

Caesar Cipher

KEY = 3

```
def encrypt(text, key):  
    ciphertext = ""  
    for char in text:  
        if char.isalpha():  
            base = ord('A') if char.isupper() else ord('a')  
            ciphertext += chr((ord(char) - base + key) % 26 + base)  
        else:  
            ciphertext += char  
    return ciphertext
```

```
def decrypt(text, key):  
    plaintext = ""  
    for char in text:  
        if char.isalpha():  
            base = ord('A') if char.isupper() else ord('a')  
            plaintext += chr((ord(char) - base - key) % 26 + base)  
        else:  
            plaintext += char  
    return plaintext
```

Main

```
plaintext = input("Enter the text to encrypt: ")  
ciphertext = encrypt(plaintext, KEY)  
print("\nEncrypted Text:", ciphertext)
```

```
show = input("Do you want to decrypt the text? (yes/no): ").strip().lower()  
if show == "yes":
```

```
print("Decrypted Text:", decrypt(ciphertext, KEY))

else:

    print("Decryption skipped.")
```

monoalphabetic substitution cipher

```
PLAIN_ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

CIPHER_ALPHABET = 'QWERTYUIOPASDFGHJKLZCVBNM'

encrypt_map = {PLAIN_ALPHABET[i]: CIPHER_ALPHABET[i] for i in range(26)}

decrypt_map = {CIPHER_ALPHABET[i]: PLAIN_ALPHABET[i] for i in range(26)}
```

```
plaintext = input("Enter the text to encrypt: ").upper()

ciphertext = "".join(encrypt_map.get(char, char) for char in plaintext)

print("\nEncrypted Text:", ciphertext)
```

```
show = input("Do you want to decrypt the text? (yes/no): ").strip().lower()

if show == "yes":

    decrypted_text = "".join(decrypt_map.get(char, char) for char in ciphertext)

    print("Decrypted Text:", decrypted_text)

else:

    print("Decryption skipped.")
```

Rail Fence Cipher

```
def rail_fence_encrypt(text, rails):
    rails_list = [""] * rails
    row = 0
    direction = 1 # +1 = down, -1 = up

    for char in text:
        rails_list[row] += char
        row += direction

        if row == 0 or row == rails - 1:
            direction *= -1

    return "".join(rails_list)

def rail_fence_decrypt(ciphertext, rails):
    length = len(ciphertext)

    pattern = [["" for _ in range(length)] for _ in range(rails)]
    row, direction = 0, 1

    for col in range(length):
        pattern[row][col] = '*'
        row += direction

        if row == 0 or row == rails - 1:
            direction *= -1

    index = 0
    for r in range(rails):
```

```
for c in range(length):
    if pattern[r][c] == '*' and index < length:
        pattern[r][c] = ciphertext[index]
        index += 1

result = []
row, direction = 0, 1
for col in range(length):
    result.append(pattern[row][col])
    row += direction
    if row == 0 or row == rails - 1:
        direction *= -1

return "" .join(result)
```

```
RAILS = 3
plaintext = input("Enter the text to encrypt: ")
ciphertext = rail_fence_encrypt(plaintext, RAILS)
print("\nEncrypted Text:", ciphertext)

show_decrypt = input("Do you want to decrypt the text? (yes/no): ").strip().lower()
if show_decrypt == "yes":
    decrypted_text = rail_fence_decrypt(ciphertext, RAILS)
    print("Decrypted Text:", decrypted_text)
else:
    print("Decryption skipped.")
```

