optimize the above cost function, such as Kruskal's steepest descent method or De Leeuw's iterative majorization method. However, I will not delve into maths this time to keep this article focused on high-level explanation.

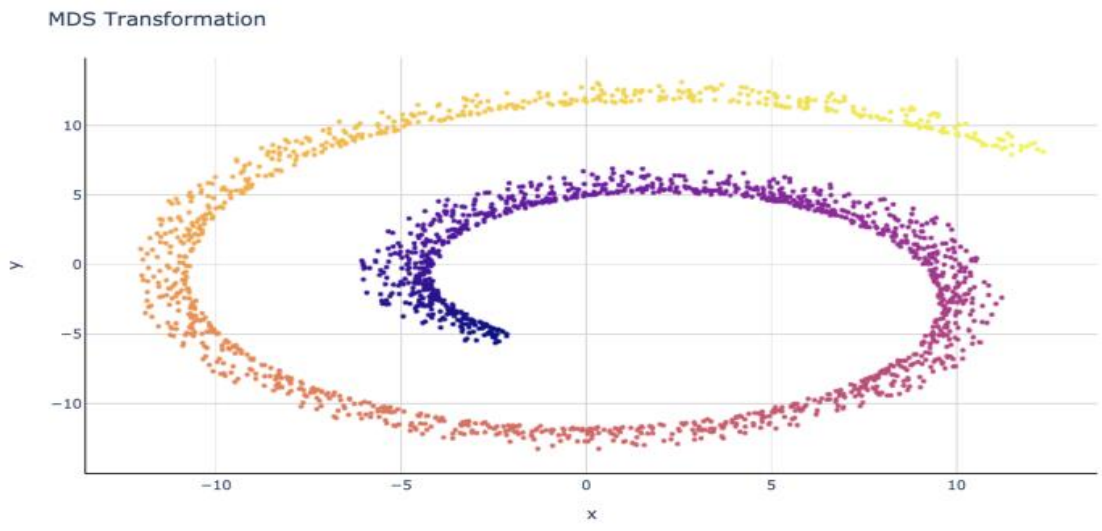
One important thing to note is that both aforementioned methods are iterative approaches, sometimes giving different results since they are sensitive to the initial starting position.

However, Sklearn's implementation of MDS allows us to specify how many times we want to initialize the process. In the end, the configuration with the lowest stress is picked as the final result.

Performing MDS

We will now use MDS to map this 3D structure to 2 dimensions while preserving distances between points as best as possible. Note that the depth of the swiss roll is smaller than its height and width. We expect this feature to be preserved in the 2D graph.

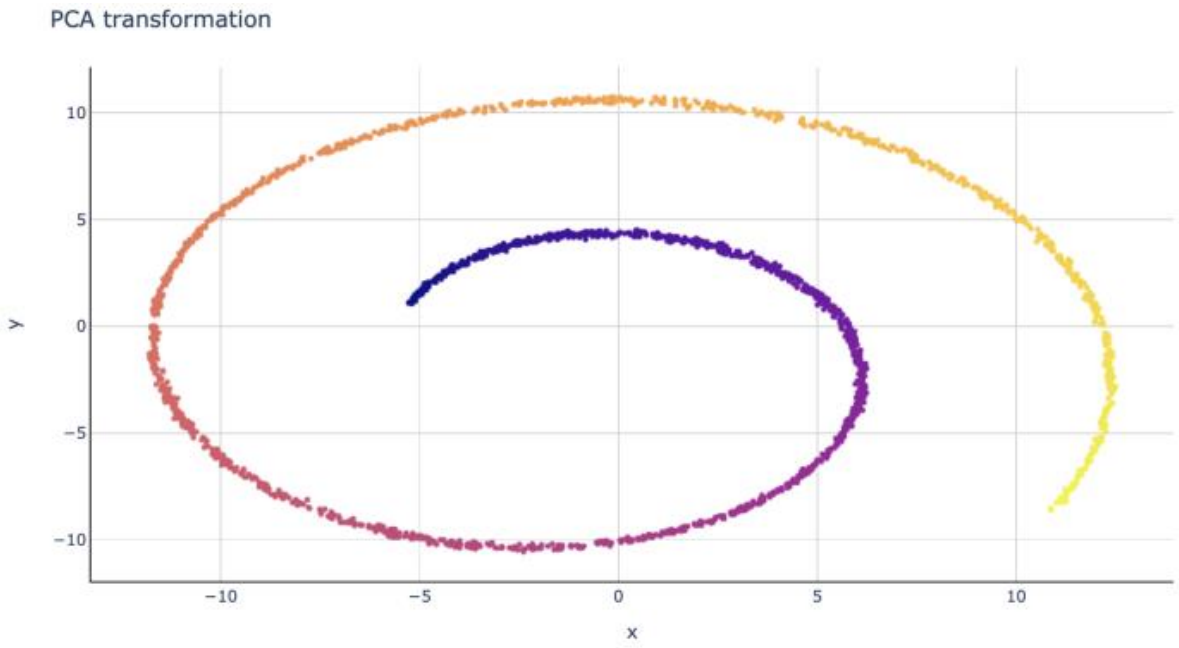
MDS TRANSFOMATION:



