UCS1505 - INTRODUCTION TO CRYPTOGRAPHIC TECHNIQUES

INTERNAL ASSIGNMENT

SHA-256

AIM:

To write a Python code for a simplified Implementation of the SHA-256 Hashing Algorithm.

A LITTLE BACKGROUND:

Hash Algorithms

For a specific input a cryptographic hash algorithm always generates the same fixed-length output or 'digest'.

A secure hash function is a One-Way Function (OWF) that is computationally impossible to reverse. This makes it useful to compare 'hashed' versions of texts to check for message authenticity, as opposed to decrypting the text to obtain the original version.

For a hash to be secure or strong, it needs to be collision resistant, i.e., it should be computationally infeasible

1. to find a message that corresponds to a given digest.
2. to find two different messages that produce the same digest.

Formal Definition

The National Institute of Standards and Technology (NIST, a non-regulatory agency of the US Dept of Commerce) has provided designs for what it calls the Secure Hash Standard (SHS FIPS) which includes the SHA-1 hash, SHA-2 family of hashes and the latest SHA-3 family of hashes.

The FIPS (Federal Information Processing Standards) are a set of standards and guidelines for federal computer systems that are developed by NIST in accordance with US based Information Security Acts. FIPS publication 180-2 (dated August 1, 2002), was the first to formally define the SHA-256 into the list of standards and has since been superseded. The latest publication specifying SHA-256 is the FIPS 180-4 released in August 2015.

SHA-2 (Secure Hash Algorithm 2), of which SHA-256 is a part, is known for its strength and speed. For a hash algorithm to be secure it is expected to have the following properties, all of which are seen in SHA-2.

- Scramble data **deterministic**ally (same input gives same output)
- Accept an input of arbitrary length and output a **fixed length result**.
- Manipulate data irreversibly (**OWF** - input cannot be derived from the output)
- Even if only one symbol is changed the algorithm will produce a (drastically) different hash value. (**Avalanche Effect**)
- Offer complete **collision resistance**.

SHA-256 Basic Information

Maximum input size : 2^{64} bits.

Digest size : 256 bits. (Usually represented in hex, so 64 hex characters)

Block size : 512 bits

So, there are 2^{256} possible SHA-256 digests, making it impossible to find out the right Input I for a digest S in an admissible period of time, even after combining all of the computational power we have worldwide.

The standard considers hashing byte-stream (or bit-stream) messages only. Hence, textual data which contains characters outside the ISO 8859-1 standards is encoded to UTF-8 before hashing it.

Throughout the specification, the “big-endian” convention is used when expressing data in words. All operations are modulo 2^{32}.

My (Simplified) SHA-256 Algorithm

The standard SHA-256 splits binary data into blocks of 512-bits each with padding and for each block mutates a set of working variables to get modified hash values. These are then carried over to the next block. The final 8 hash values are strung together to make the digest.

{This code limits data input to 1 block of size 512 bits.
In this, 64 bits are reserved for length,
8 bits contain the separator and some padding
and the remaining 440 bits contain the data.
Hence, data is restricted to 55 symbols (ASCII 55 bytes).}

INPUT HANDLING AND PADDING

1. Get string data as input, limited to 55 symbols in order to limit to 1 message block.
2. Convert this data into its binary equivalent.
3. Determine binary representation of the length of this binary data.
4. Add separator '1'.
5. Pad with 0s upto 448th bit.
6. Pad with string length for last 64 bits. (MD Strengthening)
7. Concatenate the 2 padded strings to get the final 512-bit message block.

INITIALISING ALL CONSTANTS

1. Find first 32 bits of fractional parts of square roots of first 8 prime numbers - Hash Values.

   ```
   hash_values[0] = int((math.modf(math.sqrt(2)))[0]*(1<<32))
   ```

   {Using math.sqrt(), Get square root of 2.
   Using math.modf(), split it into its fractional and integer parts as tuple(float, float).
   Get fractional part as 0-index of this tuple. This is still as a float fraction.
   Shift first 32 bits into the integer part by multiplying by 2^{32}.

Discarding the remaining fractional part by converting to int type.}

2. Find first 32 bits of fractional parts of cube roots of first 64 prime numbers - Round Constants.

```
round_constants[0] = int((math.modf(i))[0]*(1<<32))
```

Where 'i' = $2^{⅓}$ i.e., 'i' is the first element in a list of cube roots of the first 64 primes numbers.

