Information Security & Cryptography

Nehal Jhajharia
Lab Assignment 6

Write a program to calculate the message digest of a text using the MD5 algorithm.

```
import math

# This list maintains the amount by which to rotate the buffers during processing
stage
rotate_by = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
            5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20,
            4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
            6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

# This list maintains the additive constant to be added in each processing step.
constants = [int(abs(math.sin(i+1)) * 4294967296) & 0xFFFFFFFF for i in range(64)]

# STEP 1: append padding bits s.t. the length is congruent to 448 modulo 512
# which is equivalent to saying 56 modulo 64.
# padding before adding the length of the original message is conventionally done as:
# pad a one followed by zeros to become congruent to 448 modulo 512(or 56 modulo 64).
def pad(msg):
    msg_len_in_bits = (8*len(msg)) & 0xffffffffffffffff
    msg.append(0x80)

    while len(msg)%64 != 56:
        msg.append(0)

# STEP 2: append a 64-bit version of the length of the length of the original message
# in the unlikely event that the length of the message is greater than 2^64,
# only the lower order 64 bits of the length are used.

# sys.byteorder -> 'little'
    msg += msg_len_in_bits.to_bytes(8, byteorder='little') # little endian convention
    # to_bytes(8...) will return the lower order 64 bits(8 bytes) of the length.
    return msg



# STEP 3: initialise message digest buffer.
# MD buffer is 4 words A, B, C and D each of 32-bits.

init_MDBuffer = [0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476]

# UTILITY/HELPER FUNCTION:
def leftRotate(x, amount):
    x &= 0xFFFFFFFF
    return (x << amount | x >> (32-amount)) & 0xFFFFFFFF



# STEP 4: process the message in 16-word blocks
# Message block stored in buffers is processed in the follg general manner:
# A = B + rotate left by some amount<-(A + func(B, C, D) + additive constant + 1 of
the 16 32-bit(4 byte) blocks converted to int form)

def processMessage(msg):
    init_temp = init_MDBuffer[:] # create copy of the buffer init constants to preserve
them for when message has multiple 512-bit blocks

    # message length is a multiple of 512bits, but the processing is to be done
separately for every 512-bit block.
    for offset in range(0, len(msg), 64):
        A, B, C, D = init_temp # have to initialise MD Buffer for every block
        block = msg[offset : offset+64] # create block to be processed
        # msg is processed as chunks of 16-words, hence, 16 such 32-bit chunks
        for i in range(64): # 1 pass through the loop processes some 32 bits out of the
512-bit block.
            if i < 16:
                # Round 1
                func = lambda b, c, d: (b & c) | (~b & d)
                # if b is true then ans is c, else d.
                index_func = lambda i: i

            elif i >= 16 and i < 32:
                # Round 2
                func = lambda b, c, d: (d & b) | (~d & c)
                # if d is true then ans is b, else c.
                index_func = lambda i: (5*i + 1)%16

            elif i >= 32 and i < 48:
                # Round 3
                func = lambda b, c, d: b ^ c ^ d
                # Parity of b, c, d
                index_func = lambda i: (3*i + 5)%16

            elif i >= 48 and i < 64:
                # Round 4
                func = lambda b, c, d: c ^ (b | ~d)
                index_func = lambda i: (7*i)%16

            F = func(B, C, D) # operate on MD Buffers B, C, D
            G = index_func(i) # select one of the 32-bit words from the 512-bit block
of the original message to operate on.

            to_rotate = A + F + constants[i] + int.from_bytes(block[4*G : 4*G + 4],
byteorder='little')
            newB = (B + leftRotate(to_rotate, rotate_by[i])) & 0xFFFFFFFF

            A, B, C, D = D, newB, B, C
            # rotate the contents of the 4 MD buffers by one every pass through the
loop

        # Add the final output of the above stage to initial buffer states
        for i, val in enumerate([A, B, C, D]):
            init_temp[i] += val
            init_temp[i] &= 0xFFFFFFFF
        # The init_temp list now holds the MD(in the form of the 4 buffers A, B, C, D)
of the 512-bit block of the message fed.


    # The same process is to be performed for every 512-bit block to get the final
MD(message digest).


    # Construct the final message from the final states of the MD Buffers
    return sum(buffer_content<<(32*i) for i, buffer_content in enumerate(init_temp))
```

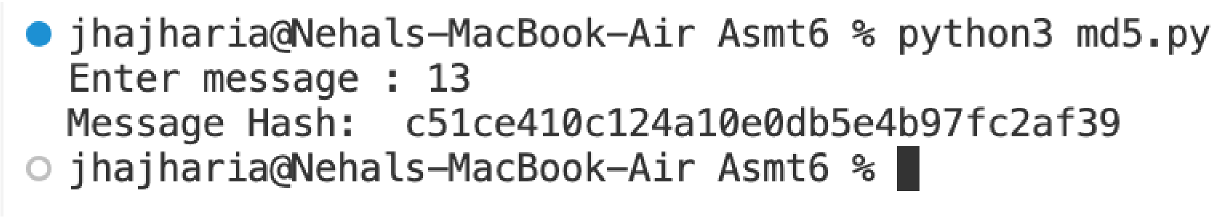
```python
def MD_to_hex(digest):
   # takes MD from the processing stage, change its endian-ness and return it as 128-bit hex hash
   raw = digest.to_bytes(16, byteorder='little')
   return '{:032x}'.format(int.from_bytes(raw, byteorder='big'))



def md5(msg):
   msg = bytearray(msg, 'ascii') # create a copy of the original message in form of a sequence of integers [0, 256)
   msg = pad(msg)
   processed_msg = processMessage(msg)
   # processed_msg contains the integer value of the hash
   message_hash = MD_to_hex(processed_msg)
   print("Message Hash: ", message_hash)




if __name__ == '__main__':
   message = input()
   md5(message)
```



```
● jhajharia@Nehals-MacBook-Air Asmt6 % python3 md5.py
  Enter message : 13
  Message Hash:  c51ce410c124a10e0db5e4b97fc2af39
○ jhajharia@Nehals-MacBook-Air Asmt6 % █
```