The results are pretty good since we could preserve the global structure while at the same time not losing the separation observed between points in the original depth dimension.

While it depends on the exact problem we want to solve, MDS seems to perform better in this scenario than PCA (Principal Component Analysis). For comparison, the below graph shows a 2D representation of the same 3D swiss roll after applying PCA transformation.

PCA TRANSFORMATION



As you can see, PCA gives us a result that looks like a picture from one side of the swiss roll, failing to preserve depth information from the third dimension.

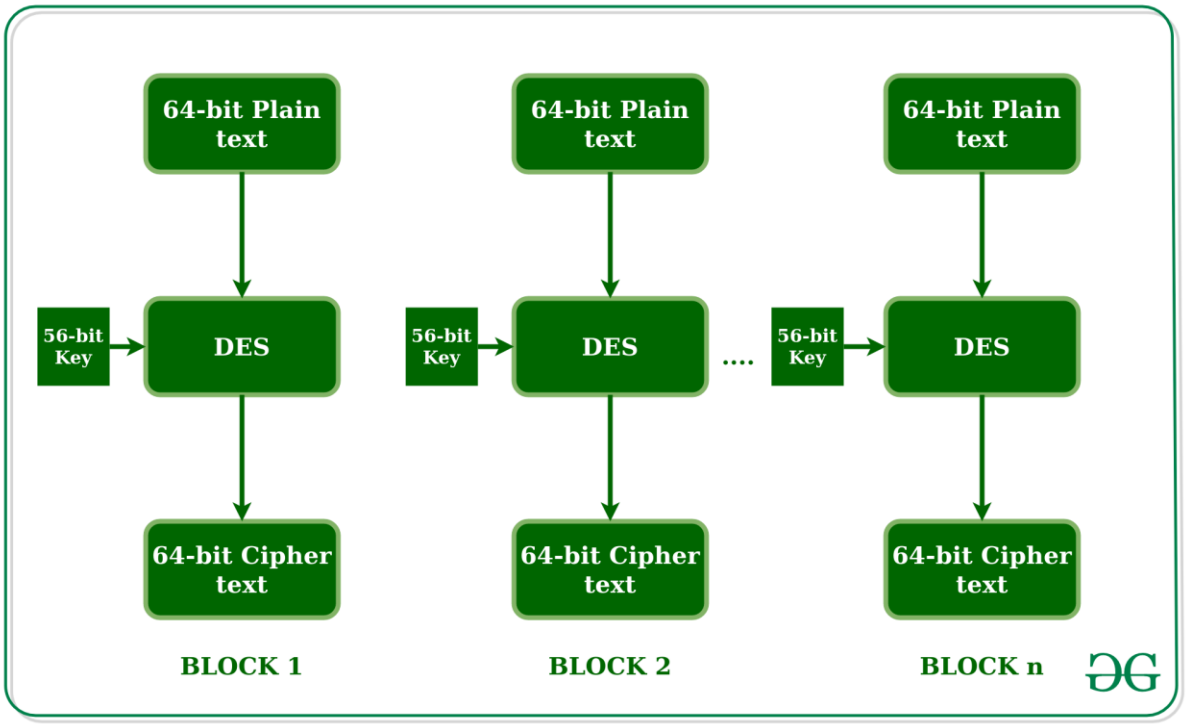
Conclusions:

Multidimensional Scaling is a good technique to use when you wish to preserve both global and local structures of your high-dimensional data. This is achieved by keeping distances between points in lower dimensions as similar as possible to distances in the original high-dimensional space.
However, if your analysis requires you to focus more on the global structures, you may wish to use PCA.

