**1 .CAESER CIPHER**

#include <iostream>

using namespace std;

string encrypt(string text, int s)

{

string result = "";

for (int i=0;i<text.length();i++)

{

result += char(int(text[i]+s-97)%26+65);

}

return result;

}

string decrypt(string text, int s)

{

string result1= "";

for (int i=0;i<text.length();i++)

{

result1 += char(int(text[i]-s-65)%26+97);

}

return result1;

}

int main()

{

string text;

cout<<"Enter Plain Text"<<endl;

cin>>text;

int s;

cout<<"Enter Shift Key"<<endl;

cin>>s;

cout << "Cipher: " << encrypt(text, s)<<endl;

cout << "Original: " << decrypt(encrypt(text,s), s);

return 0;

}

**2.BRUTEFORCE ATTACK ON CAESAR**

#include<iostream>

using namespace std;

string decrypt(string s, int k)

{

string result= "";

for (int i=0;i<str.length();i++)

{

result += char(int(str[i]-k-65)%26+97);

}

return result;

}

int main(){

string str;

int k;

cout<<"enter plain text"<<endl;

cin>>str;

for(int k=1;k<26;k++){

cout<<decrypt(str,k)<<endl;

}

return 0;

}

**3.PLAYFAIR**

#include <bits/stdc++.h>

using namespace std;

#define SIZE 30

void toLowerCase(char plain[], int ps)

{

int i;

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

if (plain[i] > 64 && plain[i] < 91)

plain[i] += 32;

}

}

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;

}

void genkeytable(char key[], int ks, char keyT[5][5])

{

int i, j, k, flag = 0;

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;

}

}

}

}

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;

}

}

}

}

int mod5(int a) { return (a % 5); }

int prepare(char str[], int ptrs)

{

if (ptrs % 2 != 0) {

str[ptrs++] = 'z';

str[ptrs] = '\0';

}

return ptrs;

}

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]];

}

}

}

void encryptPFCipher(char str[], char key[])

{

char ps, ks, keyT[5][5];

ks = strlen(key);

ks = removespaces(key, ks);

toLowerCase(key, ks);

ps = strlen(str);

toLowerCase(str, ps);

ps = removespaces(str, ps);

ps = prepare(str, ps);

genkeytable(key, ks, keyT);

encrypt(str, keyT, ps);

}

int main()

{

char str[SIZE], key[SIZE];

strcpy(key, "Krishna");

cout << "Key text: " << key << "\n";

strcpy(str, "Kittu");

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

encryptPFCipher(str, key);

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

return 0;

}

**4.VIGENERE**

#include<iostream>

#include<string>

#include<math.h>

using namespace std;

string encrypt(string s,string k){

string res="";

int a;

int j=0;

for(int i=0;i<s.length();i++){

a=int(k[j])-97;

res+=char(int(s[i]+a-97)%26+65);

j++;

j=j%k.length();

}

return res;

}

string decrypt(string s,string k){

string res="";

int a;

int j=0;

for(int i=0;i<s.length();i++){

a=int(k[j])-97;

res+=char(abs(int(s[i]-a-65))%26+97);

j++;

j=j%k.length();

}

return res;

}

int main(){

string s;

string k;

cout<<"enter the plain text: "<<endl;

cin>>s;

cout<<"enter the key value: "<<endl;

cin>>k;

if(s.length()>=k.length()){

cout<<encrypt(s,k)<<endl;

}

cout<<decrypt(encrypt(s,k),k)<<endl;

return 0;

}

**5.AFFINE**

#include<bits/stdc++.h>

using namespace std;

const int a = 23;

const int b = 10;

string encryptmsg(string msg)

{

string cipher = "";

for (int i = 0; i < msg.length(); i++)

{

if(msg[i]!=' ')

cipher = cipher +(char) ((((a \* (msg[i]-'A') ) + b) % 26) + 'A');

else

cipher += msg[i];

}

return cipher;

}

string decryptcipher(string cipher)

{

string msg = "";

int a\_inverse = 0;

int flag = 0;

for (int i = 0; i < 26; i++)

{

flag = (a \* i) % 26;

if (flag == 1)

{

a\_inverse = i;

}

}

for (int i = 0; i < cipher.length(); i++)

{

if(cipher[i]!=' ')

msg = msg +(char) (((a\_inverse \* ((cipher[i]+'A' - b)) % 26)) + 'A');

else

msg += cipher[i];

}

return msg;

}

int main(void)

{

string msg ;

cout<<"enter a message"<<endl;

getline(cin,msg);

string ciphertext = encryptmsg(msg);

cout << "Encrypted Message is : " << ciphertext<<endl;

cout << "Decrypted Message is: " << decryptcipher(ciphertext);

return 0;

}

**6.TRANSPOSITION**

#include<bits/stdc++.h>

using namespace std;

map<int,int> keymap;

void setpermutation(string key)

{

for(int i=0; i < key.length(); i++)

{

keymap[key[i]] = i;

}

}

string encryptmsg(string msg,string key)

{

int row,col,j;

string cipher = "";

col = key.length();

row = msg.length()/col;

if (msg.length() % col)

row += 1;

char matrix[row][col];

for (int i=0,k=0; i < row; i++)

{

for (int j=0; j<col; )

{

if(msg[k] == '\0')

{

matrix[i][j] = '\_';

j++;

}

if( isalpha(msg[k]) || msg[k]==' ')

{

matrix[i][j] = msg[k];

j++;

}

k++;

}

}

for (map<int,int>::iterator ii = keymap.begin(); ii!=keymap.end(); ++ii)

{

j=ii->second;

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

{

if( isalpha(matrix[i][j]) || matrix[i][j]==' ' || matrix[i][j]=='\_')

cipher += matrix[i][j];

}

}

return cipher;

}

string decryptmsg(string cipher,string key)

