**Implementation of SHA-512**

**Description**

This is a simple implementation of SHA-512 in Python.

**What is SHA-512**

SHA-512 is a cryptographic hash function that outputs a 512-bit (64-byte) hash value. It uses 1024 bits per block with 80 rounds.

**Algorithm**

- Ask for input message from the user.

- Encode the input message with `UTF-8`

- Add the padding bits as per necessity.

- Now, divide the message in `N \\* 1024` blocks if the length of message >= 896.

- Initialize the constant K and HASH\_VALUE

- For every block of input apply the `compression\_function()`

- The final hash value will be stored in the HASH\_VALUE constant. This value will be used as seed for hashing another message.

**Functions Used**

Ch(e, f, g)

return (e & f) ^ (~e & g)

The `Ch` function is a bitwise operation used in the SHA-512 compression function. It operates on three 64-bit inputs (e, f, and g) and returns a 64-bit result.

Maj(a, b, c)

return (a & b) ^ (a & c) ^ (b & c)

The `Maj` function is another bitwise operation used in the SHA-512 compression function. It operates on three 64-bit inputs (a, b, and c) and returns a 64-bit result.

rotr(x, n)

return (x >> n) | (x <<code (64 - n))

The `rotr()` (right rotate) function performs a bitwise right rotation of a 64-bit value "x" by "n" bits.

summation\_a(a)

return rotr(a, 28) ^ rotr(a, 34) ^ rotr(a, 39)

The `summation\_a()` function combines right rotations and XOR operations on a 64-bit input "a" and returns a 64-bit result.

summation\_e(e)

return rotr(e, 14) ^ rotr(e, 18) ^ rotr(e, 41)

The `summation\_e()` function combines right rotations and XOR operations on a 64-bit input "e" and returns a 64-bit result.

sigma\_0(word)

return rotr(word, 1) ^ rotr(word, 8) ^ (word >> 7)

The `sigma\_0()` function calculates a value based on right rotations and XOR operations on a 64-bit word.

sigma\_1(word)

return rotr(word, 19) ^ rotr(word, 61) ^ (word >> 6)

The `sigma\_1` function calculates another value based on right rotations and XOR operations on a 64-bit word.

addition\_modulo\_2\_64(value)

return value % (2\*\*64)

The `addition\_modulo\_2\_64()` function performs modular addition of a 64-bit value. It makes sure that the value doesn't exceed the length of 64.

pad\_message(message)

message += b"\x80" #Adding 1 byte (10000000)

while len(message) % 128 != 112:

message += b"\x00"

message += (len(message) \* 8).to\_bytes(16, "big")

return message

The `pad\_message()` function adds padding to the input message to make its length a multiple of 128 bytes. It also appends the message length in bits at the end.

divide\_to\_blocks(message)

blocks = []

for i in range(0, len(message), 128):

blocks.append(message[i : i + 128])

return blocks

The `divide\_to\_blocks` function divides a padded message into 128-byte blocks, which are processed by the SHA-512 compression function.

**SHA-512 Compression Function**

compression\_function(message)

for t in range(16):

W[t] = int.from\_bytes(message[t \* 8 : (t + 1) \* 8], byteorder="big")

for t in range(16, 80):

W[t] = sigma\_1(W[t - 2] + W[t - 7]) + sigma\_0(W[t - 15] + W[t - 16])

The first loop calculates W0 - W15.The second loop calculates W16 - W79.

for t in range(80):

T1 = h + (Ch(e, f, g) + (rotr(e, 14) ^ rotr(e, 18) ^ rotr(e, 41)) + K[t] + W[t])

T2 = (rotr(a, 28) ^ rotr(a, 34) ^ rotr(a, 39)) + Maj(a, b, c)

h = g

g = f

f = e

e = addition\_modulo\_2\_64(d + T1)

d = c

c = b

b = a

a = addition\_modulo\_2\_64(T1 + T2)

#intermediate Hash values

HASH\_VALUE[0] = addition\_modulo\_2\_64(HASH\_VALUE[0] + a)

HASH\_VALUE[1] = addition\_modulo\_2\_64(HASH\_VALUE[1] + b)

HASH\_VALUE[2] = addition\_modulo\_2\_64(HASH\_VALUE[2] + c)

HASH\_VALUE[3] = addition\_modulo\_2\_64(HASH\_VALUE[3] + d)

HASH\_VALUE[4] = addition\_modulo\_2\_64(HASH\_VALUE[4] + e)

HASH\_VALUE[5] = addition\_modulo\_2\_64(HASH\_VALUE[5] + f)

HASH\_VALUE[6] = addition\_modulo\_2\_64(HASH\_VALUE[6] + g)

HASH\_VALUE[7] = addition\_modulo\_2\_64(HASH\_VALUE[7] + h)

The `compression\_function` is the core of the SHA-512 hashing process. It takes a 128-byte message block as input and updates the `HASH\_VALUE` accordingly.

**Usage**

1. Input a message after running the program.

2. The code pads the message, divides it into blocks, and initiates the SHA-512 hashing process.

3. The final 512-bit (128-byte) hash value is printed as a hexadecimal string.