Caesar cipher

```
import java.util.*;
public class CaesarCipher {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter any String: ");
      String str = sc.nextLine();
      System.out.print("Enter the Key: ");
      int key = sc.nextInt();
      String enc = encrypt(str, key);
      System.out.println("Encrypted String: " + enc);
      System.out.println("Decrypted String: " + decrypt(enc, key));
   }
   static String encrypt(String str, int key) {
      StringBuilder sb = new StringBuilder();
      key %= 26;
      for (char c : str.toCharArray()) {
         if (Character.isUpperCase(c)) {
            c += key; if (c > 'Z') c -= 26;
         } else if (Character.isLowerCase(c)) {
            c += key; if (c > 'z') c -= 26;
         }
         sb.append(c);
      }
      return sb.toString();
   }
   static String decrypt(String str, int key) {
      StringBuilder sb = new StringBuilder();
      key %= 26
```

```
    for (char c : str.toCharArray()) {
      if (Character.isUpperCase(c)) {
        c -= key; if (c < 'A') c += 26;
      } else if (Character.isLowerCase(c)) {
        c -= key; if (c < 'a') c += 26;
      }
      sb.append(c);
    }
    return sb.toString();
  }
}
```

Caesar Cipher is a substitution cipher in which each letter in the plaintext is shifted by a fixed number of positions down the alphabet.

Algorithm for Caesar Cipher Encryption & Decryption

1. Start

2. Input

- Read the string **str**
- Read the integer **key**

3. Normalize the key

- Compute key = key % 26

🔐 **4. Encryption Algorithm**

For each character c in the input string:

1. Check if c is an **uppercase letter**
    - Add key → c = c + key
    - If c > 'Z', wrap around: c = c - 26
2. Else if c is a **lowercase letter**
    - Add key → c = c + key
    - If c > 'z', wrap around: c = c - 26

3. Else (non-alphabet), keep character unchanged
4. Append converted character to **encrypted string**

5. Output the encrypted string

🔓 6. Decryption Algorithm

For each character c in the encrypted string:

1. If c is an **uppercase letter**
    - Subtract key → c = c - key
    - If c < 'A', wrap around: c = c + 26
2. Else if c is a **lowercase letter**
    - Subtract key → c = c - key
    - If c < 'a', wrap around: c = c + 26
3. Else (non-alphabet), keep unchanged
4. Append to **decrypted string**

7. Output the decrypted string

Substitution cipher

A Substitution Cipher is an encryption technique where every character in the plaintext is replaced with a corresponding character from another set (cipher alphabet), based on a predefined rule or key.

Algorithm for Substitution Cipher (Encryption & Decryption)

1. Encryption Algorithm

Input: Plaintext string
Output: Encrypted ciphertext

Steps:

1. Define the alphabet:
   plain = "abcdefghijklmnopqrstuvwxyz"
2. Define the cipher mapping (reverse alphabet):
   cipher = "zyxwvutsrqponmlkjihgfedcba"
3. For each character c in the input string:
   - Find the index of c in plain.
   - If c is in the alphabet:
     - Replace it with the character at the same index in cipher.
   - Else:
     - Keep the character unchanged (spaces, numbers, symbols).
4. Append all mapped characters to form the final encrypted string.
5. Output the ciphertext.

---

2. Decryption Algorithm

Input: Ciphertext
Output: Original plaintext

Steps:

1. Define the alphabet:
   plain = "abcdefghijklmnopqrstuvwxyz"
2. Define the cipher mapping:
   cipher = "zyxwvutsrqponmlkjihgfedcba"