{

int col = key.length();

int row = cipher.length()/col;

char cipherMat[row][col];

for (int j=0,k=0; j<col; j++)

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

cipherMat[i][j] = cipher[k++];

int index = 0;

for( map<int,int>::iterator ii=keymap.begin(); ii!=keymap.end(); ++ii)

ii->second = index++;

char decCipher[row][col];

map<int,int>::iterator ii=keymap.begin();

int k = 0;

for (int l=0,j; key[l]!='\0'; k++)

{

j = keymap[key[l++]];

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

{

decCipher[i][k]=cipherMat[i][j];

}

}

string msg = "";

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

{

for(int j=0; j<col; j++)

{

if(decCipher[i][j] != '\_')

msg += decCipher[i][j];

}

}

return msg;

}

int main(void)

{

string msg,key ;

cout<<"enter key"<<endl;

getline(cin,key);

cout<<"enter message"<<endl;

getline(cin,msg);

setpermutation(key);

string cipher = encryptmsg(msg,key);

cout << "Encrypted Message: " << cipher << endl;

cout << "Decrypted Message: " << decryptmsg(cipher,key) << endl;

return 0;

}

**7.ATTACK ON AFFINE**

#include<bits/stdc++.h>

using namespace std;

int gcd(int a, int b)

{

return b == 0 ? a : gcd(b, a % b);

}

int main(){

string ct;

cout<<"enter the plain text;"<<endl;

getline(cin,ct);

for (int a=1;a<26;a++){

for (int b=0;b<26;b++){

if (gcd(a,b)==1){

string msg = "";

int a\_inv = 0;

int flag = 0;

for (int i = 0; i < 26; i++)

{

flag = (a \* i) % 26;

if (flag == 1)

{

a\_inv = i;

}

}

for (int i = 0; i < ct.length(); i++)

{

if(ct[i]!=' ')

msg = msg +(char) (((a\_inv \* ((ct[i]+'A' - b)) % 26)) + 'A');

else

msg += ct[i];

}

cout<<"key:"<<a<<" "<<b<<":"<<msg<<endl;

}

}

}

return 0;

}

**8.RC4**

#include <iostream>

#include <string>

#include<vector>

using namespace std;

vector<int> permute(vector<int>, vector<int>);

string encrypt(vector<int>s , vector<int> t, string p);

string decrypt(vector<int>s, vector<int> t, string p);

int main() {

string plaintext;

cout<<"enter plain text"<<endl;

cin>>plaintext;

vector<int> S(256);

vector<int> T(256);

int key[] = { 1,2,3,6 };

int tmp = 0;

for (int i = 0; i < 256;i++) {

S[i] = i;

T[i] = key[( i % (sizeof(key)/sizeof(\*key)) )];

}

S = permute(S, T);

for (int i = 0; i < 256 ;i++) {

cout << S[i] << " ";

if ((i + 1) % 16 == 0)

cout << endl;

}

cout << endl;

string p = encrypt(S, T, plaintext);

cout << "Message: " << plaintext << endl;

cout << "Encrypted Message: " << " " << p << endl;

cout << "Decrypted Message: " << decrypt(S, T, p) << endl << endl;

return 0;

}

string encrypt(vector<int>s, vector<int> t, string p) {

int i = 0;

int j = 0;

int tmp = 0;

int k = 0;

int b;

int c;

int \* cipher = new int [p.length()];

string cipher\_T;

cout << "Keys Generated for plaintext: ";

for (int r = 0; r < p.length(); r++) {

i = (i + 1) % 256;

j = (j + s[i]) % 256;

b = s[i];

s[i] = s[j];

s[j] = b;

tmp = (s[i] + s[j]) % 256;

k = s[tmp];

cout << k << " ";

c = ((int)p[r] ^ k);

cipher[r] = c;

cipher\_T += (char)cipher[r];

}

cout << endl;

return cipher\_T;

}

string decrypt(vector<int>s, vector<int> t, string p) {

int i = 0;

int j = 0;

int tmp = 0;

int k = 0;

int b;

int c;

int \* plain = new int[p.length()];

string plain\_text;

for (int r = 0; r < p.length(); r++) {

i = (i + 1) % 256;

j = (j + s[i]) % 256;

b = s[i];

s[i] = s[j];

s[j] = b;

tmp = (s[i] + s[j]) % 256;

k = s[tmp];

c = ((int)p[r] ^ k);

plain[r] = c;

plain\_text += (char)plain[r];

}

return plain\_text;

}

vector<int> permute(vector<int> s, vector<int> t) {

int j = 0;

int tmp;

for (int i = 0; i< 256; i++) {

j = (j + s[i] + t[i]) % 256;

tmp = s[i];

s[i] = s[j];

s[j] = tmp;

}

return s;

}

**8. RC4**

def key\_scheduling(key):

sched = [i for i in range(0, 256)]

i = 0

for j in range(0, 256):

i = (i + sched[j] + key[j % len(key)]) % 256

tmp = sched[j]

sched[j] = sched[i]

sched[i] = tmp

return sched

def stream\_generation(sched):

stream = []

i = 0

j = 0

while True:

i = (1 + i) % 256

j = (sched[i] + j) % 256

tmp = sched[j]

sched[j] = sched[i]

sched[i] = tmp

yield sched[(sched[i] + sched[j]) % 256]

def encrypt(text, key):

text = [ord(char) for char in text]

key = [ord(char) for char in key]

sched = key\_scheduling(key)

key\_stream = stream\_generation(sched)

ciphertext = ''

for char in text:

enc = str(hex(char ^ next(key\_stream))).upper()

ciphertext += (enc)

return ciphertext

def decrypt(ciphertext, key):

ciphertext = ciphertext.split('0X')[1:]

ciphertext = [int('0x' + c.lower(), 0) for c in ciphertext]

key = [ord(char) for char in key]

sched = key\_scheduling(key)

key\_stream = stream\_generation(sched)

plaintext = ''

for char in ciphertext:

dec = str(chr(char ^ next(key\_stream)))

plaintext += dec

return plaintext

if \_\_name\_\_ == '\_\_main\_\_':

ed = input('Enter E for Encrypt, or D for Decrypt: ').upper()

if ed == 'E':

plaintext = input('Enter your plaintext: ')

key = input('Enter your secret key: ')

result = encrypt(plaintext, key)

print('Result: ')

print(result)

elif ed == 'D':

ciphertext = input('Enter your ciphertext: ')

key = input('Enter your secret key: ')

result = decrypt(ciphertext, key)

print('Result: ')

print(result)

else:

print('Error in input - try again.')

