**CODES**

1. **AES**

import java.util.\*;

import java.io.UnsupportedEncodingException;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.Arrays;

import java.util.Base64;

import java.util.\*;

import javax.crypto.Cipher;

import javax.crypto.spec.SecretKeySpec;

public class AES {

private static SecretKeySpec secretKey;

private static byte[] key;

public static void setKey(String myKey)

{

MessageDigest sha = null;

try {

key = myKey.getBytes("UTF-8");

sha = MessageDigest.getInstance("SHA-1");

key = sha.digest(key);

key = Arrays.copyOf(key, 16);

secretKey = new SecretKeySpec(key, "AES");

}

catch (NoSuchAlgorithmException e) {

e.printStackTrace();

}

catch (UnsupportedEncodingException e) {

e.printStackTrace();

}

}

public static String encrypt(String strToEncrypt, String secret)

{

try

{

setKey(secret);

Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

cipher.init(Cipher.ENCRYPT\_MODE, secretKey);

return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes("UTF-8")));

}

catch (Exception e)

{

System.out.println("Error while encrypting: " + e.toString());

}

return null;

}

public static String decrypt(String strToDecrypt, String secret)

{

try

{

setKey(secret);

Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");

cipher.init(Cipher.DECRYPT\_MODE, secretKey);

return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));

}

catch (Exception e)

{

System.out.println("Error while decrypting: " + e.toString());

}

return null;

}

public static void main(String[] args)

{

final String secretKey = "ssshhhhhhhhhhh!!!!";

Scanner sc= new Scanner(System.in);

System.out.print("Enter a string: ");

String originalString= sc.nextLine();

String encryptedString = AES.encrypt(originalString, secretKey) ;

String decryptedString = AES.decrypt(encryptedString, secretKey) ;

System.out.println(originalString);

System.out.println(encryptedString);

System.out.println(decryptedString);

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime+" millisecond");

} }

1. **DES**

import javax.swing.\*;

import java.security.SecureRandom;

import javax.crypto.Cipher;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.crypto.spec.SecretKeySpec;

import java.util.Random ;

class DES {

byte[] skey = new byte[1000];

String skeyString;

static byte[] raw;

String inputMessage,encryptedData,decryptedMessage;

public DES() {

try {

generateSymmetricKey();

inputMessage=JOptionPane.showInputDialog(null,"Enter message to encrypt");

byte[] ibyte = inputMessage.getBytes();

byte[] ebyte=encrypt(raw, ibyte);

String encryptedData = new String(ebyte);

System.out.println("Encrypted message "+encryptedData);

JOptionPane.showMessageDialog(null,"Encrypted Data "+"\n"+encryptedData);

byte[] dbyte= decrypt(raw,ebyte);

String decryptedMessage = new String(dbyte);

System.out.println("Decrypted message "+decryptedMessage);

JOptionPane.showMessageDialog(null,"Decrypted Data "+"\n"+decryptedMessage);

}

catch(Exception e) {

System.out.println(e);

}

}

void generateSymmetricKey() {

try {

Random r = new Random();

int num = r.nextInt(10000);

String knum = String.valueOf(num);

byte[] knumb = knum.getBytes();

skey=getRawKey(knumb);

skeyString = new String(skey);

System.out.println("DES Symmetric key = "+skeyString);

}

catch(Exception e) {

System.out.println(e);

}

}

private static byte[] getRawKey(byte[] seed) throws Exception {

KeyGenerator kgen = KeyGenerator.getInstance("DES");

SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");

sr.setSeed(seed);

kgen.init(56, sr);

SecretKey skey = kgen.generateKey();

raw = skey.getEncoded();

return raw;

}

private static byte[] encrypt(byte[] raw, byte[] clear) throws Exception {

SecretKeySpec skeySpec = new SecretKeySpec(raw, "DES");

Cipher cipher = Cipher.getInstance("DES");

cipher.init(Cipher.ENCRYPT\_MODE, skeySpec);

byte[] encrypted = cipher.doFinal(clear);

return encrypted;