PLAYFAIR

```
KEYWORD = "PLAYFAIR"
```

```
def generate_matrix(keyword):  
    keyword = keyword.upper().replace("J", "I")  
    seen = set()  
    matrix_list = []
```

```
    for char in keyword:  
        if char.isalpha() and char not in seen:  
            matrix_list.append(char)  
            seen.add(char)  
  
    for char in "ABCDEFGHIJKLMNPQRSTUVWXYZ":  
        if char not in seen:  
            matrix_list.append(char)  
            seen.add(char)  
  
    return [matrix_list[i*5:(i+1)*5] for i in range(5)]
```

```
def find_pos(ch, matrix):  
    for i in range(5):  
        if ch in matrix[i]:  
            return i, matrix[i].index(ch)  
  
    return None
```

```
def prepare_text(text):  
    text = text.upper().replace("J", "I")  
    result = ""
```

```
i = 0

while i < len(text):
    if not text[i].isalpha():
        i += 1
        continue

    a = text[i]
    b = text[i+1] if i+1 < len(text) and text[i+1].isalpha() else 'X'

    if a == b:
        b = 'X'
        i += 1
    else:
        i += 2

    result += a + b

return result

def playfair_encrypt(plaintext, matrix):
    plaintext = prepare_text(plaintext)
    cipher = ""

    for i in range(0, len(plaintext), 2):
        a, b = plaintext[i], plaintext[i+1]
        r1, c1 = find_pos(a, matrix)
        r2, c2 = find_pos(b, matrix)

        if r1 == r2:
            cipher += matrix[r1][c1] + matrix[r1][c2]
        elif c1 == c2:
            cipher += matrix[r1][c1] + matrix[r2][c1]
        else:
            cipher += matrix[r1][c2] + matrix[r2][c1]

    return cipher
```

```
    cipher += matrix[r1][(c1 + 1) % 5]
    cipher += matrix[r2][(c2 + 1) % 5]

elif c1 == c2:
    cipher += matrix[(r1 + 1) % 5][c1]
    cipher += matrix[(r2 + 1) % 5][c2]

else:
    cipher += matrix[r1][c2]
    cipher += matrix[r2][c1]

return cipher
```

```
def playfair_decrypt(ciphertext, matrix):
    plain = ""

    for i in range(0, len(ciphertext), 2):
        a, b = ciphertext[i], ciphertext[i+1]
        r1, c1 = find_pos(a, matrix)
        r2, c2 = find_pos(b, matrix)

        if r1 == r2:
            plain += matrix[r1][(c1 - 1) % 5]
            plain += matrix[r2][(c2 - 1) % 5]

        elif c1 == c2:
            plain += matrix[(r1 - 1) % 5][c1]
            plain += matrix[(r2 - 1) % 5][c2]

        else:
```

```
plain += matrix[r1][c2]

plain += matrix[r2][c1]

return plain

key_matrix = generate_matrix(KEYWORD)

plaintext = input("Enter the text to encrypt: ")

ciphertext = playfair_encrypt(plaintext, key_matrix)
print("\nEncrypted Text:", ciphertext)

if input("Do you want to decrypt the text? (yes/no): ").lower() == "yes":
    print("Decrypted Text:", playfair_decrypt(ciphertext, key_matrix))
else:
    print("Decryption skipped.")
```

HILL CIPHER

```
import numpy as np

MOD = 26

LETTER_TO_NUM = {chr(i + 65): i for i in range(26)}
NUM_TO_LETTER = {i: chr(i + 65) for i in range(26)}

KEY_MATRIX = np.array([[3, 3],
                      [2, 5]])

def mod_inverse(a, m):
    for i in range(1, m):
        if (a * i) % m == 1:
            return i
    raise ValueError("No modular inverse found!")

def preprocess_text(text):
    text = text.replace(" ", "").upper()
    if len(text) % 2 != 0:
        text += "X"
    return text

def create_blocks(text):
    numbers = [LETTER_TO_NUM[ch] for ch in text]
    return [numbers[i:i+2] for i in range(0, len(numbers), 2)]

def hill_encrypt(plaintext, key_matrix):
    ciphertext = ""
    for block in create_blocks(plaintext):
        block_vector = np.array(block).reshape(-1, 1)
        cipher_block = (key_matrix.dot(block_vector)) % MOD.flatten()
        ciphertext += "".join(NUM_TO_LETTER[num] for num in cipher_block)
    return ciphertext

def hill_decrypt(ciphertext, key_matrix):
    det = int(round(np.linalg.det(key_matrix))) % MOD

    det_inv = mod_inverse(det, MOD)

    adjugate = np.array([
        [key_matrix[1,1], -key_matrix[0,1]],
        [-key_matrix[1,0], key_matrix[0,0]]
    ])
```

```
inverse_matrix = (det_inv * adjugate) % MOD

plaintext = ""
for block in create_blocks(ciphertext):
    block_vector = np.array(block).reshape(-1, 1)
    plain_block = (inverse_matrix.dot(block_vector) % MOD).flatten()
    plaintext += "".join(NUM_TO_LETTER[num] for num in plain_block)

return plaintext

text = input("Enter plaintext: ")
processed = preprocess_text(text)

ciphertext = hill_encrypt(processed, KEY_MATRIX)
print("Encrypted text:", ciphertext)

choice = input("Do you want to decrypt it? (y/n): ")
if choice.lower() == "y":
    decrypted = hill_decrypt(ciphertext, KEY_MATRIX)
    print("Decrypted text:", decrypted)
```