3. For each character c in the encrypted string:
    - Find the index of c in cipher.
    - If c is in the cipher alphabet:
        - Replace it with the character at the same index in plain.
    - Else:
        - Keep the character unchanged.
4. Build the decrypted text from the mapped characters.
5. Output the plaintext.

```
import java.util.Scanner;
public class SubstitutionCipher {
  static final String plain = "abcdefghijklmnopqrstuvwxyz";
  static final String cipher = "zyxwvutsrqponmlkjihgfedcba";
  static String encrypt(String text) {
    StringBuilder out = new StringBuilder();
    for (char c : text.toCharArray()) {
      int idx = plain.indexOf(c);
      out.append(idx >= 0 ? cipher.charAt(idx) : c);
    }
    return out.toString();
  }
  static String decrypt(String text) {
    StringBuilder out = new StringBuilder();
    for (char c : text.toCharArray()) {
      int idx = cipher.indexOf(c);
      out.append(idx >= 0 ? plain.charAt(idx) : c);
    }
    return out.toString();
```

```
  }
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter text: ");
    String input = sc.nextLine();
    String encrypted = encrypt(input);
    System.out.println("Encrypted: " + encrypted);
    String decrypted = decrypt(encrypted);
    System.out.println("Decrypted: " + decrypted);
    sc.close();
  }
}
```

Hill cipher

Hill Cipher is a classical polygraphic substitution cipher that encrypts a block of letters using matrix multiplication modulo 26.
It was invented by **Lester S. Hill** in 1929.

In this cipher:

- Letters are converted into numbers (A=0 to Z=25).
- A square **key matrix** is used for encryption.
- The plaintext block is multiplied with the key matrix.
- Decryption is done using the **inverse of the key matrix modulo 26**.

It encrypts multiple letters at once instead of one letter at a time.

Algorithm for Hill Cipher Encryption

Input:

- A key matrix **K** of size $n \times n$
- A plaintext message divided into blocks of size n
- Modulus 26

Steps:

1. Convert each plaintext letter into its numeric equivalent (A=0, B=1, ..., Z=25).
2. Divide the plaintext into blocks of size *n*.
3. For each block, form a column vector **P** of size *n* × *1*.
4. Multiply the key matrix with the plaintext vector:

$$C = K \times P$$

5. Take each element of the result modulo 26.
6. Convert each numeric value in **C** back into a letter (0→A, ..., 25→Z).
7. Combine all resulting letters to obtain the final **ciphertext**.

⭐ **Algorithm for Hill Cipher Decryption**

Input:

- Ciphertext blocks **C** of size *n*
- Inverse key matrix $\mathbf{K^{-1}}$ modulo 26

Steps:

1. Compute the determinant of the key matrix **K**.
2. Find the multiplicative inverse of the determinant modulo 26.
3. Compute the adjoint (adjugate) of the key matrix.
4. Compute the inverse key matrix modulo 26:

$$K^{-1} = (det(K)^{-1} \times adj(K)) \text{mod } 26$$

5. Convert ciphertext letters into numbers (A=0, ..., Z=25).
6. Divide ciphertext into blocks of size *n* and form vectors **C**.
7. Multiply the inverse key matrix with the ciphertext vector:

$$P = K^{-1} \times C$$

8. Take each element of **P** modulo 26.
9. Convert the numeric values in **P** back into letters.
10. Combine all letters to obtain the **plaintext**.

```
import java.util.*;
public class Hill2x2Static {

  static int[][] key = {
    {3, 3},
    {2, 5}
  };
  static int[][] inv = new int[2][2];
  static Scanner sc = new Scanner(System.in);
  public static void main(String[] args) {
    // Compute inverse once
    computeInverse();
    System.out.print("Enter text: ");
    String text = sc.next().toUpperCase().replaceAll("[^A-Z]", "");
    if (text.length() % 2 != 0) text += "X";    // Padding
    String enc = hill(text, key);
    System.out.println("Encrypted : " + enc);
    String dec = hill(enc, inv);
    System.out.println("Decrypted : " + dec);
  }

  // Hill cipher using any 2x2 matrix
  static String hill(String t, int[][] k) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < t.length(); i += 2) {
      int a = t.charAt(i) - 'A';
      int b = t.charAt(i + 1) - 'A';
      sb.append((char)(mod(k[0][0] * a + k[0][1] * b) + 'A'));
      sb.append((char)(mod(k[1][0] * a + k[1][1] * b) + 'A'));
    }
    return sb.toString();
  }
```

```
    // Compute inverse of 2x2 matrix (static key)
    static void computeInverse() {
        int det = mod(key[0][0]*key[1][1] - key[0][1]*key[1][0]);

        int detInv = -1;
        for (int i = 0; i < 26; i++)
            if (mod(det * i) == 1) { detInv = i; break; }

        if (detInv == -1) throw new RuntimeException("Key not invertible");

        inv[0][0] = mod(key[1][1] * detInv);
        inv[0][1] = mod(-key[0][1] * detInv);
        inv[1][0] = mod(-key[1][0] * detInv);
        inv[1][1] = mod(key[0][0] * detInv);
    }

    static int mod(int x) { return (x % 26 + 26) % 26; }
}
```