}

private static byte[] decrypt(byte[] raw, byte[] encrypted) throws Exception {

SecretKeySpec skeySpec = new SecretKeySpec(raw, "DES");

Cipher cipher = Cipher.getInstance("DES");

cipher.init(Cipher.DECRYPT\_MODE, skeySpec);

byte[] decrypted = cipher.doFinal(encrypted);

return decrypted;

}

public static void main(String args[]) {

DES des = new DES();

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime +" millisecond");

}

}

1. **RSA**

import java.math.\*;

import java.util.\*;

class RSA

{

public static void main(String args[])

{

Scanner sc=new Scanner(System.in);

int p,q,n,z,d=0,e,i;

System.out.println("Enter the number to be encrypted and decrypted");

int msg=sc.nextInt();

double c;

BigInteger msgback;

System.out.println("Enter 1st prime number p");

p=sc.nextInt();

System.out.println("Enter 2nd prime number q");

q=sc.nextInt();

n=p\*q;

z=(p-1)\*(q-1);

System.out.println("the value of z = "+z);

for(e=2;e<z;e++)

{

if(gcd(e,z)==1) // e is for public key exponent

{

break;

}

}

System.out.println("the value of e = "+e);

for(i=0;i<=9;i++)

{

int x=1+(i\*z);

if(x%e==0) //d is for private key exponent

{

d=x/e;

break;

}

}

System.out.println("the value of d = "+d);

c=(Math.pow(msg,e))%n;

System.out.println("Encrypted message is : -");

System.out.println(c);

//converting int value of n to BigInteger

BigInteger N = BigInteger.valueOf(n);

//converting float value of c to BigInteger

BigInteger C = BigDecimal.valueOf(c).toBigInteger();

msgback = (C.pow(d)).mod(N);

System.out.println("Derypted message is : -");

System.out.println(msgback);

long startTime = System.currentTimeMillis();

long total = 0;

for ( i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime +" millisecond");

}

static int gcd(int e, int z)

{

if(e==0)

return z;

else

return gcd(z%e,e);

}

}

1. **Diffie Hellman**

import java.util.\*;

class Diffie\_Hellman

{

public static void main(String args[])

{

Scanner sc=new Scanner(System.in);

System.out.println("Enter modulo(p)");

int p=sc.nextInt();

System.out.println("Enter primitive root of "+p);

int g=sc.nextInt();

System.out.println("Choose 1st secret no(Alice)");

int a=sc.nextInt();

System.out.println("Choose 2nd secret no(BOB)");

int b=sc.nextInt();

int A = (int)Math.pow(g,a)%p;

int B = (int)Math.pow(g,b)%p;

int S\_A = (int)Math.pow(B,a)%p;

int S\_B =(int)Math.pow(A,b)%p;

if(S\_A==S\_B)

{

System.out.println("ALice and Bob can communicate with each other!!!");

System.out.println("They share a secret no = "+S\_A);

}

else

{

System.out.println("ALice and Bob cannot communicate with each other!!!");

}

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime +" millisecond");

}

}

1. **Blowfish**

import javax.swing.\*;

import java.security.SecureRandom;

import javax.crypto.Cipher;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.crypto.spec.SecretKeySpec;

import java.util.Random ;

class Blowfish {

byte[] skey = new byte[1000];

String skeyString;

static byte[] raw;

String inputMessage,encryptedData,decryptedMessage;

public Blowfish() {

try {

generateSymmetricKey();

inputMessage=JOptionPane.showInputDialog(null,"Enter message to encrypt");

byte[] ibyte = inputMessage.getBytes();

byte[] ebyte=encrypt(raw, ibyte);

String encryptedData = new String(ebyte);

System.out.println("Encrypted message "+encryptedData);

JOptionPane.showMessageDialog(null,"Encrypted Data "+"\n"+encryptedData);

byte[] dbyte= decrypt(raw,ebyte);