Experiment 3

AIM: Apply DES algorithm for practical application

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



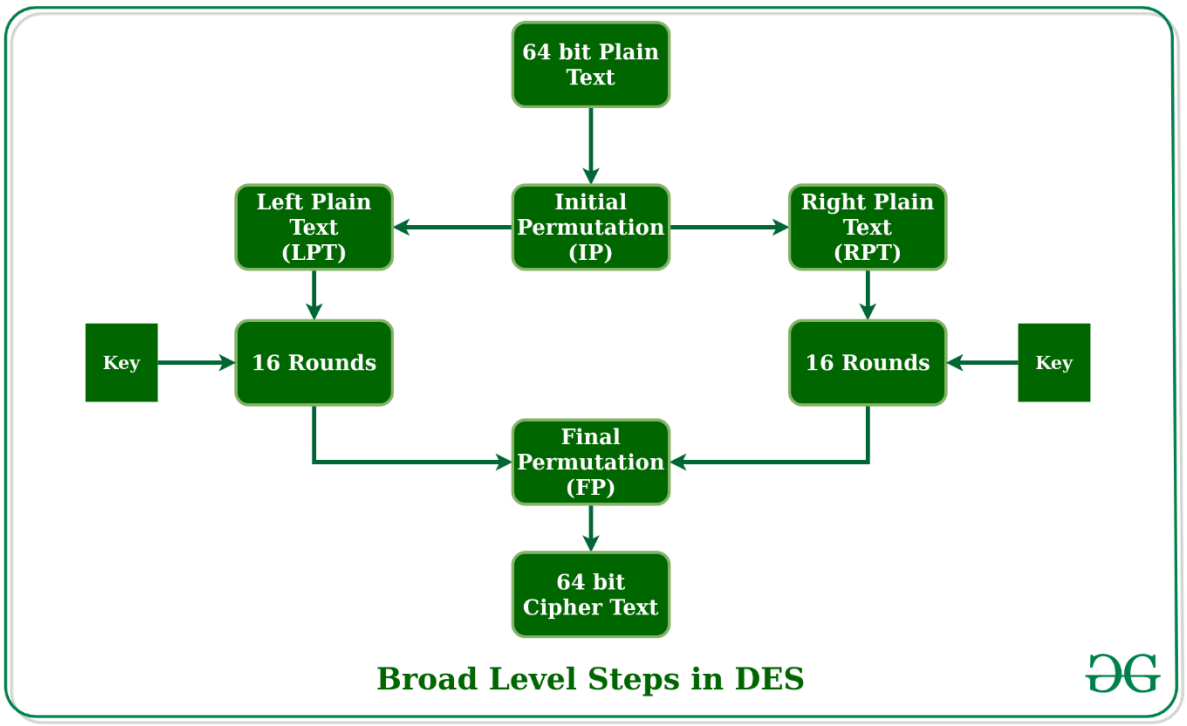
We have mentioned that DES uses a 56-bit key. Actually, the initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

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

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



Broad Level Steps in DES

Initial Permutation (IP):

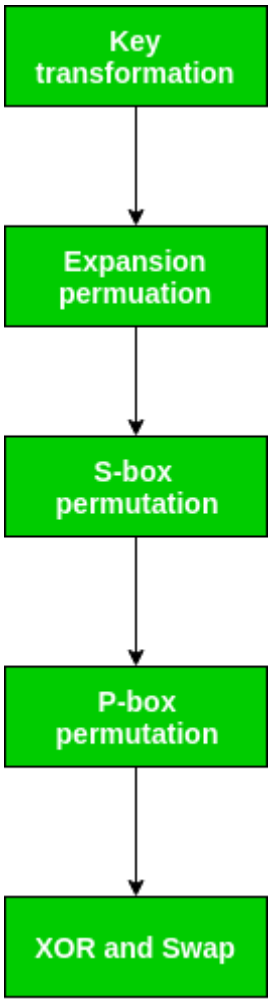
As we have noted, the initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original 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 to all the other bit positions shown 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 is 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 the 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 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 numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the 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 bits are selected. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit number 17 moves to 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 a selection of a 48-bit subset 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 makes DES not easy to crack.