RSA

```
def is_prime(n):
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return n > 1

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def modinv(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    raise Exception("No modular inverse exists")

while True:
    try:
        p = int(input("Enter prime number p: "))
        if is_prime(p):
            break
        print("Not a valid prime!")
    except:
        print("Enter valid integer!")

while True:
    try:
        q = int(input("Enter different prime q: "))
```

```
if is_prime(q) and q != p:  
    break  
  
print("Not a valid prime or same as p!")  
  
except:  
  
    print("Enter valid integer!")  
  
  
print(f"\nYou entered primes: p={p}, q={q}")  
  
  
# Key generation  
  
n = p * q  
  
phi = (p - 1) * (q - 1)  
  
  
possible_e = [i for i in range(2, phi) if gcd(i, phi) == 1]  
  
print("\nPossible values of e:")  
  
print(possible_e)  
  
  
while True:  
  
    try:  
  
        e = int(input("\nChoose e from above list: "))  
  
        if e in possible_e:  
  
            break  
  
        print("Invalid choice!")  
  
    except:  
  
        print("Enter valid integer!")  
  
  
# Find d  
  
d = modinv(e, phi)  
  
  
# Display multiple solutions for d  
for k in range(5):
```

```

print(f"{d + k * phi}")

# Keys

print("\nPublic Key (e, n):", (e, n))
print("Private Key (d, n):", (d, n))

while True:

    try:

        m = int(input("\nEnter message m (< n): "))

        if m < n:

            break

        print("Message must be smaller than n!")

    except:

        print("Enter valid integer!")

C = pow(m, e, n)

print("\nEncrypted ciphertext:", C)

decrypted = pow(C, d, n)

print("Decrypted message:", decrypted)