DES DES operates on 64-bit blocks (8 bytes) of plaintext and ciphertext.

2. Key Size

DES uses a 64-bit key, but only 56 bits are used for encryption.
The remaining 8 bits are used for parity checking.

3. Symmetric Cipher

The same key is used for both encryption and decryption.

4. Feistel Structure

DES uses a 16-round Feistel network, which means:

- Each round applies substitution, permutation, and mixing.
- Decryption **uses the same process but with round keys in reverse order**

1. Key and IV Generation

DES requires an 8-byte (64-bit) key.
Out of 64 bits, 56 bits are used and 8 bits are parity.
CBC mode also requires an 8-byte Initialization Vector (IV).

Code:

```
key = get_random_bytes(8)

iv = get_random_bytes(8)
```

---

2. Take Plaintext Input

The plaintext is converted into bytes because DES works on bits.

Code:

```
message = b"This is a secret message."
```

---

3. Padding the Plaintext (PKCS#7)

DES encrypts data in 64-bit (8-byte) blocks.
If the message length is not a multiple of 8, padding is added.
Padding ensures the message fits exactly into DES blocks.

Code:

```
padded = pad(message, 8)
```

---

4. Create DES Cipher in CBC Mode

The DES cipher is created with the key and IV.

Code:

cipher = DES.new(key, DES.MODE_CBC, iv)

---

5. CBC Mode Working (Outside the DES Core)

CBC stands for Cipher Block Chaining.

For the first block:

Block1 = Plaintext1 XOR IV

For the following blocks:

Block_i = Plaintext_i XOR Ciphertext_(i-1)

Each resulting block enters the DES encryption process.
CBC ensures that identical plaintext blocks do not produce identical ciphertext blocks, preventing pattern leakage.

DES Algorithm – Steps (Short, Clear, With Brief Explanation)

1. Convert plaintext to a 64-bit block

DES processes data in fixed 64-bit blocks.
If the plaintext is shorter, padding is added; if longer, it is split into multiple 64-bit chunks.

---

2. Apply Initial Permutation (IP)

The 64-bit block is rearranged using the IP table to prepare it for Feistel processing.
After permutation, the block is divided into:
• **L0** = left 32 bits
• **R0** = right 32 bits

---

3. Generate 16 round keys (48 bits each)

DES creates a different key for each of the 16 rounds.
Steps:

1. Apply **PC-1**, reducing the 64-bit key to 56 bits
2. Split into **C0** and **D0** (28 bits each)
3. For each round: left-shift C and D, then apply **PC-2** to form a 48-bit key **Ki**

These round keys help produce strong confusion and diffusion.

---

4. Perform 16 Feistel rounds

Each round transforms the data using the Feistel rule:
• **Li = Ri−1**
• **Ri = Li−1 XOR F(Ri−1, Ki)**
This structure ensures encryption and decryption use the same process in reverse.

---

5. Expand R (32 → 48 bits)

Inside the F-function, the 32-bit right half is expanded to 48 bits using the expansion table.
This allows mixing with the 48-bit round key.

---

6. XOR with subkey

The expanded 48-bit value is XORed with the round key Ki:
B = E(Ri−1) XOR Ki
This step mixes the data with the key.