SETTING UP PROCESSING FUNCTIONS AND MESSAGE SCHEDULE

1. Define the logical functions that operate on 32-bit words as follows:

```
SHR = lambda x, n: x >> n
SHL = lambda x, n: x << n
ROR = lambda x, n: SHR(x & 0xffffffff, n & 31) | SHL(x,
      (WORDSIZE-(n & 31))) & 0xffffffff
```

{SHR is a shift right operation on x by n bits and similarly SHL is the left shift.
ROR is right rotate of x by n bits where the word limit is set to 32 bits using 0xffffffff.}

2. The SHA-256 standard defines the following 6 logical functions for 32-bit words:

```
Ch =  lambda x, y, z: (x & y) ^ (~x & z)
Maj = lambda x, y, z: (x & y) ^ (x & z) ^ (y & z)
Sigma0 = lambda x: (ROR(x, 2) ^ ROR(x, 13) ^ ROR(x, 22))
Sigma1 = lambda x: (ROR(x, 6) ^ ROR(x, 11) ^ ROR(x, 25))
Gamma0 = lambda x: (ROR(x, 7) ^ ROR(x, 18) ^ SHR(x, 3))
Gamma1 = lambda x: (ROR(x, 17) ^ ROR(x, 19) ^ SHR(x, 10))
```

{The standard actually names the functions Sigma0 as Σ0, Sigma1 as Σ1, Gamma0 as σ0 and Gamma1 as σ1.}

3. Split padded data into 32-bit words to get 16 such words.
4. Initialize 48 more words to 0 having word size of 32-bits to bring total number of words to 64.
5. Determine actual values of the 17th to 64th words using the defined functions and adding using modulo 2^{32}:

```
words[t] = Gamma1(words[t-2]) + words[t-7] +
           Gamma0(words[t-15]) + words[t-16]
```

HASH COMPRESSION

1. Initialize the set of working variables {a, b,.. h} to the initial Hash Values as previously determined.

2. For each word in the message schedule, 2 temporary variables are created as temp1 and temp2 and the working set is overwritten with modulo 2^{32} addition:

```
temp1 = (h + Sigma1(e) + Ch(e ,f, g) + round_constants[t]
+ words[t]) % 2**32

temp2 = (Sigma0(a) + Maj(a, b, c)) % 2**32

h = g

g = f

f = e

e = (d + temp1) % 2**32

d = c

c = b

b = a

a = (temp1 + temp2) % 2**32
```

3. Finally, the Hash Values are rewritten by adding the values of the working variables modulo 2^{32}.

```
hash_values[0] = (a + hash_values[0]) % 2**32
```

4. The resulting 256-bit message digest is formed by concatenating the bit represented Hash Values as strings. The 64-character hex representation is formed by converting each hash value to its hexadecimal equivalent and stringing them together.

Analysing the Algorithm

Algorithm-wise, the SHA-2 family offers much higher diffusion and are less likely to be susceptible to the same class of attacks that are plaguing the MD5 and SHA-1 algorithms currently. This is because:

- they apply two parallel nonlinear functions (Ch and Maj) that ensure the function is one way and differences are hard to control. Without these functions, SHA-256 would be almost entirely linear and collisions would be trivial to find. Further Gamm0 and Gamm1 are used within the expansion phase to promote collision resistance.
- two more complicated diffusion primitives (Sigma0 and Sigma1). A single bit difference in the input will spread to 2 other bits in the output. This helps promote rapid diffusion through the design by being placed at just the right spot.
- and more feedback (into the working variables 'a' and 'e'). The output of Sigma0 and Sigma1 eventually feeds back into the same word of the state the input came from. This means that after 1 round, a single bit affects 3 bits of the neighbouring word. After 2 rounds, it affects at least 9 bits, and so on. This feedback scheme has so far proven very effective at making the designs immune to cryptanalysis.

Uses of SHA-256

- Challenge Handshake Authentication – a client can send the hash of a password over the internet for validation by a server without risk of the original password being intercepted.
- Digital Signatures – you can sign the hash of a document by encrypting it with your private key, producing a digital signature for the document. Anyone else can then check that you authenticated the text by decrypting the signature with your public key to obtain the original hash again, and comparing it with their hash of the text.
- Used in Network Encryption and Authentication protocols like SSL, TLS, SSH etc.
- It is one of the strongest hash functions available. It has not yet been compromised in any way. The 256-bit key makes it a good partner-function for AES.
- Bitcoin Mining – To mine bitcoins, you have to find an input that produces an SHA-256 output with 70-something zeros at the beginning. Since there is no known formula to do this, the only method used is a brute-force algorithm.