```

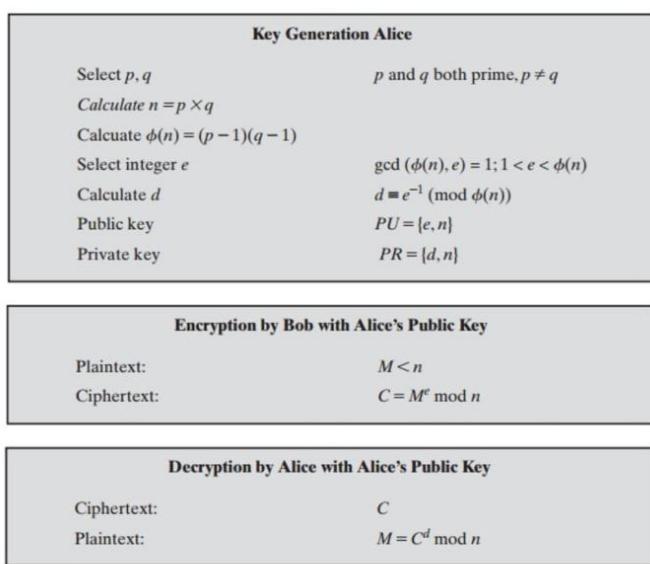


Figure 9.5 The RSA Algorithm

DIFFIE-HELLMAN

```
def is_prime(n):  
    if n < 2:  
        return False  
  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
  
    return True  
  
  
def is_primitive_root(a, q):  
    results = set()  
  
    for k in range(1, q):  
        value = pow(a, k, q)  
        results.add(value)  
  
  
    expected = set(range(1, q))  
  
    if results == expected:  
        return True  
  
    else:  
        return False  
  
  
def all_primitive_roots(q):  
    roots = []  
  
    expected = set(range(1, q))  
  
  
    for a in range(1, q):  
        results = set()  
  
        for k in range(1, q):  
            value = pow(a, k, q)  
            results.add(value)  
    
```

```
if results == expected:  
    roots.append(a)  
return roots  
  
while True:  
    q = int(input("Enter a prime number q: "))  
    if is_prime(q):  
        break  
    else:  
        print("Not a prime! Enter again.")  
  
print("all primitive roots for given number is:", all_primitive_roots(q))  
  
while True:  
    alpha = int(input("Enter alpha (primitive root of q): "))  
    if alpha < q and is_primitive_root(alpha, q):  
        break  
    else:  
        print("Not a valid primitive root! Try again.")  
  
while True:  
    xa = int(input("Enter sender private key xa (< q): "))  
    if xa < q:  
        break  
    else:  
        print("Invalid, try again!")  
  
while True:  
    xb = int(input("Enter receiver private key xb (< q): "))  
    if xb < q:
```

```

break

else:
    print("Invalid, try again!")

ya = pow(alpha, xa, q)
yb = pow(alpha, xb, q)
print("\nPublic keys:")
print("Sender public key (ya) =", ya)
print("Receiver public key (yb) =", yb)

k1 = pow(yb, xa, q)
k2 = pow(ya, xb, q)

print("\nShared keys:")
print("k1 (sender side) =", k1)
print("k2 (receiver side) =", k2)

```

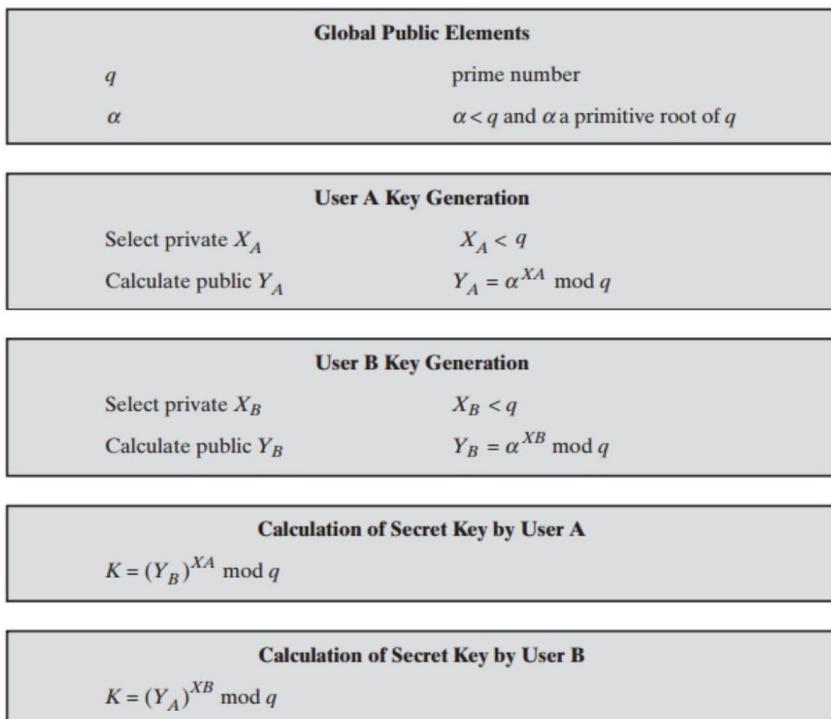


Figure 10.1 The Diffie-Hellman Key Exchange Algorithm

ELGAMAL

```
def is_prime(n):  
    if n < 2:  
        return False  
  
    for i in range(2, int(n**0.5) + 1):  
        if n % i == 0:  
            return False  
  
    return True  
  
  
def is_primitive_root(g, p):  
    required = set()  
  
    for i in range(1, p):  
        required.add(pow(g, i, p))  
  
    return len(required) == p - 1  
  
  
def primitive_roots(p):  
    roots = []  
  
    for g in range(1, p):  
        if is_primitive_root(g, p):  
            roots.append(g)  
  
    return roots  
  
  
while True:  
    p = int(input("Enter a prime number p: "))  
  
    if is_prime(p):  
        break  
  
    else:  
        print(" Not a prime! Enter again.")  
  
  
print("\n Prime number accepted:", p)
```

```
print("All Primitive Roots of p:", primitive_roots(p))
```

```
while True:
```

```
    g = int(input("\nEnter a primitive root g: "))
```

```
    if is_primitive_root(g, p):
```

```
        print(" Valid primitive root!")
```

```
        break
```

```
    else:
```

```
        print(" Not a primitive root! Enter again.")
```

```
x = int(input("\nEnter private key x (secret): "))
```

```
y = pow(g, x, p)
```

```
print("\nPublic Key (p, g, y):", p, g, y)
```

```
while True:
```

```
    m = int(input("\nEnter message m (0 < m < p): "))
```

```
    if 0 < m < p:
```

```
        break
```

```
    print(" Message must satisfy 0 < m < p")
```

```
while True:
```

```
    k = int(input("Enter random key k (1 < k < p): "))
```

```
    if 1 < k < p:
```

```
        break
```

```
    print("k must be between 1 and p-1")
```

```
c1 = pow(g, k, p)
```

```
c2 = (m * pow(y, k, p)) % p
```

```
print("\nCiphertext (c1, c2):", c1, c2)
```

```
s = pow(c1, x, p)
```

```
inv_s = pow(s, p-2, p)
```

```
m_dec = (c2 * inv_s) % p
```

```
print("\nDecrypted Message:", m_dec)
```