Step-2: Expansion Permutation:

Recall that after the initial permutation, we had two 32-bit plain text areas called 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 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.

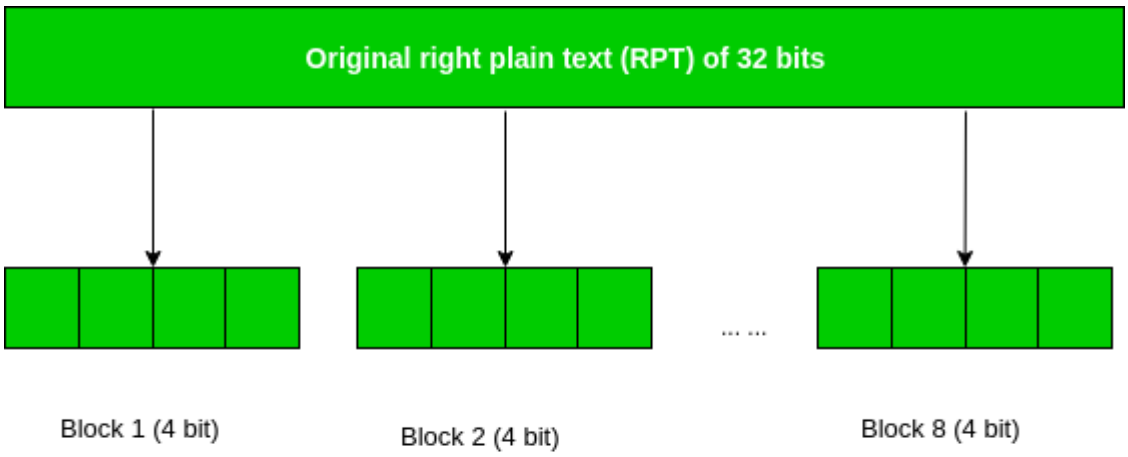


Figure - division of 32 bit RPT into 8 bit blocks

This process results in expansion as well as a permutation of the input bit while creating output. The 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 the resulting output is given to the next step, which is the **S-Box substitution**.

Experiment 4

AIM: Apply AES algorithm for practical applications

THEORY: Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S National Institute of Standards and Technology (NIST) in 2001. AES is widely used today as it is a much stronger than DES and triple DES despite being harder to implement. Points to remember

- AES is a block cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of 128 bits each.

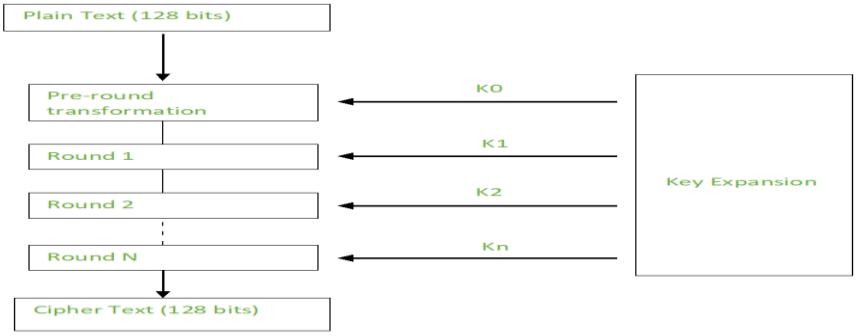
That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text as output. AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

Working of the cipher :
AES performs operations on bytes of data rather than in bits. Since the block size is 128 bits, the cipher processes 128 bits (or 16 bytes) of the input data at a time.

The number of rounds depends on the key length as follows :

- 128 bit key – 10 rounds
- 192 bit key – 12 rounds
- 256 bit key – 14 rounds

Creation of Round keys :
A Key Schedule algorithm is used to calculate all the round keys from the key. So the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.