String decryptedMessage = new String(dbyte);

System.out.println("Decrypted message "+decryptedMessage);

JOptionPane.showMessageDialog(null,"Decrypted Data "+"\n"+decryptedMessage);

}

catch(Exception e) {

System.out.println(e);

}

}

void generateSymmetricKey() {

try {

Random r = new Random();

int num = r.nextInt(10000);

String knum = String.valueOf(num);

byte[] knumb = knum.getBytes();

skey=getRawKey(knumb);

skeyString = new String(skey);

System.out.println("Blowfish Symmetric key = "+skeyString);

}

catch(Exception e) {

System.out.println(e);

}

}

private static byte[] getRawKey(byte[] seed) throws Exception {

KeyGenerator kgen = KeyGenerator.getInstance("Blowfish");

SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");

sr.setSeed(seed);

kgen.init(128, sr); // 128, 256 and 448 bits may not be available

SecretKey skey = kgen.generateKey();

raw = skey.getEncoded();

return raw;

}

private static byte[] encrypt(byte[] raw, byte[] clear) throws Exception {

SecretKeySpec skeySpec = new SecretKeySpec(raw, "Blowfish");

Cipher cipher = Cipher.getInstance("Blowfish");

cipher.init(Cipher.ENCRYPT\_MODE, skeySpec);

byte[] encrypted = cipher.doFinal(clear);

return encrypted;

}

private static byte[] decrypt(byte[] raw, byte[] encrypted) throws Exception {

SecretKeySpec skeySpec = new SecretKeySpec(raw, "Blowfish");

Cipher cipher = Cipher.getInstance("Blowfish");

cipher.init(Cipher.DECRYPT\_MODE, skeySpec);

byte[] decrypted = cipher.doFinal(encrypted);

return decrypted;

}

public static void main(String args[]) {

Blowfish bf = new Blowfish();

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime+" millisecond");

}

}

1. **Hybrid Algorithm 1**

import java.util.\*;

import java.lang.\*;

class DAA\_Hybrid {

public static void main(String[] args)

{

String pass = "DAA Assignment";

String key = "Algorithm";

System.out.println(encryptText(pass, key));

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime+" millisecond");

}

public static String encryptText(String password, String key)

{

int a = 0, b = 1, c = 0, m = 0, k = 0, j = 0;

String cipher = "", temp = "";

// Declare a password string

StringBuffer pw = new StringBuffer(password);

// Reverse the String

pw = pw.reverse();

pw = pw.append(key);

// For future Purpose

temp = pw.toString();

char stringArray[] = temp.toCharArray();

String evenString = "", oddString = "";

// Declare EvenArray for storing

// even index of stringArray

char evenArray[];

// Declare OddArray for storing

// odd index of stringArray

char oddArray[];

// Storing the positions in their respective arrays

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

if (i % 2 == 0) {

oddString = oddString + Character.toString(stringArray[i]);

}

else {

evenString = evenString + Character.toString(stringArray[i]);

}

}

evenArray = new char[evenString.length()];

oddArray = new char[oddString.length()];

// Generate a Fibonacci Series

// Upto the Key Length

while (m <= key.length()) {

// As it always starts with 1

if (m == 0)

m = 1;

else {

// Logic For Fibonacci Series

a = b;

b = c;

c = a + b;

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

// Caesar Cipher Algorithm Start for even positions

int p = evenString.charAt(i);

int cip = 0;

if (p == '0' || p == '1' || p == '2' || p == '3' || p == '4'

|| p == '5' || p == '6'

|| p == '7' || p == '8' || p == '9') {

cip = p - c;

if (cip < '0')

cip = cip + 9;

}

else {

cip = p - c;

if (cip < 'a') {

cip = cip + 26;

}

}

evenArray[i] = (char)cip;

/\* Caesar Cipher Algorithm End\*/

}

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

// Caesar Cipher Algorithm Start for odd positions

int p = oddString.charAt(i);

int cip = 0;

if (p == '0' || p == '1' || p == '2' || p == '3' || p == '4'

|| p == '5' || p == '6'

|| p == '7' || p == '8' || p == '9') {

cip = p + c;

if (cip > '9')

cip = cip - 9;