**9.DES**

#include <bits/stdc++.h>

using namespace std;

string hex2bin(string s)

{

unordered\_map<char, string> mp;

mp['0'] = "0000";

mp['1'] = "0001";

mp['2'] = "0010";

mp['3'] = "0011";

mp['4'] = "0100";

mp['5'] = "0101";

mp['6'] = "0110";

mp['7'] = "0111";

mp['8'] = "1000";

mp['9'] = "1001";

mp['A'] = "1010";

mp['B'] = "1011";

mp['C'] = "1100";

mp['D'] = "1101";

mp['E'] = "1110";

mp['F'] = "1111";

string bin = "";

for (int i = 0; i < s.size(); i++) {

bin += mp[s[i]];

}

return bin;

}

string bin2hex(string s)

{

unordered\_map<string, string> mp;

mp["0000"] = "0";

mp["0001"] = "1";

mp["0010"] = "2";

mp["0011"] = "3";

mp["0100"] = "4";

mp["0101"] = "5";

mp["0110"] = "6";

mp["0111"] = "7";

mp["1000"] = "8";

mp["1001"] = "9";

mp["1010"] = "A";

mp["1011"] = "B";

mp["1100"] = "C";

mp["1101"] = "D";

mp["1110"] = "E";

mp["1111"] = "F";

string hex = "";

for (int i = 0; i < s.length(); i += 4) {

string ch = "";

ch += s[i];

ch += s[i + 1];

ch += s[i + 2];

ch += s[i + 3];

hex += mp[ch];

}

return hex;

}

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;

}

string xor\_(string a, string b)

{

string ans = "";

for (int i = 0; i < a.size(); i++) {

if (a[i] == b[i]) {

ans += "0";

}

else {

ans += "1";

}

}

return ans;

}

string encrypt(string pt, vector<string> rkb, vector<string> rk)

{

pt = hex2bin(pt);

int initial\_perm[64] = { 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, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7 };

pt = permute(pt, initial\_perm, 64);

cout << "After initial permutation: " << bin2hex(pt) << endl;

// Splitting

string left = pt.substr(0, 32);

string right = pt.substr(32, 32);

cout << "After splitting: L0=" << bin2hex(left)

<< " R0=" << bin2hex(right) << endl;

int exp\_d[48] = { 32, 1, 2, 3, 4, 5, 4, 5,

6, 7, 8, 9, 8, 9, 10, 11,

12, 13, 12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21, 20, 21,

22, 23, 24, 25, 24, 25, 26, 27,

28, 29, 28, 29, 30, 31, 32, 1 };

int s[8][4][16] = { { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,

0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,

4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,

15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 },

{ 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,

3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,

0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,

13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 },

{ 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,

13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,

13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,

1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 },

{ 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,

13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,

10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,

3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14 },

{ 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,

14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,

4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,

11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 },

{ 12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,

10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,

9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,

4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13 },

{ 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,

13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,

1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,

6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12 },

{ 13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,

1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,

7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,

2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11 } };

int per[32] = { 16, 7, 20, 21,

29, 12, 28, 17,

1, 15, 23, 26,

5, 18, 31, 10,

2, 8, 24, 14,

32, 27, 3, 9,

19, 13, 30, 6,

22, 11, 4, 25 };

cout << endl;

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

string right\_expanded = permute(right, exp\_d, 48);

string x = xor\_(rkb[i], right\_expanded);

string op = "";

for (int i = 0; i < 8; i++) {

int row = 2 \* int(x[i \* 6] - '0') + int(x[i \* 6 + 5] - '0');

int col = 8 \* int(x[i \* 6 + 1] - '0') + 4 \* int(x[i \* 6 + 2] - '0') + 2 \* int(x[i \* 6 + 3] - '0') + int(x[i \* 6 + 4] - '0');

int val = s[i][row][col];

op += char(val / 8 + '0');

val = val % 8;

op += char(val / 4 + '0');

val = val % 4;

op += char(val / 2 + '0');

val = val % 2;

op += char(val + '0');

}

op = permute(op, per, 32);

x = xor\_(op, left);

left = x;

if (i != 15) {

swap(left, right);

}

cout << "Round " << i + 1 << " " << bin2hex(left) << " "

<< bin2hex(right) << " " << rk[i] << endl;

}

string combine = left + right;

int final\_perm[64] = { 40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25 };

string cipher = bin2hex(permute(combine, final\_perm, 64));

return cipher;

}

int main()

{

string pt, key;

cout<<"enter plain text"<<endl;

cin>>pt;

cout<<"enter key"<<endl;

cin>>key;

key = hex2bin(key);

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);

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);

vector<string> rkb;

vector<string> rk;

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);

rkb.push\_back(RoundKey);

rk.push\_back(bin2hex(RoundKey));

}

cout << "\nEncryption:\n\n";

string cipher = encrypt(pt, rkb, rk);

cout << "\nCipher Text: " << cipher << endl;

cout << "\nDecryption\n\n";

reverse(rkb.begin(), rkb.end());

reverse(rk.begin(), rk.end());

string text = encrypt(cipher, rkb, rk);

cout << "\nPlain Text: " << text << endl;

