Name: **Mareena Fernandes**
Roll No.: **8669**
Class: **TE IT**
Batch: **B**

EXPERIMENT NO: 4

SHA1:

```
import sys
import io
import struct

# Predefined hex values for SHA1
h = (0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0)

# Input length in bytes
message_len = 0

# Extra leftover out of 64 sized chunk
extras = b""


def left_shift(num, bits):
   return ((num << bits | num >> (32 - bits))) & 0xFFFFFFFF


def process_chunk(chunk, h0, h1, h2, h3, h4):
   assert len(chunk) == 64
   w = [0] * 80

   # Break chunk into sixteen 4-byte big-endian words w[i]
   for i in range(16):
      w[i] = struct.unpack(b">I", chunk[i * 4 : i * 4 + 4])[0]

   # Extend the sixteen 4-byte words into eighty 4-byte words
   for i in range(16, 80):
      w[i] = left_shift(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1)

   # Initialize hash value for this chunk
```

```python
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    for i in range(80):
        if 0 <= i <= 19:
            f = d ^ (b & (c ^ d))
            k = 0x5A827999
        elif 20 <= i <= 39:
            f = b ^ c ^ d
            k = 0x6ED9EBA1
        elif 40 <= i <= 59:
            f = (b & c) | (b & d) | (c & d)
            k = 0x8F1BBCDC
        elif 60 <= i <= 79:
            f = b ^ c ^ d
            k = 0xCA62C1D6

        a, b, c, d, e = (
            (left_shift(a, 5) + f + e + k + w[i]) & 0xFFFFFFFF,
            a,
            left_shift(b, 30),
            c,
            d,
        )
    # Add this chunk's hash to result so far
    h0 = (h0 + a) & 0xFFFFFFFF
    h1 = (h1 + b) & 0xFFFFFFFF
    h2 = (h2 + c) & 0xFFFFFFFF
    h3 = (h3 + d) & 0xFFFFFFFF
    h4 = (h4 + e) & 0xFFFFFFFF

    return h0, h1, h2, h3, h4


def update_hash(message, message_len, extras, h):
    message_io_bytes = io.BytesIO(message)
```

```
    chunk = message_io_bytes.read(64)
    while len(chunk) == 64:
        message_len += 64
        h = process_chunk(chunk, *h)
        chunk = message_io_bytes.read(64)
    extras = chunk
    return message_len, extras, h


def hashed_bytes(message_len, extras, h):
    message_len += len(extras)
    message_len_bits = message_len * 8

    extras += b"\x80"
    extras += b"\x00" * int((56 - (message_len_bits / 8 + 1) % 64) % 64)
    extras += struct.pack(b">Q", message_len_bits)
    h = process_chunk(extras[:64], *h)
    if not len(extras) == 64:
        h = process_chunk(extras[64:], *h)
    return h[0], h[1], h[2], h[3], h[4]


def hexdigest(message_len, extras, h):
    h0, h1, h2, h3, h4 = hashed_bytes(message_len, extras, h)
    return "{0:08x}{1:08x}{2:08x}{3:08x}{4:08x}".format(h0, h1, h2, h3, h4)


message = input("Enter message to hash: ")
msg_b = message.encode("utf-8")
message_len, extras, h = update_hash(msg_b, message_len, extras, h)
sha_digest = hexdigest(message_len, extras, h)
print("sha1-digest:", sha_digest)
```