Encryption :
AES considers each block as a 16 byte (4 byte x 4 byte = 128) grid in a column major arrangement.
[b0 | b4 | b8 | b12 |
| b1 | b5 | b9 | b13 |
| b2 | b6 | b10| b14 |
| b3 | b7 | b11| b15]
Each round comprises of 4 steps :

- SubBytes
- ShiftRows
- MixColumns
- Add Round Key

The last round doesn't have the MixColumns round.

The SubBytes does the substitution and ShiftRows and MixColumns performs the permutation in the algorithm.

SubBytes :
This step implements the substitution.
In this step each byte is substituted by another byte. Its performed using a lookup table also called the S-box. This substitution is done in a way that a byte is never substituted by itself and also not substituted by another byte which is a compliment of the current byte. The result of this step is a 16 byte (4 x 4) matrix like before.

The next two steps implement the permutation.

ShiftRows :
This step is just as it sounds. Each row is shifted a particular number of times.

- The first row is not shifted
- The second row is shifted once to the left.
- The third row is shifted twice to the left.
- The fourth row is shifted thrice to the left.

(A left circular shift is performed.)

[b0 | b1 | b2 | b3] [b0 | b1 | b2 | b3]
| b4 | b5 | b6 | b7 | -> | b5 | b6 | b7 | b4 |
| b8 | b9 | b10 | b11 | | b10 | b11 | b8 | b9 |
[b12 | b13 | b14 | b15] [b15 | b12 | b13 | b14]

MixColumns :
This step is basically a matrix multiplication. Each column is multiplied with a

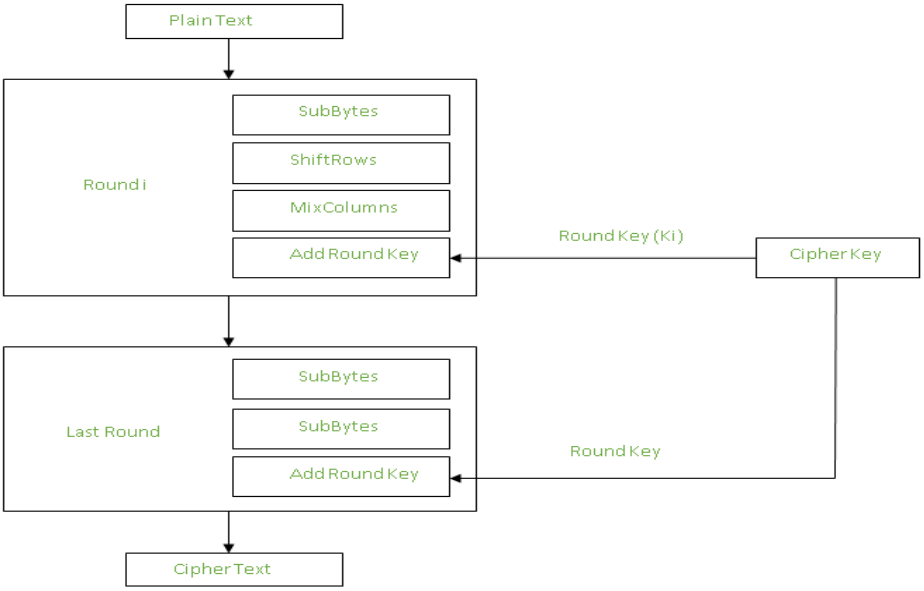
specific matrix and thus the position of each byte in the column is changed as a result.

This step is skipped in the last round.

[c0] [2 3 1 1] [b0]
| c1 | = | 1 2 3 1 | | b1 |
| c2 | | 1 1 2 3 | | b2 |
[c3] [3 1 1 2] [b3]

Add Round Keys :

Now the resultant output of the previous stage is XOR-ed with the corresponding round key. Here, the 16 bytes is not considered as a grid but just as 128 bits of data.



After all these rounds 128 bits of encrypted data is given back as output. This process is repeated until all the data to be encrypted undergoes this process.

Decryption :

The stages in the rounds can be easily undone as these stages have an opposite to it which when performed reverts the changes.Each 128 blocks goes through the 10,12 or 14 rounds depending on the key size.

The stages of each round in decryption is as follows :

- Add round key
- Inverse MixColumns
- ShiftRows
- Inverse SubByte

The decryption process is the encryption process done in reverse so i will explain the steps with notable differences.