}

10. DES

USING LIBRARY

pip install des

python setup.py install

from des import DesKey key0 = DesKey(b"some key") # for DES key1 = DesKey(b"a key for TRIPLE") # for 3DES, same as "a key for TRIPLEa key fo" key2 = DesKey(b"a 24-byte key for TRIPLE") # for 3DES key3 = DesKey(b"1234567812345678REAL\_KEY") # for DES, same as "REAL\_KEY"

key0.is\_single() # -> True key1.is\_triple() # -> True key2.is\_single() # -> False key3.is\_triple() # -> False

key0.encrypt(b"any long message")

key0.encrypt(b"abc", padding=True)

**Triple DES:**

Code:

def hex2bin(s):

mp = {'0' : "0000",

'1' : "0001", '2' : "0010", '3' : "0011", '4' : "0100", '5' : "0101", '6' : "0110", '7' :

"0111", '8' : "1000", '9' : "1001", 'A' : "1010", 'B' : "1011", 'C' : "1100", 'D' : "1101",

'E' : "1110", 'F' : "1111" }

bin = ""

for i in range(len(s)):

bin = bin + mp[s[i]]

return bin

def bin2hex(s):

mp = {"0000" : '0', "0001" : '1', "0010" : '2', "0011" : '3', "0100" : '4', "0101" :

'5', "0110" : '6', "0111" : '7', "1000" : '8', "1001" : '9', "1010" : 'A', "1011" : 'B', "1100"

: 'C', "1101" : 'D', "1110" : 'E', "1111" : 'F' }

hex = ""

for i in range(0,len(s),4):

ch = ""

ch = ch + s[i]

ch = ch + s[i + 1]

ch = ch + s[i + 2]

ch = ch + s[i + 3]

hex = hex + mp[ch]

return hex

def bin2dec(binary):

decimal, i= 0, 0

while(binary != 0):

dec = binary % 10

decimal = decimal + dec \* pow(2, i)

binary = binary//10

i += 1

return decimal

def dec2bin(num):

res = bin(num).replace("0b", "")

if(len(res)%4 != 0):

div = len(res) / 4

div = int(div)

counter =(4 \* (div + 1)) - len(res)

for i in range(0, counter):

res = '0' + res

return res

def permute(k, arr, n):

permutation = ""

for i in range(0, n):

permutation = permutation + k[arr[i] - 1]

return permutation

def shift\_left(k, nth\_shifts):

s = ""

for i in range(nth\_shifts):

for j in range(1,len(k)):

s = s + k[j]

s = s + k[0]

k = s

s = ""

return k

def xor(a, b):

ans = ""

for i in range(len(a)):

if a[i] == b[i]:

ans = ans + "0"

else:

ans = ans + "1"

return ans

initial\_perm = [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, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7]

exp\_d = [32, 1 , 2 , 3 , 4 , 5 , 4 , 5,

6 , 7 , 8 , 9 , 8 , 9 , 10, 11,

12, 13, 12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21, 20, 21,

22, 23, 24, 25, 24, 25, 26, 27,

28, 29, 28, 29, 30, 31, 32, 1 ]

per = [ 16, 7, 20, 21,

29, 12, 28, 17,

1, 15, 23, 26,

5, 18, 31, 10,

2, 8, 24, 14,

32, 27, 3, 9,

19, 13, 30, 6,

22, 11, 4, 25 ]

sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

[ 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

[ 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 ]],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 ]],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 ]],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14] ],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 ]],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13] ],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12] ],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11] ] ]

final\_perm = [ 40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25 ]

keyp = [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 ]

shift\_table = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1 ]

key\_comp = [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 ]

def encrypt(pt, key):

leftk = key[0:28]

rightk = key[28:56]

rkb = []

rk = []

for i in range(0, 16):

leftk = shift\_left(leftk, shift\_table[i])

rightk = shift\_left(rightk, shift\_table[i])

combine\_str = leftk + rightk

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

pt = hex2bin(pt)

pt = permute(pt, initial\_perm, 64)

print("After initial permutation: ", bin2hex(pt))

left = pt[0:32]

right = pt[32:64]

for i in range(0, 16):

right\_expanded = permute(right, exp\_d, 48)

xor\_x = xor(right\_expanded, rkb[i])

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3]

+ xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

sbox\_str = permute(sbox\_str, per, 32)

result = xor(left, sbox\_str)

left = result

if(i != 15):

left, right = right, left

print("Round ", i + 1, " ", bin2hex(left), " ", bin2hex(right), " ", rk[i])

combine = left + right

cipher\_text = permute(combine, final\_perm, 64)

return cipher\_text

def decrypt(ct, key):

leftk = key[0:28]

rightk = key[28:56]

rkb = []

rk = []

for i in range(0, 16):

leftk = shift\_left(leftk, shift\_table[i])

rightk = shift\_left(rightk, shift\_table[i])

combine\_str = leftk + rightk

round\_key = permute(combine\_str, key\_comp, 48)

rkb.append(round\_key)

rk.append(bin2hex(round\_key))

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

ct = hex2bin(ct)

ct = permute(ct, initial\_perm, 64)

print("After initial permutation: ", bin2hex(ct))

left = ct[0:32]

right = ct[32:64]

for i in range(0, 16):

right\_expanded = permute(right, exp\_d, 48)

xor\_x = xor(right\_expanded, rkb\_rev[i])

sbox\_str = ""

for j in range(0, 8):

row = bin2dec(int(xor\_x[j \* 6] + xor\_x[j \* 6 + 5]))

col = bin2dec(int(xor\_x[j \* 6 + 1] + xor\_x[j \* 6 + 2] + xor\_x[j \* 6 + 3]