---

7. S-Box substitution (48 → 32 bits)

The 48-bit result is divided into eight 6-bit parts.
Each part passes through an S-Box, which replaces it with a 4-bit output.
All eight S-Boxes together produce a **32-bit** output.
This introduces non-linearity and strengthens security.

---

8. Apply P-Permutation

The 32-bit output is permuted using the P-table.
This spreads the bits to achieve better diffusion.
The result becomes **F(Ri−1, Ki)**.

---

9. Swap after round 16

After all 16 rounds, the final pair is swapped:
Output = R16 || L16
This swap is a standard part of DES structure.

---

10. Apply Final Permutation (IP-1)

The swapped 64-bit block is passed through the inverse permutation IP-1.
This produces the final **64-bit ciphertext block**, completing DES encryption.

---

7. Combining Encrypted Blocks

CBC mode encrypts all 64-bit blocks and chains them together to form the final ciphertext.

Code:

encrypted = cipher.encrypt(padded)

The ciphertext is displayed in hexadecimal.

---

8. Decryption Process

Decryption uses the same key, same IV, and the same DES steps.
However, the 16 round keys are applied in reverse order (from K16 back to K1).

```
from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get_random_bytes


key = get_random_bytes(8)  # DES uses 8 bytes (64 bits) for key

iv = get_random_bytes(8)   # DES uses 8 bytes for IV


message = b”This is a secret message”;


cipher = DES.new(key, DES.MODE_CBC, iv)

padded = pad(message, DES.block_size)  # Pad to 8-byte block size

encrypted = cipher.encrypt(padded)

print(&quot;Encrypted (hex):&quot;, encrypted.hex())


decipher = DES.new(key, DES.MODE_CBC, iv)

decrypted_padded = decipher.decrypt(encrypted)

decrypted = unpad(decrypted_padded, DES.block_size)

print(“Decrypted”, decrypted.decode(‘utf-8’))
```

BLOW FISH

from Crypto.Cipher import Blowfish

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get_random_bytes

Generate a random key for Blowfish (between 4 and 56 bytes)

key = get_random_bytes(16) # 16 bytes key

iv = get_random_bytes(Blowfish.block_size) # Blowfish block size is 8 bytes

message = b"This is a secret message for Blowfish encryption."

--- Encryption ---

cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)

padded_message = pad(message, Blowfish.block_size)

encrypted = cipher.encrypt(padded_message)

print("Encrypted (hex):", encrypted.hex())

decipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)

decrypted_padded = decipher.decrypt(encrypted)

decrypted = unpad(decrypted_padded, Blowfish.block_size)

print("Decrypted:", decrypted.decode('utf-8'))

Blowfish Algorithm (Combined Normal Working + Code Logic)

Blowfish Algorithm – Working (Short & Clear)

Blowfish is a 64-bit block cipher with a variable key size (32–448 bits).
It uses a Feistel structure with 16 rounds, similar to DES but stronger and faster.

1. Key and IV Generation

Blowfish supports variable key sizes from 32 bits to 448 bits (4 to 56 bytes).
An IV of 8 bytes is required for CBC mode because Blowfish block size is 64 bits.

Code:

key = get_random_bytes(16)

iv = get_random_bytes(Blowfish.block_size) # 8 bytes

---

2. Plaintext Input

The plaintext message is taken and converted to bytes because Blowfish operates on binary data.

Code:

message = b"This is a secret message for Blowfish encryption."

---

3. Padding (PKCS#7)

Blowfish, like DES, works on 64-bit (8-byte) blocks.
If the plaintext is not a multiple of 8 bytes, padding is added so encryption can proceed.

Code:

padded_message = pad(message, Blowfish.block_size)

---

4. Create Blowfish Cipher in CBC Mode

CBC mode requires the key and IV. A Blowfish cipher object is created using these parameters.

Code:

cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)

---

5. CBC Mode Operation (Before Blowfish Core)

CBC mode processes each block as follows:

For the first block:

Block1 = Plaintext1 XOR IV

For each following block:

Block_i = Plaintext_i XOR Ciphertext_(i-1)

Each XOR output is then fed into the Blowfish encryption core.