SAMPLE OUTPUT:

```
Enter data to hash (MAX 55 Symbols): abcdefg 123456 hjklm 98

PADDED DATA:-
011000010110001001100011011001000110010101100110011001110010000000110001001100100011001100110100001101010011011000100000011010000110101
001101011011011000110110100100000001110010011100010000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010111000
```

```
INITIAL HASH VALUES:-
 ['0x6a09e667' '0xbb67ae85' '0x3c6ef372' '0xa54ff53a' '0x510e527f'
 '0x9b05688c' '0x1f83d9ab' '0x5be0cd19']

ROUND CONSTANTS:-
 ['0x428a2f98' '0x71374491' '0xb5c0fbcf' '0xe9b5dba5' '0x3956c25b'
 '0x59f111f1' '0x923f82a4' '0xab1c5ed5' '0xd807aa98' '0x12835b01'
 '0x243185be' '0x550c7dc3' '0x72be5d74' '0x80deb1fe' '0x9bdc06a7'
 '0xc19bf174' '0xe49b69c1' '0xefbe4786' '0x0fc19dc6' '0x240ca1cc'
 '0x2de92c6f' '0x4a7484aa' '0x5cb0a9dc' '0x76f988da' '0x983e5152'
 '0xa831c66d' '0xb00327c8' '0xbf597fc7' '0xc6e00bf3' '0xd5a79147'
 '0x06ca6351' '0x14292967' '0x27b70a85' '0x2e1b2138' '0x4d2c6dfc'
 '0x53380d13' '0x650a7354' '0x766a0abb' '0x81c2c92e' '0x92722c85'
 '0xa2bfe8a1' '0xa81a664b' '0xc24b8b70' '0xc76c51a3' '0xd192e819'
 '0xd6990624' '0xf40e3585' '0x106aa070' '0x19a4c116' '0x1e376c08'
 '0x2748774c' '0x34b0bcb5' '0x391c0cb3' '0x4ed8aa4a' '0x5b9cca4f'
 '0x682e6ff3' '0x748f82ee' '0x78a5636f' '0x84c87814' '0x8cc70208'
 '0x90befffa' '0xa4506ceb' '0xbef9a3f7' '0xc67178f2']
```

```
MESSAGE SCHEDULE:-
 ['01100001011000100110001101100100' '01100101011001100110011100100000'
 '00110001001100100011001100110100' '00110101001101100010000001101000'
 '01101010011010110110110001101101' '00100000001110010011100010000000'
 '00000000000000000000000000000000' '00000000000000000000000000000000'
 '00000000000000000000000000000000' '00000000000000000000000000000000'
 '00000000000000000000000000000000' '00000000000000000000000000000000'
 '00000000000000000000000000000000' '00000000000000000000000000000000'
 '00000000000000000000000000000000' '00000000000000000000000010111000'
 '00110111000100000111110011010111' '01001000001110101001010101101100'
 '11000010000000101100100011010101' '01011001110000101010010111110110'
 '00101111101001000010001000000010' '00100110100011001100111111110000'
 '10010101010010101011101111100110' '00110101001000000011000101111011'
 '01010010111001010110000000001110' '11100000101001010100010111100001'
 '01110101110101010000000001101100' '00111010000110010000011100011001'
 '01000110101100111001000100000000' '00111000101010111111011111110100'
 '01011111111111111011100011111100' '10001111101010011111010100101010'
 '10110111111110000101010000101111' '10110001110100100101101111001011'
 '01101011110101001011001101011101' '00010000010100100101100100101000'
 '00001111100100001001000101110100' '01011111101111000000111011111010'
 '01111100000111001001111000110011' '11000010110010010000001101001100'
 '01110000001001100010110010100001' '10000100110000110101011100000110011'
 '11001110111100010101010110011011' '11111011100000110001111101101001'
 '00111001010101011110101000101000' '10111111010101110110110000010110'
 '10100100110010001101100111100111' '00010011101010000101110011010010'
 '11001100010011001100100011100111' '01000010001101011011100100010100'
 '00101001000101001100101011100110' '10000011000100001001100001101010'
 '11000111010110110110110001101001' '10110010100101010011010010101010'
 '00101100010111001000101101111111' '10010011110011110100110001001100'
 '00100111110101000101011001001010' '01111000100010001000101100001010'
 '11111111010001001010101100111101' '01000101101010101111011000001101'
 '10101001010101001101100101010000' '11111110100010110010100001000111'
 '11100001101001110001101010110010' '00010000101110100100100100011000']
```

```
SHA-256 HASH:-
 41b885198d2b3c688f8625474656ba5da4db2a8e7c23d1e129c9a3af6f4b55d1
```