}

else {

cip = p + c;

if (cip > 'z') {

cip = cip - 26;

}

}

oddArray[i] = (char)cip;

// Caesar Cipher Algorithm End

}

m++;

}

}

// Storing content of even and

// odd array to the string array

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

if (i % 2 == 0) {

stringArray[i] = oddArray[k];

k++;

}

else {

stringArray[i] = evenArray[j];

j++;

}

}

// Generating a Cipher Text

// by stringArray (Caesar Cipher)

for (char d : stringArray) {

cipher = cipher + d;

}

// Return the Cipher Text

return cipher;

}

}

1. **Hybrid Algorithm 2**

import java.math.BigInteger;

import java.security.InvalidKeyException;

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.NoSuchAlgorithmException;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.util.Base64;

import javax.crypto.BadPaddingException;

import javax.crypto.Cipher;

import javax.crypto.IllegalBlockSizeException;

import javax.crypto.KeyGenerator;

import javax.crypto.NoSuchPaddingException;

import javax.crypto.SecretKey;

import javax.crypto.spec.SecretKeySpec;

public class EncDeHybrid

{

// Symmetric encryption algorithms supported - AES, RC4, DES

// encryption algorithm - DES, key size - 56

protected static String DEFAULT\_ENCRYPTION\_ALGORITHM = "AES";

protected static int DEFAULT\_ENCRYPTION\_KEY\_LENGTH = 256;

// key encryption algorithms supported - RSA, Diffie-Hellman, DSA

// key pair generator - RSA: keyword - RSA, key size: 1024, 2048

// key pair generator - Diffie-Hellman: keyword i DiffieHellman, key size - 1024

// key pair generator - DSA: keyword - DSA, key size: 1024

// NOTE: using asymmetric algorithms other than RSA needs to be worked out

protected static String DEFAULT\_KEY\_ENCRYPTION\_ALGORITHM = "RSA";

protected static int DEFAULT\_KEY\_ENCRYPTION\_KEY\_LENGTH = 1024;

protected static String DEFAULT\_TRANSFORMATION = "RSA/ECB/PKCS1Padding";

protected SecretKey mSecretKey;

protected String mEncryptionAlgorithm, mKeyEncryptionAlgorithm, mTransformation;

protected int mEncryptionKeyLength, mKeyEncryptionKeyLength;

protected PublicKey mPublicKey;

protected PrivateKey mPrivateKey;

EncDeHybrid()

{

mSecretKey = null;

mEncryptionAlgorithm = EncDeHybrid.DEFAULT\_ENCRYPTION\_ALGORITHM;

mEncryptionKeyLength = EncDeHybrid.DEFAULT\_ENCRYPTION\_KEY\_LENGTH;

mKeyEncryptionAlgorithm = EncDeHybrid.DEFAULT\_KEY\_ENCRYPTION\_ALGORITHM;

mKeyEncryptionKeyLength = EncDeHybrid.DEFAULT\_KEY\_ENCRYPTION\_KEY\_LENGTH;

mPublicKey = null;

mPrivateKey = null;

mTransformation = EncDeHybrid.DEFAULT\_TRANSFORMATION;

}

EncDeHybrid(String encAlgo, int encKeyLength, String keyEncAlgo, int keyEncKeyLength, String transformation)

{

mSecretKey = null;

mEncryptionAlgorithm = encAlgo;

mEncryptionKeyLength = encKeyLength;

mKeyEncryptionAlgorithm = keyEncAlgo;

mKeyEncryptionKeyLength = keyEncKeyLength;

mTransformation = transformation;

}

public static BigInteger keyToNumber(byte[] byteArray)

{

return new BigInteger(1, byteArray);

}

public SecretKey getSecretKey()

{

return mSecretKey;