---

6. Blowfish Core Working (Normal Blowfish Algorithm)

For each 64-bit block, Blowfish performs these standard operations.

6.1 Block Split

A 64-bit block is divided into two 32-bit halves:

- Left half (L0)
- Right half (R0)

6.2 Key Expansion (P-array and S-boxes)

Blowfish generates internal subkeys as follows:

1. Initialize the P-array (18 entries) and four S-boxes (256 entries each) with fixed values.
2. XOR the P-array entries with the key bytes.
3. Encrypt an all-zero block using the current P-array and S-boxes.
4. Replace P-array entries with results.
5. Continue encrypting and updating until all P-array and S-boxes are filled.

This produces the Blowfish round keys.

6.3 Sixteen Feistel Rounds

Blowfish performs 16 rounds using a Feistel structure:

- For each round i (1 to 16):
    1. XOR the left half with P[i]
    2. Apply the Blowfish F-function to the left half
    3. XOR result with right half
    4. Swap the halves

Format:

Li = Ri-1

Ri = Li-1 XOR F(Ri-1) XOR P[i]

6.4 Final Transformation

After 16 rounds:

1. Swap the halves again.
2. XOR left half with P[17]
3. XOR right half with P[18]

This produces the final 64-bit encrypted block.

---

7. Combining Encrypted Blocks

CBC mode chains all encrypted blocks together to form the final ciphertext.

Code:

encrypted = cipher.encrypt(padded_message)

The ciphertext is displayed in hexadecimal format.

---

8. Decryption Process

Blowfish decryption uses:

- The same key
- The same IV
- The same S-boxes and P-array
- Round keys applied in reverse order

Code:

decipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)

decrypted_padded = decipher.decrypt(encrypted)

CBC reverse operation:

Plaintext_i = Blowfish_decrypt(Block_i) XOR Ciphertext_(i-1)

---

9. Removing Padding

PKCS#7 padding added before encryption is removed after decryption.

Code:

decrypted = unpad(decrypted_padded, Blowfish.block_size)

---

10. Output Plaintext

The decrypted byte data is converted back into a UTF-8 string to get the original message.

Code:

print(decrypted.decode("utf-8"))

Diffie-Hellman Key Exchange

Diffie–Hellman (DH) is a method that allows two parties to **securely generate a shared secret key** over an **insecure channel** — even if an attacker is listening.

No secret is sent over the network

```
<!DOCTYPE html>
<html>
<head>
 <title>Diffie-Hellman Key Exchange</title>
</head>
<body>
 <h2>Diffie-Hellman Key Exchange</h2>

 <label>Prime (P):</label>
 <input id="P" /><br>

 <label>Primitive Root (G):</label>
 <input id="G" /><br>

 <label>Alice Private Key (a):</label>
 <input id="a" /><br>

 <label>Bob Private Key (b):</label>
 <input id="b" /><br><br>

 <button onclick="computeDH()">Compute</button>

 <h3>Output:</h3>
 <pre id="out"></pre>

 <script>
  // (a^b) % p
```

```
function modPow(a, b, p) {
  let result = 1;
  a = a % p;
  while (b > 0) {
    if (b % 2 === 1) result = (result * a) % p;
    b = Math.floor(b / 2);
    a = (a * a) % p;
  }
  return result;
}

function computeDH() {
  let P = parseInt(document.getElementById("P").value);
  let G = parseInt(document.getElementById("G").value);
  let a = parseInt(document.getElementById("a").value);
  let b = parseInt(document.getElementById("b").value);

  let out = "";

  out += "Public prime (P): " + P + "\n";
  out += "Primitive root (G): " + G + "\n";
  out += "Alice's private key (a): " + a + "\n";
  out += "Bob's private key (b): " + b + "\n\n";

  // Public values
  let x = modPow(G, a, P);
  let y = modPow(G, b, P);

  out += "Alice sends (G^a mod P): " + x + "\n";
  out += "Bob sends (G^b mod P): " + y + "\n\n";
```

```
    // Shared secret
    let secretA = modPow(y, a, P);
    let secretB = modPow(x, b, P);

    out += "Shared secret (Alice): " + secretA + "\n";
    out += "Shared secret (Bob): " + secretB + "\n\n";
    out += "Are both equal? " + (secretA === secretB);

    document.getElementById("out").textContent = out;
  }
 </script>
</body>
</html>
```