+ xor\_x[j \* 6 + 4]))

val = sbox[j][row][col]

sbox\_str = sbox\_str + dec2bin(val)

sbox\_str = permute(sbox\_str, per, 32)

result = xor(left, sbox\_str)

left = result

if(i != 15):

left, right = right, left

print("Round ", i + 1, " ", bin2hex(left), " ", bin2hex(right), " ", rk\_rev[i])

combine = left + right

plain\_text = permute(combine, final\_perm, 64)

return plain\_text

pt = input("Enter Plaintext: ")

key1 = input("Enter Key 1: ")

key2 = input("Enter Key 2: ")

key3 = input("Enter Key 3: ")

key1 = hex2bin(key1)

key2 = hex2bin(key2)

key3 = hex2bin(key3)

key1 = permute(key1, keyp, 56)

key2 = permute(key2, keyp, 56)

key3 = permute(key3, keyp, 56)

print("Encryption:")

cipher\_text1 = bin2hex(encrypt(pt, key1))

cipher\_text2 = bin2hex(decrypt(cipher\_text1, key2))

cipher\_text = bin2hex(encrypt(cipher\_text2, key3))

print("Cipher Text obtained: ",cipher\_text)

print("Decryption:")

text1 = bin2hex(decrypt(cipher\_text, key3))

text2 = bin2hex(encrypt(text1, key2))

text = bin2hex(decrypt(text2, key1))

print("Plain Text: ",text)

**11. 3 DES**

ANOTHER METHOD

from Crypto.Cipher import DES3 >>> from Crypto.Random import get\_random\_bytes >>> >>> # Avoid Option 3 >>> while True: >>> try: >>> key = DES3.adjust\_key\_parity(get\_random\_bytes(24)) >>> break >>> except ValueError: >>> pass >>> >>> cipher = DES3.new(key, DES3.MODE\_CFB) >>> plaintext = b'We are no longer the knights who say ni!' >>> msg = cipher.iv + cipher.encrypt(plaintext)

**12 . ELGAMAL**

import random from math import pow a = random.randint(2, 10) def gcd(a, b): if a < b: return gcd(b, a) elif a % b == 0: return b; else: return gcd(b, a % b) def gen\_key(q): key = random.randint(pow(10, 20), q) while gcd(q, key) != 1: key = random.randint(pow(10, 20), q) return key def power(a, b, c): x = 1 y = a while b > 0: if b % 2 != 0: x = (x \* y) % c; y = (y \* y) % c b = int(b / 2) return x % c def encrypt(msg, q, h, g): en\_msg = [] k = gen\_key(q) s = power(h, k, q) p = power(g, k, q) for i in range(0, len(msg)): en\_msg.append(msg[i]) print("g^k used : ", p) print("g^ak used : ", s) for i in range(0, len(en\_msg)): en\_msg[i] = s \* ord(en\_msg[i]) return en\_msg, p def decrypt(en\_msg, p, key, q): dr\_msg = [] h = power(p, key, q) for i in range(0, len(en\_msg)): dr\_msg.append(chr(int(en\_msg[i]/h))) return dr\_msg def main(): msg = 'encryption' print("Original Message :", msg) q = random.randint(pow(10, 20), pow(10, 50)) g = random.randint(2, q) key = gen\_key(q) h = power(g, key, q) print("g used : ", g) print("g^a used : ", h) en\_msg, p = encrypt(msg, q, h, g) dr\_msg = decrypt(en\_msg, p, key, q) dmsg = ''.join(dr\_msg) print("Decrypted Message :", dmsg); if \_\_name\_\_ == '\_\_main\_\_': main()

**13. Diffie hellmann**

#include <cmath> #include <iostream> using namespace std; long long int power(long long int a, long long int b, long long int P) { if (b == 1) return a; else return (((long long int)pow(a, b)) % P); } int main() { long long int P, G, x, a, y, b, ka, kb; cout << "enter P value"<<endl; cin>>P; cout << "enter G value : "<< endl; cin>>G; cout << "enter private key a for Alice : " <<endl; cin>>a; x = power(G, a, P); cout << "enter private key b for Bob : " <<endl; cin>>b; y = power(G, b, P); ka = power(y, a, P); kb = power(x, b, P); cout << "Secret key for the Alice is : " << ka << endl; cout << "Secret key for the Alice is : " << kb << endl; return 0; }

**14.SHA1**

def ROTL(x, n, w):

return((x << n & (2 \*\* w - 1)) | (x >> w - n))

def Ch(x, y, z):

return((x & y) ^ (~x & z))

def Parity(x, y, z):

return(x ^ y ^ z)

def Maj(x, y, z):

return((x & y) ^ (x & z) ^ (y & z))

def sha1(x):

K = []

for t in range(80):

if t <= 19:

K.append(0x5a827999)

elif t <= 39:

K.append(0x6ed9eba1)

elif t <= 59:

K.append(0x8f1bbcdc)

else:

K.append(0xca62c1d6)

x\_bytes = bytearray(x, 'ascii')

x\_bits = [format(x, '08b') for x in x\_bytes]

print('x\_bits:', x\_bits)

x\_bits\_string = ''.join(x\_bits)

print('x\_bits\_string:', x\_bits\_string)

pad\_bits = '1' + ('0' \* (448 - (8 \* len(x) + 1))) + format(len(x) \* 8, '064b')

x\_padded = x\_bits\_string + pad\_bits

print('x\_padded:', x\_padded)

print('len(x\_padded):', len(x\_padded))

assert(len(x\_padded) == 512)

M1 = x\_padded

H = [0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0]

N = 1

for i in range(1, N + 1):

print('------' \* 2)

print('i = ', i)

W = list()

for t in range(80):

if t <= 15:

W.extend([ int(M1[ (32 \* t) : (32 \* (t + 1)) ], 2) ])

else:

W.extend([ ROTL( W[t - 3] ^ W[t - 8] ^ W[t - 14] ^ W[t - 16], n=1, w=32) ])

print('W:', W[0:16])

a = H[0]

b = H[1]

c = H[2]

d = H[3]

e = H[4]

print('hex(a):', hex(a))

print('hex(b):', hex(b))

print('hex(c):', hex(c))

print('hex(d):', hex(d))

print('hex(e):', hex(e))

for t in range(80):

print('------')

print('t =', t)

if t <= 19:

f = Ch

elif t <= 39:

f = Parity

elif t <= 59:

f = Maj

else:

f = Parity

T = (ROTL(a, n=5, w=32) + f(b, c, d) + e + K[t] + W[t]) % (2 \*\* 32)

e = d

d = c

c = ROTL(b, n=30, w=32)

b = a

a = T

print('hex(a):', hex(a))

print('hex(b):', hex(b))

print('hex(c):', hex(c))

print('hex(d):', hex(d))

print('hex(e):', hex(e))

H[0] = (a + H[0]) % (2 \*\* 32)

H[1] = (b + H[1]) % (2 \*\* 32)

H[2] = (c + H[2]) % (2 \*\* 32)

H[3] = (d + H[3]) % (2 \*\* 32)

H[4] = (e + H[4]) % (2 \*\* 32)

print(H)

H = [format(x, '08x') for x in H]

print('SHA-1 : ')

return("".join(H))

msg = input("Enter a input : ")

print(sha1(msg))

**14.kerberos**

import datetime, random

class xor\_cipher():

def \_\_init\_\_(self):

pass

def ascii2bin(self, string):

return ''.join('{:08b}'.format(ord(asc)) for asc in string)

def bin2ascii(self, binary):

return ''.join(chr(int(binary[i:i+8], 2)) for i in range(0, len(binary), 8))

def bin2hex(self, bn):

return ''.join('{:x}'.format(int(bn[i:i+4], 2)) for i in range(0, len(bn), 4))

def hex2bin(self, hx):

return ''.join('{:04b}'.format(int(h, 16)) for h in hx)

def xor(self, a, b):

return ''.join('0' if i == j else '1' for i, j in zip(a, b))

def encrypt(self, message, key):

key += key\*(len(message)-len(key))

message, key = self.ascii2bin(message), self.ascii2bin(key)

encrypted = self.xor(message, key)

return self.bin2hex(encrypted)

def decrypt(self, message, key):

key += key\*(len(message)-len(key))

message, key = self.hex2bin(message), self.ascii2bin(key)

decrypted = self.xor(message, key)

return self.bin2ascii(decrypted)

class user():

def \_\_init\_\_(self, name, authenticate\_shared\_key, database, nonce):

self.name = name

self.auth\_key = authenticate\_shared\_key

self.database = database

self.nonce = nonce

def prepare\_auth\_request(self):

return (self.name, self.database, self.nonce)

def process\_auth\_response(self, cipher1, cipher2):

response\_to\_user = eval(cipher.decrypt(cipher1, self.auth\_key))

self.user\_ticket\_key, nonce, time, ttl, dest = response\_to\_user

assert nonce == self.nonce

assert dest == 'ticket\_granting\_server'

self.ticket\_granting\_ticket = cipher2

return response\_to\_user

def prepare\_ticket\_request(self):

request = str((self.name, str(datetime.datetime.now().date()), self.database, self.nonce))

encrypted\_request = cipher.encrypt(request, self.user\_ticket\_key)

return (encrypted\_request, self.ticket\_granting\_ticket)

def process\_ticket\_response(self, cipher1, cipher2):

response\_to\_user = eval(cipher.decrypt(cipher1, self.user\_ticket\_key))

self.user\_database\_key, nonce, time, life, destination = response\_to\_user

self.database\_ticket = cipher2

assert self.nonce == nonce

assert self.database == destination

if self.database == destination:

num\_db = dataserver()

return num\_db

def prepare\_database\_request(self):

self.token = random.randint(1, 100)

request = str((self.name, self.token))

encrypted\_request = cipher.encrypt(request, self.user\_database\_key)

return (encrypted\_request, self.database\_ticket)

def process\_database\_response(self, response):

assert self.token+1 == response

def prepare\_database\_data\_request(self, database, request):

return cipher.encrypt(f'{database}.get\_data({request})', self.user\_database\_key)

class authentication\_server():

def \_\_init\_\_(self):

self.keys = {'tgs': 'tgs123'}

def set\_key(self, name, key):

self.keys[name] = key

def get\_key(self, name):

return self.keys[name]

def process\_request\_respond(self, request):

self.client\_name, self.destination, self.nonce = request

return (self.response\_for\_user(), self.response\_for\_TGS())

def response\_for\_user(self):

self.user\_ticket\_key = str(random.randint(1, 100))

response = (self.user\_ticket\_key, self.nonce, str(datetime.datetime.now().date()), '3 days', 'ticket\_granting\_server')

return cipher.encrypt(str(response), self.keys[self.client\_name])

def response\_for\_TGS(self):

response = (self.user\_ticket\_key, self.client\_name, '3 days')

return cipher.encrypt(str(response), self.keys['tgs'])

class ticket\_granting\_server():

def \_\_init\_\_(self):

self.personal\_key = 'tgs123'

self.keys = {'number\_database': 'alpha'}

def process\_auth\_user\_request\_respond(self, user\_request, auth\_response):

auth\_response = eval(cipher.decrypt(auth\_response, self.personal\_key))

self.user\_ticket\_key, client\_name, life = auth\_response

user\_request = eval(cipher.decrypt(user\_request, self.user\_ticket\_key))

self.user\_name, time, self.destination, self.nonce = user\_request

return (self.response\_for\_user(), self.response\_for\_database())

def response\_for\_user(self):

self.user\_database\_key = str(random.randint(1, 100))

response = str((self.user\_database\_key, self.nonce, str(datetime.datetime.now().date()), '5 days', self.destination))

return cipher.encrypt(response, self.user\_ticket\_key)