Global Public Elements

q	prime number
α	$\alpha < q$ and α a primitive root of q

Key Generation by Alice

Select private X_A	$X_A < q - 1$
Calculate Y_A	$Y_A = \alpha^{X_A} \text{ mod } q$
Public key	$PU = \{q, \alpha, Y_A\}$
Private key	X_A

Encryption by Bob with Alice's Public Key

Plaintext:	$M < q$
Select random integer k	$k < q$
Calculate K	$K = (Y_A)^k \text{ mod } q$
Calculate C_1	$C_1 = \alpha^k \text{ mod } q$
Calculate C_2	$C_2 = KM \text{ mod } q$
Ciphertext:	(C_1, C_2)

Decryption by Alice with Alice's Private Key

Ciphertext:	(C_1, C_2)
Calculate K	$K = (C_1)^{X_A} \text{ mod } q$
Plaintext:	$M = (C_2 K^{-1}) \text{ mod } q$

Figure 10.3 The ElGamal Cryptosystem

MD5 ALGORITHM

```
import java.security.MessageDigest;  
  
import java.util.Scanner;  
  
  
public class MD5Demo {  
  
    public static String getMD5Hash(String input) {  
  
        try {  
  
            MessageDigest md = MessageDigest.getInstance("MD5");  
  
            byte[] messageDigest = md.digest(input.getBytes());  
  
            StringBuilder hexString = new StringBuilder();  
  
            for (byte b : messageDigest) {  
  
                hexString.append(String.format("%02x", b));  
  
            }  
  
            return hexString.toString();  
  
        } catch (Exception e) {  
  
            throw new RuntimeException(e);  
  
        }  
  
    }  
  
  
    public static void main(String[] args) {  
  
        Scanner sc = new Scanner(System.in);  
  
        System.out.println("----- MD5 HASHING -----");  
  
        System.out.print("Enter first message: ");  
  
        String msg1 = sc.nextLine();  
  
  
        System.out.print("Enter slightly changed message: ");  
  
        String msg2 = sc.nextLine();  
  
  
        String hash1 = getMD5Hash(msg1);  
  
        String hash2 = getMD5Hash(msg2);  
  
    }  
}
```

```
System.out.println("\nOriginal Message : " + msg1);
System.out.println("MD5 Hash      : " + hash1);

System.out.println("\nModified Message : " + msg2);
System.out.println("MD5 Hash      : " + hash2);

if (!hash1.equals(hash2)) {
    System.out.println("\n Avalanche Effect: Small input change → Different hash completely!");
}

sc.close();
}
```

SHA

```
import java.security.MessageDigest;  
import java.security.NoSuchAlgorithmException;  
import java.nio.charset.StandardCharsets;  
import java.util.Scanner;  
  
public class SHA1UserInput {  
  
    public static void main(String[] args) {  
  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter the message to hash using SHA-1: ");  
        String input = sc.nextLine();  
        sc.close();  
  
        if (input == null || input.trim().isEmpty()) {  
            System.out.println(" Error: Input cannot be null or empty!");  
            return;  
        }  
  
        try {  
            MessageDigest sha1 = MessageDigest.getInstance("SHA-1");  
  
            byte[] messageBytes = input.getBytes(StandardCharsets.UTF_8);  
  
            byte[] hashBytes = sha1.digest(messageBytes);  
  
            StringBuilder hexHash = new StringBuilder();  
            for (byte b : hashBytes) {  
                hexHash.append(String.format("%02x", b));  
            }  
            System.out.println("The SHA-1 hash of the input is: " + hexHash.toString());  
        } catch (NoSuchAlgorithmException e) {  
            System.out.println("Error: SHA-1 algorithm not found.");  
        }  
    }  
}
```

```
}

System.out.println("\n===== SHA-1 HASHING RESULT =====");
System.out.println("Input Message : " + input);
System.out.println("SHA-1 Hash   : " + hexHash.toString());

if (hexHash.length() == 40) {
    System.out.println(" Hash length verified: 40 characters (160 bits)");
} else {
    System.out.println(" Invalid SHA-1 hash length!");
}

} catch (NoSuchAlgorithmException e) {
    System.out.println(" Error: SHA-1 algorithm not found!");
}
}
```