Diffie–Hellman Key Exchange – Working Algorithm

Step 1: Select Public Parameters

Two values are agreed upon publicly:

- p= a large prime number
- g= a primitive root modulo p

These values can be known to everyone; they don't affect security.

Step 2: Each User Selects a Private Key

- Alice chooses a private key a(kept secret)
- Bob chooses a private key b(kept secret)

These are random numbers.

Step 3: Generate Public Keys

Each user computes a public value based on their private key.

- Alice computes:

$$A = g^a \bmod p$$

- Bob computes:

$$B = g^b \bmod p$$

These **A** and **B** are sent over the insecure channel.

Step 4: Exchange Public Keys

- Alice sends $A$to Bob
- Bob sends $B$to Alice

Even if an attacker sees these values, they still cannot derive the private keys.

Step 5: Compute Shared Secret Key

After receiving the public value from the other side, both compute the same secret key:

- Alice computes:

$$K = B^a \bmod p$$

- Bob computes:

$$K = A^b \bmod p$$

Both generate the same shared secret because:

$$(g^b)^a \bmod p = (g^a)^b \bmod p$$

Final Shared Secret Key = K

This key is then used for encryption (e.g., AES) in secure communication.

📌 **Example (Simple Numbers)**

Let:
$p = 23, g = 5$
Alice picks $a = 6$
Bob picks $b = 15$

Alice computes:

$$A = 5^6 \bmod 23 = 8$$

Bob computes:

$$B = 5^{15} \bmod 23 = 19$$

Shared Key:
Alice: $19^6 \bmod 23 = 2$
Bob: $8^{15} \bmod 23 = 2$

Shared secret key = 2

Rijndael algorithm

Rijndael is the original algorithm from which AES was standardized.)

Rijndael is a symmetric block cipher that encrypts data using blocks of size 128, 160, 192, 224, or 256 bits and supports key sizes of
128, 160, 192, 224, or 256 bits.
AES is a restricted version of Rijndael where the block size is fixed to 128 bits.

```
from py3rijndael import RijndaelCbc, Pkcs7Padding
from Crypto.Random import get_random_bytes

block_size = 32  # 256-bit block size

key = get_random_bytes(32)   # 32-byte key
iv = get_random_bytes(32)    # 32-byte IV

rijndael = RijndaelCbc(key=key, iv=iv, padding=Pkcs7Padding(block_size),
                       block_size=block_size)

plaintext = b"Rijndael block cipher test message."
ciphertext = rijndael.encrypt(plaintext)
decrypted = rijndael.decrypt(ciphertext)

print("Key:", key.hex())
print("IV:", iv.hex())
print("Encrypted:", ciphertext.hex())
print("Decrypted:", decrypted.decode("utf-8"))
```

Rijndael (AES) Algorithm – Steps (Short, Clear, With Brief Explanation)

1. Convert plaintext into a 128-bit block

AES (Rijndael) works on fixed **128-bit (16-byte)** blocks.
The plaintext is padded or split into 16-byte blocks as needed.

---

2. Arrange data into a 4×4 State matrix

The 128-bit block is placed column-wise into a **4×4 matrix**, called the *State*.
AES transformations operate on this matrix.

---

3. Key Expansion (Generate Round Keys)

AES derives multiple round keys from the main key.
Number of rounds depends on key size:
• 128-bit key → 10 rounds
• 192-bit key → 12 rounds
• 256-bit key → 14 rounds

Key expansion steps:

1. Split the main key into 4-byte words
2. Apply RotWord (cyclic left shift)
3. Apply SubWord (byte substitution using S-Box)
4. XOR with the Rcon value
5. Continue generating enough round keys for all rounds

These round keys are used in every stage of AES.