}

public byte[] getSecretKeyAsByteArray()

{

return mSecretKey.getEncoded();

}

// get base64 encoded version of the key

public String getEncodedSecretKey()

{

String encodedKey = Base64.getEncoder().encodeToString(mSecretKey.getEncoded());

return encodedKey;

}

// decode the base64 encoded string

public SecretKey getDecodedSecretKey(String encodedKey, String algo)

{

byte[] decodedKey = Base64.getDecoder().decode(encodedKey);

// rebuild key using SecretKeySpec

SecretKey originalKey = null;

if ( null == algo ) {

originalKey = new SecretKeySpec(decodedKey, 0, decodedKey.length, mEncryptionAlgorithm);

} else {

originalKey = new SecretKeySpec(decodedKey, 0, decodedKey.length, algo);

}

return originalKey;

}

public PublicKey getPublicKey()

{

return mPublicKey;

}

public byte[] getPublicKeyAsByteArray()

{

return mPublicKey.getEncoded();

}

public String getEncodedPublicKey()

{

String encodedKey = Base64.getEncoder().encodeToString(mPublicKey.getEncoded());

return encodedKey;

}

public PrivateKey getPrivateKey()

{

return mPrivateKey;

}

public byte[] getPrivateKeyAsByteArray()

{

return mPrivateKey.getEncoded();

}

public String getEncodedPrivateKey()

{

String encodedKey = Base64.getEncoder().encodeToString(mPrivateKey.getEncoded());

return encodedKey;

}

// step 1 -- generate the symmetric key

public void generateSymmetricKey()

{

KeyGenerator generator;

try {

generator = KeyGenerator.getInstance(mEncryptionAlgorithm);

generator.init(mEncryptionKeyLength);

mSecretKey = generator.generateKey();

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

}

}

// step 2 -- encrypt the plain text

public byte[] encryptText(String textToEncrypt)

{

byte[] byteCipherText = null;

try {

Cipher encCipher = Cipher.getInstance(mEncryptionAlgorithm);

encCipher.init(Cipher.ENCRYPT\_MODE, mSecretKey);

byteCipherText = encCipher.doFinal(textToEncrypt.getBytes());

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (NoSuchPaddingException e) {

e.printStackTrace();

} catch (InvalidKeyException e) {

e.printStackTrace();

} catch (IllegalBlockSizeException e) {

e.printStackTrace();

} catch (BadPaddingException e) {

e.printStackTrace();

}

return byteCipherText;

}

// step 3 -- encrypt the secret key using key encryption algorithm

public byte[] encryptSecretKey()

{

byte[] encryptedKey = null;

try {

KeyPairGenerator kpg = KeyPairGenerator.getInstance(mKeyEncryptionAlgorithm);

kpg.initialize(mKeyEncryptionKeyLength);

KeyPair keyPair = kpg.generateKeyPair();

mPublicKey = keyPair.getPublic();

mPrivateKey = keyPair.getPrivate();

Cipher cipher = Cipher.getInstance(mTransformation);

cipher.init(Cipher.PUBLIC\_KEY, mPublicKey);

encryptedKey = cipher.doFinal(mSecretKey.getEncoded());

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (NoSuchPaddingException e) {

e.printStackTrace();

} catch (InvalidKeyException e) {

e.printStackTrace();

} catch (IllegalBlockSizeException e) {

e.printStackTrace();

} catch (BadPaddingException e) {

e.printStackTrace();

}

return encryptedKey;

}

public byte[] encryptSecretKeyNoTransformation()

{

byte[] encryptedKey = null;

try {

KeyPairGenerator kpg = KeyPairGenerator.getInstance(mKeyEncryptionAlgorithm);

kpg.initialize(mKeyEncryptionKeyLength);

KeyPair keyPair = kpg.generateKeyPair();

mPublicKey = keyPair.getPublic();

mPrivateKey = keyPair.getPrivate();

