```
# Simple RSA Implementation

from math import gcd

# Key generation
def generate_keys():
    p, q = 61, 53  # Two prime numbers
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1
    e = 3
    while gcd(e, phi) != 1:
        e += 2  # Increment by 2 to ensure e remains odd (odd numbers are more likely to be coprime)

    # Calculate modular inverse of e
    d = pow(e, -1, phi)

    return (e, n), (d, n)  # Public and Private keys

# Encryption
def encrypt(public_key, plaintext):
    e, n = public_key
    return [(ord(char) ** e) % n for char in plaintext]

# Decryption
def decrypt(private_key, ciphertext):
    d, n = private_key
    return ''.join([chr((char ** d) % n) for char in ciphertext])

# Example usage
```

```
public_key, private_key = generate_keys()
message = "HELLO"
encrypted = encrypt(public_key, message)
decrypted = decrypt(private_key, encrypted)

print("Public Key:", public_key)
print("Private Key:", private_key)
print("Original Message:", message)
print("Encrypted Message:", encrypted)
print("Decrypted Message:", decrypted)
```

Imports

python

Copy code

from math import gcd

- **gcd**: This function computes the greatest common divisor of two numbers. It is used to ensure that the encryption key e is coprime with ϕ\phiϕ (explained later).

Key Generation

python

Copy code

def generate_keys():

- This function generates the public and private keys required for RSA encryption and decryption.

---

python

Copy code

```
    p, q = 61, 53  # Two prime numbers
```

- RSA starts by selecting two prime numbers, ppp and qqq. These numbers are essential for generating the keys.
- Here, p is 61, and q is 53.

---

python

Copy code

```
    n = p * q
```

- n is the product of ppp and qqq.
- n is part of both the public and private keys and serves as the modulus for encryption and decryption.

---

python

Copy code

```
    phi = (p - 1) * (q - 1)
```

- ϕ\phiϕ (Euler's totient function) is calculated as (p−1)(q−1)(p - 1)(q - 1)(p−1)(q−1). It represents the number of integers less than n that are coprime with n.

---

python

Copy code

```
e = 3

while gcd(e, phi) != 1:

    e += 2  # Increment by 2 to ensure e remains odd (odd numbers are more likely to be coprime)
```

- The encryption exponent eee is initialized to 3.
- The while loop ensures that eee is coprime with ϕ\phiϕ by checking gcd(e,ϕ)=1\text{gcd}(e, \phi) = 1gcd(e,ϕ)=1.
- eee is incremented by 2 each time to test the next odd number.

---

python

Copy code

```
d = pow(e, -1, phi)
```

- ddd is the modular multiplicative inverse of eee modulo ϕ\phiϕ.
- It satisfies the equation (e·d)mod ϕ=1(e \cdot d) \mod \phi = 1(e·d)modϕ=1. This value is essential for decryption.

---

python

Copy code

```
return (e, n), (d, n)  # Public and Private keys
```

- The function returns the public key (e,n)(e, n)(e,n) and the private key (d,n)(d, n)(d,n).
    - The **public key** is used for encryption.
    - The **private key** is used for decryption.

Encryption

python

Copy code

```
def encrypt(public_key, plaintext):
```

- This function encrypts a message (plaintext) using the public key.

---

python

Copy code

```
e, n = public_key
```

- Extracts the values of eee (encryption exponent) and nnn (modulus) from the public key.

---

python

Copy code

```
return [(ord(char) ** e) % n for char in plaintext]
```

- Encrypts each character of the plaintext:
    - ord(char): Converts the character to its ASCII value.
    - ** e: Raises the ASCII value to the power of eee.
    - % n: Takes the result modulo nnn.
- The result is a list of integers representing the encrypted message.

Decryption

python

Copy code

```
def decrypt(private_key, ciphertext):
```

- This function decrypts the ciphertext using the private key.

---

python

Copy code

```
d, n = private_key
```

- Extracts the values of ddd (decryption exponent) and nnn (modulus) from the private key.

---

python

Copy code

```
return ''.join([chr((char ** d) % n) for char in ciphertext])
```

- Decrypts each integer in the ciphertext:
    - ** d: Raises the encrypted value to the power of ddd.

- % n: Takes the result modulo nnn.
- chr(...): Converts the result back to a character.

- Joins the characters into the original plaintext message.

Example Usage

python

Copy code

public_key, private_key = generate_keys()

- Calls the generate_keys function to create the public and private keys.

---

python

Copy code

message = "HELLO"

- Defines the plaintext message to be encrypted.

---

python

Copy code

encrypted = encrypt(public_key, message)

- Encrypts the plaintext message using the public key.

---

python

Copy code

decrypted = decrypt(private_key, encrypted)

- Decrypts the ciphertext using the private key.

Output

python

Copy code

print("Public Key:", public_key)

print("Private Key:", private_key)

print("Original Message:", message)

```
print("Encrypted Message:", encrypted)

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

- Prints the keys, original message, encrypted message, and decrypted message to verify the implementation.

RSA Workflow Summary:

1. **Key Generation:**
    - Generate two large prime numbers ppp and qqq.
    - Compute n=p·qn = p \cdot qn=p·q and ϕ=(p−1)(q−1)\phi = (p - 1)(q - 1)ϕ=(p−1)(q−1).
    - Select eee such that 1<e<ϕ1 < e < \phi1<e<ϕ and gcd(e,ϕ)=1\text{gcd}(e, \phi) = 1gcd(e,ϕ)=1.
    - Compute ddd, the modular inverse of eee modulo ϕ\phiϕ.
2. **Encryption:**
    - Ciphertext = (ASCII of character)emod n(\text{ASCII of character})^e \mod n(ASCII of character)emodn.
3. **Decryption:**
    - Plaintext = (Ciphertext)dmod n(\text{Ciphertext})^d \mod n(Ciphertext)dmodn.