---

4. Initial AddRoundKey

Before any round begins, the State matrix is XORed with the first round key.
This mixes the key with the plaintext immediately.

---

5. Perform the AES rounds

AES has 10, 12, or 14 rounds depending on key size.
Each round except the final one performs four operations.

Round Step 1: SubBytes

Each byte in the State is replaced using a fixed **non-linear S-Box**.
This gives confusion and protects against algebraic attacks.

Round Step 2: ShiftRows

Rows of the State matrix are cyclically shifted:
- Row 0 → no shift
- Row 1 → shift left by 1
- Row 2 → shift left by 2
- Row 3 → shift left by 3

This spreads byte positions across columns.

Round Step 3: MixColumns

Each column of the State is multiplied by a fixed matrix in **GF(2^8)**.
This mixes each column's bytes to provide diffusion.

Round Step 4: AddRoundKey

The State matrix is XORed with the round key for that round.
This binds encryption to the expanded key schedule.

---

6. Final Round (No MixColumns)

The last round omits MixColumns.
It performs only:

1. SubBytes
2. ShiftRows
3. AddRoundKey

This produces the final encrypted State.

---

7. Convert the State matrix back to 128-bit block

The final State matrix is read column-wise to produce the **128-bit ciphertext block**.

---

8. Output ciphertext

For multiple blocks, AES encrypts each block independently (ECB) or with chaining modes like CBC, CTR, etc., depending on the mode used.

Rijndael (AES) Decryption Steps (Short & Clear)

AES decryption essentially reverses the encryption process using **inverse steps** and **round keys in reverse order**.

---

1. Input Preparation

Convert ciphertext into a 128-bit block and arrange into a 4×4 State matrix.

---

2. Key Expansion

Generate the same round keys used during encryption.

---

3. Initial AddRoundKey

XOR the State with the **last round key**.

---

4. Perform Inverse Rounds

For each round (except the last):

1. **InvShiftRows** – shift rows right
2. **InvSubBytes** – apply inverse S-box
3. **AddRoundKey** – XOR State with the round key
4. **InvMixColumns** – inverse column mixing using GF(2^8)

---

5. Final Inverse Round (No InvMixColumns)

1. InvShiftRows
2. InvSubBytes
3. AddRoundKey

---

6. Reconstruct Plaintext

Convert the final State matrix back to a 128-bit block.
This is the recovered plaintext.

SHA-1 algorithm

SHA-1 (Secure Hash Algorithm 1) is another hashing algorithm, but stronger than MD5.
It produces a 160-bit hash (40-character hex string).

Example:

hello → f572d396fae9206628714fb2ce00f72e94f2258f

- **More secure than MD5 (but still considered broken now).**

import hashlib

```
text = "This is the message to be hashed."

# Create a SHA-1 hash object and feed it the encoded text

sha1_hash = hashlib.sha1(text.encode('utf-8'))

# Get the hexadecimal digest of the hash

digest = sha1_hash.hexdigest()

   print("SHA-1 Digest:", digest)
```

Expanded SHA-1 Algorithm (Detailed Explanation + Full Sentences)

---

1. INPUT: message M

The algorithm begins by taking the original message M, which may be of any length and any byte content.
This message is the data that will be transformed into a fixed-size 160-bit hash.

---

2. PAD THE MESSAGE

Before processing, the message must be padded so that its length fits the SHA-1 block structure.

- First, the message is converted into its byte representation.
- Then a single **1 bit** (in hex: 0x80) is appended.
- After this, enough **zero bytes (0x00)** are added so that the total length becomes exactly **56 bytes less than a multiple of 64**, ensuring the final 64 bits can store the message length.
- Finally, append the **original message length** as a 64-bit big-endian integer.
  This padding ensures the message length is correctly encoded and prevents ambiguity in later processing.

---

3. SPLIT INTO 512-BIT BLOCKS

The padded message is then divided into equal chunks of **512 bits (64 bytes)**.
Each block will be processed independently, and the results will be chained together to produce the final hash.
The number of blocks N depends on the size of the original message after padding.

