Here are the algorithms followed by the corresponding code for each encryption method:

1. DES Algorithm

Algorithm:

1. **Input Key and Data**: Accept the key and data to encrypt from the user.
2. **Adjust Key Length**: Repeat or truncate the key so that it matches the length of the data.
3. **Encrypt Data**: For each character in the data, add the ASCII values of the data and the key, then take modulo 256 of the sum.
4. **Decrypt Data**: For each character in the encrypted data, subtract the ASCII values of the key and the encrypted character, then take modulo 256.
5. **Display Results**: Print the encrypted and decrypted data.

Code:

```
key = input("Enter key: ")

data = input("Enter the data: ")



key = (key * (len(data) // len(key) + 1))[:len(data)]



encrypted = ""

for i in range(len(data)):

    encrypted += chr((ord(data[i]) + ord(key[i])) % 256)



decrypted = ""

for i in range(len(encrypted)):

    decrypted += chr((ord(encrypted[i]) - ord(key[i])) % 256)



print("Encrypted:", encrypted)

print("Decrypted:", decrypted)
```

2. AES Algorithm

Algorithm:

1. **Input Key and Data**: Accept the AES key (16 bytes) and data to encrypt from the user.
2. **Validate Key Length**: Ensure the AES key is exactly 16 bytes long.

3. **Encrypt Data**: Use AES in ECB mode to encrypt the data.
4. **Decrypt Data**: Decrypt the encrypted data and remove any padding.
5. **Display Results**: Print the encrypted data in hexadecimal format and the decrypted data.

Code:

```
from Crypto.Cipher import AES


aes_key = input("Enter 16-byte AES key: ").encode('utf-8')

data = input("Enter data to encrypt: ").encode('utf-8')


if len(aes_key) != 16:

    print("AES key must be 16 bytes")

    exit()


cipher = AES.new(aes_key, AES.MODE_ECB)

encrypted = cipher.encrypt(data.ljust(16))

decrypted = cipher.decrypt(encrypted).rstrip()


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

print("Decrypted:", decrypted.decode())
```

3. RSA Algorithm

Algorithm:

1. **Input Primes and Message**: Accept two prime numbers p and q, and the message to be encrypted.
2. **Key Generation**:
   - Calculate n = p * q.
   - Calculate t = (p - 1) * (q - 1).
   - Find e (the public exponent) such that gcd(e, t) = 1.
   - Find d (the private exponent) such that (d * e) % t = 1.
3. **Encryption**: Encrypt the message using the public key formula c = (message^e) % n.
4. **Decryption**: Decrypt the ciphertext using the private key formula message = (c^d) % n.
5. **Display Results**: Print the encrypted and decrypted messages.

Code:

```
from math import gcd


def RSA(p: int, q: int, message: int):
    n = p * q
    t = (p - 1) * (q - 1)
    e = 0
    for i in range(2, t):
        if gcd(i, t) == 1:
            e = i
            break
    if e == 0:
        print("Failed to find an appropriate value for e.")
    d = 0
    for j in range(1, t):
        if (j * e % t) == 1:
            d = j
            break
    if d == 0:
        print("Failed to find an appropriate value for d.")
    c = (message ** e) % n
    print(f"Encrypted message is: {c}")
    mes = (c ** d) % n
    print(f"Decrypted message is: {mes}")


# Example usage
RSA(p=10, q=20, message=100)
```

4. Blowfish Algorithm

Algorithm:

1. **Input Key and Data**: Accept a Blowfish key and data from the user.
2. **Encrypt Data**: Use Blowfish in CBC mode to encrypt the data, applying padding to match the block size.
3. **Decrypt Data**: Decrypt the encrypted data using the same key and IV, then remove padding.
4. **Display Results**: Print the encrypted data in hexadecimal format and the decrypted data.

Code:

```
from Crypto.Cipher import Blowfish

from Crypto.Util.Padding import pad, unpad



key = input("Enter Blowfish key (4-56 bytes): ").encode('utf-8')

data = input("Enter data to encrypt: ").encode('utf-8')



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

encrypted = cipher.encrypt(pad(data, Blowfish.block_size))

iv = cipher.iv

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

decrypted = unpad(decipher.decrypt(encrypted), Blowfish.block_size)



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

print("Decrypted:", decrypted.decode())
```

5. Caesar Cipher

Algorithm:

1. **Input Plaintext and Shift**: Accept the plaintext and shift value from the user.
2. **Encrypt Data**: For each character, shift it by the given value, wrapping around if necessary.
3. **Display Encrypted Text**: Print the encrypted text.