MD5:

```
"""
The implementation of the MD5 algorithm is based on the original RFC at
https://www.ietf.org/rfc/rfc1321.txt and contains optimizations from
https://en.wikipedia.org/wiki/MD5.
"""

import struct
from enum import Enum
from math import (
    floor,
    sin,
)

from bitarray import bitarray


class MD5Buffer(Enum):
    A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476


class MD5(object):
    _string = None
    _buffers = {
        MD5Buffer.A: None,
        MD5Buffer.B: None,
        MD5Buffer.C: None,
        MD5Buffer.D: None,
    }

    @classmethod
    def hash(cls, string):
        cls._string = string

        preprocessed_bit_array = cls._step_2(cls._step_1())
```

```
        cls._step_3()
        cls._step_4(preprocessed_bit_array)
        return cls._step_5()

    @classmethod
    def _step_1(cls):
        # Convert the string to a bit array.
        bit_array = bitarray(endian="big")
        bit_array.frombytes(cls._string.encode("utf-8"))

        # Pad the string with a 1 bit and as many 0 bits required such that
        # the length of the bit array becomes congruent to 448 modulo 512.
        # Note that padding is always performed, even if the string's bit
        # length is already conguent to 448 modulo 512, which leads to a
        # new 512-bit message block.
        bit_array.append(1)
        while bit_array.length() % 512 != 448:
            bit_array.append(0)

        # For the remainder of the MD5 algorithm, all values are in
        # little endian, so transform the bit array to little endian.
        return bitarray(bit_array, endian="little")

    @classmethod
    def _step_2(cls, step_1_result):
        # Extend the result from step 1 with a 64-bit little endian
        # representation of the original message length (modulo 2^64).
        length = (len(cls._string) * 8) % pow(2, 64)
        length_bit_array = bitarray(endian="little")
        length_bit_array.frombytes(struct.pack("<Q", length))

        result = step_1_result.copy()
        result.extend(length_bit_array)
        return result

    @classmethod
    def _step_3(cls):
        # Initialize the buffers to their default values.
```

```
    for buffer_type in cls._buffers.keys():
        cls._buffers[buffer_type] = buffer_type.value

@classmethod
def _step_4(cls, step_2_result):
    # Define the four auxiliary functions that produce one 32-bit word.
    F = lambda x, y, z: (x & y) | (~x & z)
    G = lambda x, y, z: (x & z) | (y & ~z)
    H = lambda x, y, z: x ^ y ^ z
    I = lambda x, y, z: y ^ (x | ~z)

    # Define the left rotation function, which rotates `x` left `n` bits.
    rotate_left = lambda x, n: (x << n) | (x >> (32 - n))

    # Define a function for modular addition.
    modular_add = lambda a, b: (a + b) % pow(2, 32)

    # Compute the T table from the sine function. Note that the
    # RFC starts at index 1, but we start at index 0.
    T = [floor(pow(2, 32) * abs(sin(i + 1))) for i in range(64)]

    # The total number of 32-bit words to process, N, is always a
    # multiple of 16.
    N = len(step_2_result) // 32

    # Process chunks of 512 bits.
    for chunk_index in range(N // 16):
        # Break the chunk into 16 words of 32 bits in list X.
        start = chunk_index * 512
        X = [
            step_2_result[start + (x * 32) : start + (x * 32) + 32]
            for x in range(16)
        ]

        # Convert the `bitarray` objects to integers.
        X = [int.from_bytes(word.tobytes(), byteorder="little") for word in X]

        # Make shorthands for the buffers A, B, C and D.
```

```python
    A = cls._buffers[MD5Buffer.A]
    B = cls._buffers[MD5Buffer.B]
    C = cls._buffers[MD5Buffer.C]
    D = cls._buffers[MD5Buffer.D]

    # Execute the four rounds with 16 operations each.
    for i in range(4 * 16):
        if 0 <= i <= 15:
            k = i
            s = [7, 12, 17, 22]
            temp = F(B, C, D)
        elif 16 <= i <= 31:
            k = ((5 * i) + 1) % 16
            s = [5, 9, 14, 20]
            temp = G(B, C, D)
        elif 32 <= i <= 47:
            k = ((3 * i) + 5) % 16
            s = [4, 11, 16, 23]
            temp = H(B, C, D)
        elif 48 <= i <= 63:
            k = (7 * i) % 16
            s = [6, 10, 15, 21]
            temp = I(B, C, D)

        # The MD5 algorithm uses modular addition. Note that we need a
        # temporary variable here. If we would put the result in `A`, then
        # the expression `A = D` below would overwrite it. We also cannot
        # move `A = D` lower because the original `D` would already have
        # been overwritten by the `D = C` expression.
        temp = modular_add(temp, X[k])
        temp = modular_add(temp, T[i])
        temp = modular_add(temp, A)
        temp = rotate_left(temp, s[i % 4])
        temp = modular_add(temp, B)

        # Swap the registers for the next operation.
        A = D
        D = C
```

```python
            C = B
            B = temp

        # Update the buffers with the results from this chunk.
        cls._buffers[MD5Buffer.A] = modular_add(cls._buffers[MD5Buffer.A], A)
        cls._buffers[MD5Buffer.B] = modular_add(cls._buffers[MD5Buffer.B], B)
        cls._buffers[MD5Buffer.C] = modular_add(cls._buffers[MD5Buffer.C], C)
        cls._buffers[MD5Buffer.D] = modular_add(cls._buffers[MD5Buffer.D], D)

    @classmethod
    def _step_5(cls):
        # Convert the buffers to little-endian.
        A = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.A]))[0]
        B = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.B]))[0]
        C = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.C]))[0]
        D = struct.unpack("<I", struct.pack(">I", cls._buffers[MD5Buffer.D]))[0]

        # Output the buffers in lower-case hexadecimal format.
        return (
            f"{format(A, '08x')}{format(B, '08x')}{format(C, '08x')}{format(D, '08x')}"
        )
```

Post labs:

List advantages and drawbacks of various password hashing functions (bcrypt and scrypt)

Ans:

- Hashing is a one-way cryptographic function generally used to store pass words and other sensitive data that don't need to be retrieved back from the hash generated,
- Bcrypt is a computation expensive and difficult algorithm designed specifically to store passwords. It takes an input (password) and after significant computations, produces a hash output. It has been tried and tested by security community having been around since long time. In recent years, specialized hardware created for super-fast computation (FPGA/ ASIC/ GPU) have been feared to be able to break bcrypt at scale.
- Scrypt is another hash function built on basis of bcrypt solving heavy computation only requirement by introducing heavy memory requirement which hampers ability of specialised hardware to guess passwords effectively increasing time required per hashing and comparing significantly. The drawback is its expensive to servers using this method and can be a limitation to have much server power is available to the actual application.