---

4. INITIALIZE HASH REGISTERS

SHA-1 uses five **32-bit registers** to store its running state.
These values are initialized to specific constants defined in the SHA-1 standard:

H0 = 0x67452301

H1 = 0xEFCDAB89

H2 = 0x98BADCFE

H3 = 0x10325476

H4 = 0xC3D2E1F0

These constants act as the **initial vector** and provide the starting point for the hash computation.

---

5. PROCESS EACH 512-BIT BLOCK

Each block undergoes the same sequence of operations.

---

a. Construct the Message Schedule W[0..79]

The 512-bit block is divided into **16 words of 32 bits** each.
These form W[0] to W[15].

To increase diffusion, the remaining **64 words** (W[16] to W[79]) are created by taking XORs of earlier words and applying a **1-bit left rotation**.
This expansion ensures each part of the message influences many rounds.

---

b. Initialize Working Variables

Five working variables (A, B, C, D, E) are set to the current values of H0..H4.
These variables will be updated in every round and represent the evolving internal state of the hash computation.

---

c. Perform 80 Mixing Rounds

SHA-1 performs **80 rounds** of operations for each block.
Each round uses:

- A nonlinear function (combination of AND, OR, XOR)
- A constant value (K)
- A left rotation
- The current word W[i]

These operations mix the data thoroughly and produce the avalanche effect, where small changes in the input create large differences in the output.

At each round, values A, B, C, D, and E are updated based on the computed values.
All operations are performed on 32-bit words.

---

d. Update the Hash Registers

After all 80 rounds are complete, the working variables (A..E) are added back to the original hash registers (H0..H4).
This chaining step ensures that every block influences the final hash, and blocks processed later depend on all previous blocks.

---

6. OUTPUT THE FINAL DIGEST

After all blocks have been processed, the values in the registers:

H0, H1, H2, H3, H4

are concatenated in order to form the final **160-bit SHA-1 hash**.
This digest is typically represented as a **40-character hexadecimal string**, which is the standard output produced by hashing libraries such as Python's hashlib.

MD5

MD5 (Message Digest 5) is a hashing algorithm that turns any input (text, file, etc.) into a 128-bit hash (32-character hex string).

Example:

hello → 5d41402abc4b2a76b9719d911017c592

- **It is fast, but not secure anymore.**

```
def md5_digest(text):
# Encode the text to bytes, compute MD5 hash, and get the hex digest
return hashlib.md5(text.encode(&amp;#39;utf-8&amp;#39;)).hexdigest()

# Example usage
text = &amp;quot;Your text here&amp;quot;
digest = md5_digest(text)
print(&amp;quot;MD5 digest:&amp;quot;, digest)
```

1. INPUT: message M

Take the message in bytes.

---

2. PAD THE MESSAGE

- Append a single 1 bit (0x80).
- Add 0x00 bytes until the message length ≡ **56 mod 64**.
- Append the original length as a **64-bit little-endian** integer.
  This makes the total length a multiple of **512 bits**.

---

3. SPLIT INTO 512-BIT BLOCKS

Break the padded message into 64-byte blocks.

---

4. INITIALIZE 4 REGISTERS

MD5 uses four 32-bit registers:

A = 0x67452301

B = 0xEFCDAB89

C = 0x98BADCFE

D = 0x10325476

---

5. PROCESS EACH BLOCK

a. Break block into 16 words (32-bit each)

b. Perform 64 steps

- Each step uses:
    - A nonlinear function (F, G, H, or I)
    - One of the 16 words
    - A constant from a predefined table
    - Fixed left-rotations
- Update the registers A, B, C, D in each round.

MD5 has **4 rounds × 16 steps each = 64 steps**.

---

6. UPDATE FINAL HASH STATE

Add the updated A, B, C, D values back into the original registers.

---

7. OUTPUT

Concatenate:

A || B || C || D

to produce the final **128-bit MD5 hash** (32-character hex digest).