Code:

```
def caesar_encrypt(plaintext, shift):

    result = ""
```

```
    for char in plaintext:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            result += chr((ord(char) - shift_base + shift) % 26 + shift_base)
        else:
            result += char
    return result


text = input("Enter text to encrypt: ")
shift = int(input("Enter shift value: "))
encrypted = caesar_encrypt(text, shift)
print("Encrypted text:", encrypted)
```

6. Diffie-Hellman Key Exchange

Algorithm:

1. **Input Prime and Generator**: Accept prime number p and generator g.
2. **Input Private Keys**: Accept private keys a_private and b_private for Alice and Bob.
3. **Public Key Calculation**: Compute public keys A = g^a_private % p and B = g^b_private % p.
4. **Shared Secret Calculation**: Compute shared secrets shared_secret_A = B^a_private % p and shared_secret_B = A^b_private % p.
5. **Display Results**: If both shared secrets match, print the shared secret and success message.

Code:

```
p = 23
g = 5


# Private keys (secret)
a_private = 6
b_private = 15
print("a_private key:", a_private)
print("b_private key:", b_private)
```

```
# Public values (A and B)
A = pow(g, a_private, p)
B = pow(g, b_private, p)
print("Alice's public key:", A)
print("Bob's public key:", B)

# Shared secrets
shared_secret_A = pow(B, a_private, p)
shared_secret_B = pow(A, b_private, p)

if shared_secret_A == shared_secret_B:
    print(f"Shared secret: {shared_secret_A}")
    print("Key exchange successful!")
else:
    print("Error: Shared secrets do not match.")
```

Understanding MD5 in Simple Terms

The *MD5 (Message-Digest Algorithm 5)* is a hashing algorithm that takes an input (message) and converts it into a fixed-size string of 128 bits (16 bytes). Think of it as a unique "digital fingerprint" for your input data.

How MD5 Works

MD5 creates a unique hash value (fingerprint) for any input data, no matter the size. Let’s break the process into simple steps:

Step 1: Message Padding

- If your message doesn’t fit neatly into chunks of *512 bits (64 bytes)*, MD5 adds extra data to make it fit.

- How? It:

  1. Appends a 1 bit to the message.

  2. Fills the rest with `0`s until there’s space for the length of the message at the end.

  3. Appends the length of the original message in bits (as a 64-bit number).

Example:

- Original message: "Hello" (40 bits in binary).

- After padding: "Hello" + 1 + 000... + 00000000 00000000 00101000 (length = 40 in binary).

Step 2: Message Splitting

- The padded message is split into *512-bit (64-byte) blocks*.

- Each block will go through the hashing process one at a time.

Step 3: Hash Computation

Each block is processed to create the hash value. MD5 does this in *4 rounds* using mathematical and bitwise operations:

1. *Initialize:* Start with a fixed set of values (called registers: A, B, C, D).

2. *Mix:* Process each 512-bit block with:

   - *Bitwise operations*: AND, OR, XOR, ROTATE.

   - *Modular addition*: Adding numbers modulo \(2^{32}\).

3. *Update:* Modify the registers after each operation.

4. *Repeat:* Go through 64 iterations (16 per round) to mix the data thoroughly.

This process ensures that even a tiny change in the input creates a completely different hash value (avalanche effect).

Step 4: Final Hash

After processing all blocks:

- Combine the modified registers (A, B, C, D) to produce the *128-bit hash*.

- The hash is usually represented as a 32-character hexadecimal string.

Example:

- Input: "Hello"

- Output Hash: b1946ac92492d2347c6235b4d2611184

Properties of MD5

1. *Deterministic:* The same input will always produce the same hash.

2. *Fixed Size:* No matter the size of the input, the hash is always 128 bits (16 bytes).

3. *Non-reversible:* You can’t get the original message from the hash (one-way function).

4. *Sensitive to Changes:* Even the smallest change in input completely alters the hash.

Applications of MD5

1. *Data Integrity:* Verify if a file or message has been altered by comparing hash values.

2. *Password Storage (Historically):* Hash passwords before storing them.

3. *Digital Signatures:* Check authenticity and integrity of digital data.

Security Issues with MD5

Although MD5 was widely used, it is no longer considered secure because:

1. *Collision Attacks:* Two different inputs can produce the same hash.

2. *Pre-image Attacks:* An attacker could potentially find the original message from a given hash.

Better Alternatives

Because MD5 has weaknesses, more secure hashing algorithms are recommended:

- *SHA-256*: Produces a 256-bit hash and is secure against collisions.

- *SHA-3*: Newer algorithm designed to be resistant to modern attacks.

- *BLAKE2*: Highly efficient and secure.

If you'd like, I can show you how to implement MD5 in Python or compare it to modern hashing methods!