Inverse MixColumns :

This step is similar to the MixColumns step in encryption, but differs in the matrix used to carry out the operation.

```
[ b0 ]         [ 14  11  13  9  ]     [ c0 ]
| b1 |  =      | 9   14  11  13 |     | c1 |
| b2 |         | 13  9   14  11 |     | c2 |
[ b3 ]         [ 11  13  9   14 ]     [ c3 ]
```

Inverse SubBytes :

Inverse S-box is used as a lookup table and using which the bytes are substituted during decryption.

Summary :

AES instruction set is now integrated into the CPU (offers throughput of several GB/s)to improve the speed and security of applications that use AES for encryption and decryption. Even though its been 20 years since its introduction we have failed to break the AES algorithm as it is infeasible even with the current technology. Till date the only vulnerability remains in the implementation of the algorithm.

Experiment 5

AIM: Implement RSA Algorithm using HTML and JavaScript

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.

Since this is asymmetric, nobody else except browser can decrypt the data even if a third party has public key of browser.

The idea! The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024 bit keys could be broken in the near future. But till now it seems to be an infeasible task

Let us learn the mechanism behind RSA algorithm :

>> 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 = 89^e mod n**.
- Thus our Encrypted Data comes out to be 1394

Now we will decrypt **1394** :

- **Decrypted Data = c^d mod n**.
- Thus our Encrypted Data comes out to be 89

8 = H and I = 9 i.e. "HI".

Experiment 6

AIM: Implements the Diffie -Hellman Key exchange algorithm for a given problem

THEORY: Diffie Helman key exchange is a way in by which we excahange the key securely over a public channel and was the first algorithm been implemented. The Algorithm is as follows

1. The two communicating parties Alice and Bob, agree on a two large numbers p and g.
2. . Alice chooses some large random integer xA < p and keeps it secret. Likewise Bob chooses xB < p and keeps it secret. These are their "private keys".
3. Alice computes her "public key" yA g x A (mod p) and sends it to Bob using insecure communication. Bob computes his public key yB g x B and sends it to Alice. Here 0 < yA < p, 0 < yB < p. As already mentioned, sending these public keys with insecure communication is safe because it would be too hard for someone to compute xA from yA or xBfrom yB, just like the powers of 2 above.
4. Alice computes zA yB x A (mod p) and Bob computes zB yA x B (mod p). Here zA < p, zB < p.

Program

```
 Package diffie.helmanalgo;
 import java.util.Scanner;
 public class DiffieHelmanAlgo
{
public static double alice(double n, double g, double x)
{
 double a, a1;
          a1=Math.pow(g,x);
          a=a1%n;
     return(a);
}
public static double bob(double n, double g, double y)
{
double b,b1,k2,t2;
          b1=Math.pow(g,y);
          b=b1%n;
return(b);
}
Public static void main(String args[]){
Double g,x,y,a,b,k1,k2,n;
```

```
Scanner input=new Scanner(System.in);
System.out.print(“Enter value of n=>”);
n =input.nextDouble();
System.out.print(“Enter value of g=>”);
g =input.nextDouble();
System.out.print(“Enter value of x=>”);
x =input.nextDouble();


System.out.print(“Enter value of y=>”);
y=input.nextDouble();
a=alice(n,g,x);
System.out.print(“alice end value: ”+a);
b =bob(n,g,y);
System.out.print(“bob end value:”+a);
k1 =alice(n,b,x);

System.out.print(“valueofk1:”+k1);
k2 =alice(n,a,y);
System.out.print(“valueofk2:”+k2);

}
   }
```

Experiment 7

AIM: Calculate the message digest of a text using the SHA-1 algorithm

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**.

MessageDigest 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.

Examples:

Input : hello world **Output** :
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed **Input** :
GeeksForGeeks **Output** : addf120b430021c36c232c99ef8d926aea2acd6b

Below program shows the implementation of SHA-1 hash in Java.

// Java program to calculate SHA-1 hash value

```
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));
        }
}
```

Experiment 8

AIM: Demonstrate instruction detection system(ids) using any tool eg. Snort or any other s/w.