DES

```
import javax.crypto.Cipher;  
import javax.crypto.KeyGenerator;  
import javax.crypto.SecretKey;  
import javax.crypto.spec.SecretKeySpec;  
import java.util.Base64;  
import java.util.Scanner;  
  
public class DESExample {  
  
    public static void main(String[] args) {  
        try {  
            Scanner sc = new Scanner(System.in);  
  
            System.out.print("Enter plaintext to encrypt: ");  
            String plaintext = sc.nextLine();  
  
            System.out.print("Enter 8-character secret key: ");  
            String keyInput = sc.nextLine();  
  
            sc.close();  
  
            if (keyInput.length() != 8) {  
                System.out.println(" Error: Key must be exactly 8 characters long (64 bits).");  
                return;  
            }  
  
            byte[] keyBytes = keyInput.getBytes();  
            SecretKey secretKey = new SecretKeySpec(keyBytes, "DES");
```

```
Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");

desCipher.init(Cipher.ENCRYPT_MODE, secretKey);

byte[] encryptedBytes = desCipher.doFinal(plaintext.getBytes());

String encryptedText = Base64.getEncoder().encodeToString(encryptedBytes);

desCipher.init(Cipher.DECRYPT_MODE, secretKey);

byte[] decryptedBytes = desCipher.doFinal(Base64.getDecoder().decode(encryptedText));

String decryptedText = new String(decryptedBytes);

System.out.println("\n===== DES ENCRYPTION & DECRYPTION =====");

System.out.println("Plaintext : " + plaintext);

System.out.println("Key      : " + keyInput);

System.out.println("Encrypted : " + encryptedText);

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

if (plaintext.equals(decryptedText)) {

    System.out.println(" Verification Successful: Decrypted text matches the original
plaintext.");

} else {

    System.out.println("Verification Failed: Decrypted text does not match the original
plaintext.");

}

} catch (Exception e) {

    System.out.println(" Error occurred: " + e.getMessage());

}

}
```

AES

```
import javax.crypto.Cipher;  
import javax.crypto.spec.SecretKeySpec;  
import java.util.Scanner;  
import java.util.Arrays;  
  
public class AESExample {  
    public static void main(String[] args) throws Exception {  
        Scanner sc = new Scanner(System.in);  
  
        // Get plaintext and key from user  
        System.out.print("Enter text to encrypt: ");  
        String inputText = sc.nextLine();  
  
        System.out.print("Enter key (less than or equal to 16 chars): ");  
        String keyInput = sc.nextLine();  
  
        // Pad both plaintext and key if shorter than 16 bytes  
        byte[] keyBytes = padTo16Bytes(keyInput.getBytes());  
        byte[] inputBytes = padTo16Bytes(inputText.getBytes());  
  
        // Create AES key  
        SecretKeySpec secretKey = new SecretKeySpec(keyBytes, "AES");  
  
        // Initialize Cipher for AES encryption  
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");  
  
        // Encrypt  
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);  
        byte[] encrypted = cipher.doFinal(inputBytes);
```

```
System.out.println("\nEncrypted (Hex): " + bytesToHex(encrypted));

// Decrypt
cipher.init(Cipher.DECRYPT_MODE, secretKey);
byte[] decrypted = cipher.doFinal(encrypted);
System.out.println("Decrypted text: " + new String(decrypted).trim());
}

// Pads any byte array to 16 bytes with zeros
private static byte[] padTo16Bytes(byte[] input) {
    return Arrays.copyOf(input, 16);
}

// Converts bytes to hex string
private static String bytesToHex(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (byte b : bytes)
        sb.append(String.format("%02X", b));
    return sb.toString();
}
}
```