Cipher cipher = Cipher.getInstance(mTransformation);

cipher.init(Cipher.PUBLIC\_KEY, mPublicKey);

encryptedKey = cipher.doFinal(mSecretKey.getEncoded());

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (NoSuchPaddingException e) {

e.printStackTrace();

} catch (InvalidKeyException e) {

e.printStackTrace();

} catch (IllegalBlockSizeException e) {

e.printStackTrace();

} catch (BadPaddingException e) {

e.printStackTrace();

}

return encryptedKey;

}

// step 4 -- send across the encrypted text and the encrypted secret key

// setp 5 -- decrypt secret key

public byte[] decryptSecretKey(byte[] encryptedSecretKey)

{

byte[] decryptedKey = null;

try {

Cipher cipher = Cipher.getInstance(mTransformation);

cipher.init(Cipher.PRIVATE\_KEY, mPrivateKey);

decryptedKey = cipher.doFinal(encryptedSecretKey);

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (NoSuchPaddingException e) {

e.printStackTrace();

} catch (InvalidKeyException e) {

e.printStackTrace();

} catch (IllegalBlockSizeException e) {

e.printStackTrace();

} catch (BadPaddingException e) {

e.printStackTrace();

}

return decryptedKey;

}

// step 6 -- Decrypt the cipher using decrypted symmetric key

public String decryptText(byte[] decryptedKey, byte[] encryptedText)

{

String decryptedPlainText = null;

try {

SecretKey originalKey = new SecretKeySpec(decryptedKey , 0, decryptedKey.length, mEncryptionAlgorithm);

Cipher aesCipher2 = Cipher.getInstance(mEncryptionAlgorithm);

aesCipher2.init(Cipher.DECRYPT\_MODE, originalKey);

byte[] bytePlainText = aesCipher2.doFinal(encryptedText);

decryptedPlainText = new String(bytePlainText);

} catch (NoSuchAlgorithmException e) {

e.printStackTrace();

} catch (NoSuchPaddingException e) {

e.printStackTrace();

} catch (InvalidKeyException e) {

e.printStackTrace();

} catch (IllegalBlockSizeException e) {

e.printStackTrace();

} catch (BadPaddingException e) {

e.printStackTrace();

}

return decryptedPlainText;

}

}

import javax.crypto.SecretKey;

public class DAA

{

public static void main(String[] args)

{

String plainText = "This is our DAA Assignment";

System.out.println("plainText: '" + plainText + "'");

EncDeHybrid ed = new EncDeHybrid();

ed.generateSymmetricKey();

System.out.println("secret key: '" + EncDeHybrid.keyToNumber(ed.getSecretKey().getEncoded()).toString() + "'" );

String strEncodedSecretKey = ed.getEncodedSecretKey();

System.out.println("encoded secret key: '" + strEncodedSecretKey + "'" );

SecretKey sk = ed.getDecodedSecretKey(strEncodedSecretKey, null);

System.out.println("decoded secret key: '" + EncDeHybrid.keyToNumber(sk.getEncoded()).toString() + "'" );

byte[] encryptedSecretKey = ed.encryptSecretKey();

System.out.println("encrypted secret key: '" + EncDeHybrid.keyToNumber(encryptedSecretKey).toString() + "'" );

byte[] encryptedText = ed.encryptText(plainText);

System.out.println("encrypted text: '" + EncDeHybrid.keyToNumber(encryptedText).toString() + "'" );

byte[] decryptedSecretKey = ed.decryptSecretKey(encryptedSecretKey);

System.out.println("decrypted secret key: '" + EncDeHybrid.keyToNumber(decryptedSecretKey).toString() + "'" );

String decryptedText = ed.decryptText(decryptedSecretKey, encryptedText);

System.out.println("decryptedText: '" + decryptedText + "'");

long startTime = System.currentTimeMillis();

long total = 0;

for (int i = 0; i < 10000000; i++) {

total += i;

}

long stopTime = System.currentTimeMillis();

long elapsedTime = stopTime - startTime;

System.out.println(elapsedTime+" millisecond");

}

}