THEORY: An **Intrusion Detection System (IDS)** is a system that monitors **network traffic** for suspicious activity and issues alerts when such activity is discovered. It is a software application that scans a network or a system for the harmful activity or policy breaching. Any malicious venture or violation is normally reported either to an administrator or collected centrally using a security information and event management (SIEM) system. A SIEM system integrates outputs from multiple sources and uses alarm filtering techniques to differentiate malicious activity from false alarms.

Although intrusion detection systems monitor networks for potentially malicious activity, they are also disposed to false alarms. Hence, organizations need to fine-tune their IDS products when they first install them. It means properly setting up the intrusion detection systems to recognize what normal traffic on the network looks like as compared to malicious activity.

Intrusion prevention systems also monitor network packets inbound the system to check the malicious activities involved in it and at once send the warning notifications.

Classification of Intrusion Detection System:

IDS are classified into 5 types:

1. **Network Intrusion Detection System (NIDS)**
   Network intrusion detection systems (NIDS) are set up at a planned point within the network to examine traffic from all devices on the network. It performs an observation of passing traffic on the entire subnet and matches the traffic that is passed on the subnets to the collection of known attacks. Once an attack is identified or abnormal behavior is observed, the alert can be sent to the administrator. An example of a NIDS is installing it on the subnet where firewalls are located in order to see if someone is trying to crack the firewall.

2. **Host Intrusion Detection System (HIDS)**

   Host intrusion detection systems (HIDS) run on independent hosts or devices on the network. A HIDS monitors the incoming and outgoing packets from the device only and will alert the administrator if suspicious or malicious activity is detected. It takes a snapshot of existing system files and compares it with the previous snapshot. If the analytical system files were edited or deleted, an alert is sent to the administrator to investigate. An example of HIDS usage can be seen on mission-critical machines, which are not expected to change their layout.

3. **Protocol-based Intrusion Detection System (PIDS)**

   Protocol-based intrusion detection system (PIDS) comprises a system or agent that would consistently resides at the front end of a server, controlling and interpreting the protocol between a user/device and the server. It is trying to secure the web server by regularly monitoring the HTTPS protocol stream and accept the related HTTP protocol. As HTTPS is un-encrypted and before instantly entering its web presentation layer then this system would need to reside in this interface, between to use the HTTPS.

4. **Application Protocol-based Intrusion Detection System (APIDS)**

   Application Protocol-based Intrusion Detection System (APIDS) is a system or agent that generally resides within a group of servers. It identifies the intrusions by monitoring and interpreting the communication on application-specific protocols. For example, this would monitor the SQL protocol explicit to the middleware as it transacts with the database in the web server.

5. **Hybrid Intrusion Detection System**

   Hybrid intrusion detection system is made by the combination of two or more approaches of the intrusion detection system. In the hybrid intrusion detection system, host agent or system data is combined with network information to develop a complete view of the network system. Hybrid intrusion detection system is more effective in comparison to the other intrusion detection system. Prelude is an example of Hybrid IDS.

Detection Method of IDS

1. **Signature-based-Method:**
   Signature-based IDS detects the attacks on the basis of the specific patterns such as number of bytes or number of 1's or number of 0's in the network traffic. It also detects on the basis of the already known malicious instruction sequence that is used by the malware. The detected patterns in the IDS are known as signatures.
   Signature-based IDS can easily detect the attacks whose pattern (signature) already exists in system but it is quite difficult to detect the new malware attacks as their pattern (signature) is not known.

2. **Anomaly-based-Method:**
   Anomaly-based IDS was introduced to detect unknown malware attacks as new malware are developed rapidly. In anomaly-based IDS there is use of machine learning to create a trustful activity model and anything coming is compared with that model and it is declared suspicious if it is not found in model. Machine learning-based method has a better-generalized property in comparison to signature-based IDS as these models can be trained according to the applications and hardware configurations.

Comparison of IDS with Firewalls:
IDS and firewall both are related to network security but an IDS differs from a firewall as a firewall looks outwardly for intrusions in order to stop them from happening. Firewalls restrict access between networks to prevent intrusion and if an attack is from inside the network it doesn't signal. An IDS describes a suspected intrusion once it has happened and then signals an alarm.