def response\_for\_database(self):

response = str((self.user\_database\_key, self.user\_name, '5 days'))

return cipher.encrypt(response, self.keys[self.destination])

class dataserver():

def \_\_init\_\_(self):

self.personal\_key = 'alpha'

self.data = {1: 'One', 2: 'Two', 3: 'Three'}

def \_\_str\_\_(self):

return 'number\_database'

def get\_data(self, index):

assert index in self.data.keys()

return self.data[index]

def process\_client\_request\_respond(self, client\_token, db\_ticket):

db\_ticket = eval(cipher.decrypt(db\_ticket, self.personal\_key))

self.user\_database\_key, user\_name, life = db\_ticket

client\_token = eval(cipher.decrypt(client\_token, self.user\_database\_key))

self.client\_name, self.token = client\_token

assert self.client\_name == user\_name

return self.token + 1

def process\_data\_request\_respond(self, request):

return eval(cipher.decrypt(request, self.user\_database\_key))

cipher = xor\_cipher()

db = 'number\_database'

username = 'ajay'

userkey = 'secret\_key'

user1 = user(username, userkey, db, 12)

auth\_server = authentication\_server()

auth\_server.set\_key(username, userkey)

ticket\_server = ticket\_granting\_server()

user\_request\_to\_auth = user1.prepare\_auth\_request()

response\_to\_user\_from\_auth, response\_to\_tgs\_from\_auth = auth\_server.process\_request\_respond(user\_request\_to\_auth)

auth\_response\_to\_user = user1.process\_auth\_response(response\_to\_user\_from\_auth, response\_to\_tgs\_from\_auth)

user\_request\_to\_tgs, ticket\_request\_from\_auth = user1.prepare\_ticket\_request()

response\_to\_user\_from\_tgs, response\_to\_db\_from\_tgs = ticket\_server.process\_auth\_user\_request\_respond(user\_request\_to\_tgs, ticket\_request\_from\_auth)

database1 = user1.process\_ticket\_response(response\_to\_user\_from\_tgs, response\_to\_db\_from\_tgs)

user\_request\_to\_db, db\_ticket\_from\_tgs = user1.prepare\_database\_request()

database\_response = database1.process\_client\_request\_respond(user\_request\_to\_db, db\_ticket\_from\_tgs)

user1.process\_database\_response(database\_response)

assert user1.user\_database\_key == database1.user\_database\_key

print('Key Establishment successful..!')

request\_data = 1

request = user1.prepare\_database\_data\_request('database1', request\_data)

response = database1.process\_data\_request\_respond(request)

print(request\_data, response)

**RSA DIGITAL SIGNATURE**

# Function to find gcd

# of two numbers

def euclid(m, n):

if n == 0:

return m

else:

r = m % n

return euclid(n, r)

# Program to find

# Multiplicative inverse

def exteuclid(a, b):

r1 = a

r2 = b

s1 = int(1)

s2 = int(0)

t1 = int(0)

t2 = int(1)

while r2 > 0:

q = r1//r2

r = r1-q \* r2

r1 = r2

r2 = r

s = s1-q \* s2

s1 = s2

s2 = s

t = t1-q \* t2

t1 = t2

t2 = t

if t1 < 0:

t1 = t1 % a

return (r1, t1)

# Enter two large prime

# numbers p and q

p = 823

q = 953

n = p \* q

Pn = (p-1)\*(q-1)

# Generate encryption key

# in range 1<e<Pn

key = []

for i in range(2, Pn):

gcd = euclid(Pn, i)

if gcd == 1:

key.append(i)

# Select an encryption key

# from the above list

e = int(313)

# Obtain inverse of

# encryption key in Z\_Pn

r, d = exteuclid(Pn, e)

if r == 1:

d = int(d)

print("decryption key is: ", d)

else:

print("Multiplicative inverse for\

the given encryption key does not \

exist. Choose a different encryption key ")

# Enter the message to be sent

M = 19070

# Signature is created by Alice

S = (M\*\*d) % n

# Alice sends M and S both to Bob

# Bob generates message M1 using the

# signature S, Alice's public key e

# and product n.

M1 = (S\*\*e) % n

# If M = M1 only then Bob accepts

# the message sent by Alice.

if M == M1:

print("As M = M1, Accept the\

message sent by Alice")

else:

print("As M not equal to M1,\

Do not accept the message\

sent by Alice ")

**MIDDLE MAN DH**

import random

# public keys are taken

# p is a prime number

# g is a primitive root of p

p = int(input('Enter a prime number : '))

g = int(input('Enter a number : '))

class A:

def \_\_init\_\_(self):

# Generating a random private number selected by alice

self.n = random.randint(1, p)

def publish(self):

# generating public values

return (g\*\*self.n)%p

def compute\_secret(self, gb):

# computing secret key

return (gb\*\*self.n)%p

class B:

def \_\_init\_\_(self):

# Generating a random private number selected for alice

self.a = random.randint(1, p)

# Generating a random private number selected for bob

self.b = random.randint(1, p)

self.arr = [self.a,self.b]

def publish(self, i):

# generating public values

return (g\*\*self.arr[i])%p

def compute\_secret(self, ga, i):

# computing secret key

return (ga\*\*self.arr[i])%p

alice = A()

bob = A()

eve = B()

# Printing out the private selected number by Alice and Bob

print(f'Alice selected (a) : {alice.n}')

print(f'Bob selected (b) : {bob.n}')

print(f'Eve selected private number for Alice (c) : {eve.a}')

print(f'Eve selected private number for Bob (d) : {eve.b}')

# Generating public values

ga = alice.publish()

gb = bob.publish()

gea = eve.publish(0)

geb = eve.publish(1)

print(f'Alice published (ga): {ga}')

print(f'Bob published (gb): {gb}')

print(f'Eve published value for Alice (gc): {gea}')

print(f'Eve published value for Bob (gd): {geb}')

# Computing the secret key

sa = alice.compute\_secret(gea)

sea = eve.compute\_secret(ga,0)

sb = bob.compute\_secret(geb)

seb = eve.compute\_secret(gb,1)

print(f'Alice computed (S1) : {sa}')

print(f'Eve computed key for Alice (S1) : {sea}')

print(f'Bob computed (S2) : {sb}')

print(f'Eve computed key for Bob (S2) : {seb}')

**RSA**

#include<iostream>

#include<math.h>

using namespace std;

int gcd(int a, int b) {

int t;

while(1) {

t= a%b;

if(t==0)

return b;

a = b;

b= t;

}

}

int main() {

double p = 17;

double q = 19;

double n=p\*q;

double track;

double phi= (p-1)\*(q-1);

double e=7;

while(e<phi) {

track = gcd(e,phi);

if(track==1)

break;

else

e++;

}

double d1=1/e;

double d=fmod(d1,phi);

double message = 10;

double c = pow(message,e);

double m = pow(c,d);

c=fmod(c,n);

m=fmod(m,n);

cout<<"Original Message = "<<message;

cout<<"p = "<<p<<endl;

cout<<"q = "<<q<<endl;

cout<<"n = pq = "<<n<<endl;

cout<<"phi = "<<phi<<endl;

cout<<"e = "<<e<<endl;

cout<<"d = "<<d<<endl;

cout<<"Encrypted message = "<<c<<endl;

cout<<"Decrypted message = "<<m<<endl;

return 0;

}

**ECDSA**

# Python program to implement

# ECDSA

p = pow(2, 255) - 19

base = 15112221349535400772501151409588531511454012693041857206046113283949847762202, 46316835694926478169428394003475163141307993866256225615783033603165251855960

# Function for finding positive modulus

# of the number

def findPositiveModulus(a, p):

if a < 0:

a = (a + p \* int(abs(a)/p) + p) % p

return a

# Function for typecasting from

# string to int

def textToInt(text):

encoded\_text = text.encode('utf-8')

hex\_text = encoded\_text.hex()

int\_text = int(hex\_text, 16)

return int\_text

# Function to find greatest

# common divisor(gcd) of a and b

def gcd(a, b):

while a != 0:

a, b = b % a, a

return b

# Function to find the modular inverse

# of a mod m

def findModInverse(a, m):

if a < 0:

a = (a + m \* int(abs(a)/m) + m) % m

# no mod inverse if a & m aren't

# relatively prime

if gcd(a, m) != 1:

return None

# Calculate using the Extended

# Euclidean Algorithm:

u1, u2, u3 = 1, 0, a

v1, v2, v3 = 0, 1, m

while v3 != 0:

# // is the integer division operator

q = u3 // v3

v1, v2, v3, u1, u2, u3 = (u1 - q \* v1), (u2 - q \* v2), (u3 - q \* v3), v1, v2, v3

return u1 % m

def applyDoubleAndAddMethod(P, k, a, d, mod):

additionPoint = (P[0], P[1])

# 0b1111111001

kAsBinary = bin(k)

# 1111111001

kAsBinary = kAsBinary[2:len(kAsBinary)]

# print(kAsBinary)

for i in range(1, len(kAsBinary)):

currentBit = kAsBinary[i: i+1]

# always apply doubling

additionPoint = pointAddition(additionPoint, additionPoint, a, d, mod)

if currentBit == '1':

# add base point

additionPoint = pointAddition(additionPoint, P, a, d, mod)

return additionPoint

# Function to calculate the point addition

def pointAddition(P, Q, a, d, mod):

x1 = P[0]; y1 = P[1]

x2 = Q[0]; y2 = Q[1]

x3 = (((x1\*y2 + y1\*x2) % mod) \* findModInverse(1+d\*x1\*x2\*y1\*y2, mod)) % mod

y3 = (((y1\*y2 - a\*x1\*x2) % mod) \* findModInverse(1- d\*x1\*x2\*y1\*y2, mod)) % mod

return x3, y3

# ax^2 + y^2 = 1 + dx^2y^2

# ed25519

a = -1; d = findPositiveModulus(-121665 \* findModInverse(121666, p), p)

# print("curve: ",a,"x^2 + y^2 = 1 + ",d,"x^2 y^2")

x0 = base[0]; y0 = base[1]

print("----------------------")

print("Key Generation: ")

# privateKey = 47379675103498394144858916095175689

# 779086087640336534911165206022228115974270 #32 byte secret key

import random

privateKey = random.getrandbits(256) #32 byte secret key

# print("private key: ",privateKey)

publicKey = applyDoubleAndAddMethod(base, privateKey, a, d, p)

print("public key: ", publicKey)

message = textToInt("Hello, world!")

# Function for hashing the message

def hashing(message):

import hashlib

return int(hashlib.sha512(str(message).encode("utf-8")).hexdigest(), 16)

# ---------------------------------------

# sign

r = hashing(hashing(message) + message) % p

R = applyDoubleAndAddMethod(base, r, a, d, p)

h = hashing(R[0] + publicKey[0] + message) % p

# % p

s = (r + h \* privateKey)

print("----------------------")

print("Signing:")

print("message: ",message)

print("Signature (R, s)")

print("R: ",R)

print("s: ",s)

# -----------------------------------

# verify

h = hashing(R[0] + publicKey[0] + message) % p

P1 = applyDoubleAndAddMethod(base, s, a, d, p)

P2 = pointAddition(R, applyDoubleAndAddMethod(publicKey, h, a, d, p), a, d, p)

print("----------------------")

print("Verification:")

print("P1: ",P1)

print("P2: ",P2)

print("----------------------")

print("result")

if P1[0] == P2[0] and P1[1] == P2[1]:

print("The Signature is valid")

else:

print("The Signature violation detected!")

# ----------------------------------