|  |  |
| --- | --- |
| **Course Title**: Cryptography and Network Security Lab | **Course Code**: BCSE309P |
| **Faculty**: Prof. Karthika V | Slot: L11-L12 |
| Regno: 21BCE1964 | Name: Aadityaa.N |

**Exercise-5: MD5**

**Code:**

import struct

from enum import Enum

from math import floor, sin

from bitarray import bitarray

# Define the four auxiliary functions that produce one 32-bit word.

def F(x, y, z):

    return x & y | ~x & z

def G(x, y, z):

    return x & z | y & ~z

def H(x, y, z):

    return x ^ y ^ z

def I(x, y, z):

    return y ^ (x | ~z)

def rotate\_left(x, n):

    return x << n | x >> 32 - n

def modular\_add(a, b):

    return (a + b) % pow(2, 32)

# T table

T = [floor(pow(2, 32) \* abs(sin(i + 1))) for i in range(64)]

class MD5:

    input\_string = None

    buffers = {

        "A": 0x67452301,

        "B": 0xEFCDAB89,

        "C": 0x98BADCFE,

        "D": 0x10325476,

    }

    def hash(self, string):

        self.input\_string = string

        # convert string to bit array for easier operation and add padding

        temp = self.step\_1()

        # Append length of message to end of data

        preprocessed\_bit\_array = self.step\_2(temp)

        self.step\_3(preprocessed\_bit\_array)

        return self.step\_4()

    def step\_1(self):

        bit\_array = bitarray(endian="big")

        bit\_array.frombytes(self.input\_string.encode("utf-8"))

        bit\_array.append(1)

        while len(bit\_array) % 512 != 448:

            bit\_array.append(0)

        # go back to littler endian for ease

        return bitarray(bit\_array, endian="little")

    def step\_2(self, step\_1\_result):

        # get length of message in bits

        length = (len(self.input\_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

    def step\_3(self, step\_2\_result):

        # The total number of 32-bit words to process

        N = len(step\_2\_result) // 32

        # Process each block

        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 to prevent errors in the F,G,H,I functions

            X = [int.from\_bytes(word.tobytes(), byteorder="little") for word in X]

            # Simplify

            A = self.buffers["A"]

            B = self.buffers["B"]

            C = self.buffers["C"]

            D = self.buffers["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)

                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)

                A = D

                D = C

                C = B

                B = temp

                print("Round", i+1, "A:", A, "B:", B, "C:", C, "D:", D)

            # Final Updated for this chunk

            self.buffers["A"] = modular\_add(self.buffers["A"], A)

            self.buffers["B"] = modular\_add(self.buffers["B"], B)

            self.buffers["C"] = modular\_add(self.buffers["C"], C)

            self.buffers["D"] = modular\_add(self.buffers["D"], D)

            print("Buffers:", self.buffers)

            print(f"Chunk {chunk\_index + 1} of {N // 512} done")

    def step\_4(self):

        # Convert the buffers to little-endian to make it easier

        A = struct.unpack("<I", struct.pack(">I", self.buffers["A"]))[0]

        B = struct.unpack("<I", struct.pack(">I", self.buffers["B"]))[0]

        C = struct.unpack("<I", struct.pack(">I", self.buffers["C"]))[0]

        D = struct.unpack("<I", struct.pack(">I", self.buffers["D"]))[0]

        # return all the blocks joined together

        return f"{format(A, '08x')}{format(B, '08x')}{format(C, '08x')}{format(D, '08x')}"

if \_\_name\_\_ == "\_\_main\_\_":

    print("Exactly 448 Bits: 56 characters")

    string1 = "This is aadityaa 21BCE1964 studying in VIT Chennai India"

    print("Input Message", string1)

    print("Hash:", MD5().hash(string1))

    print("\nLess than 448 Bits: 11 characters")

    string2 = "Aadityaa .N"

    print("Input Message", string2)

    print("Hash:", MD5().hash(string2))

    print("\nGreater than 448 bits: 118 characters")

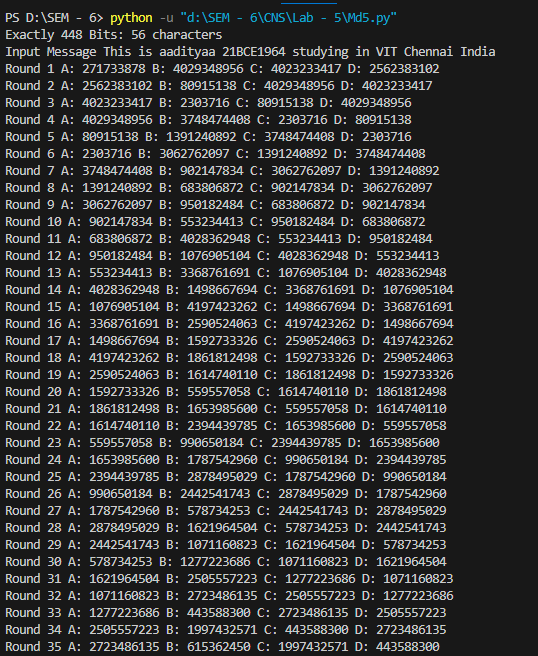
    string3 = "This is Aadityaa Nagarajan 21BCE1964 studying in vit chennai India i like programming and this is a wonderfull world ."

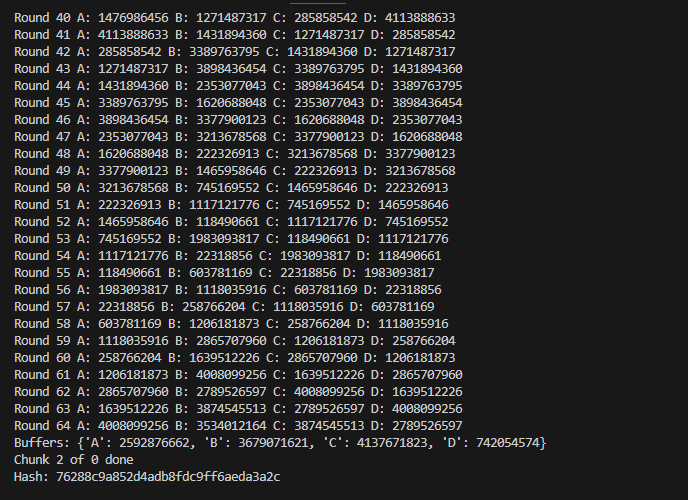
    print("Input Message", string3)

    print("Hash:", MD5().hash(string3))

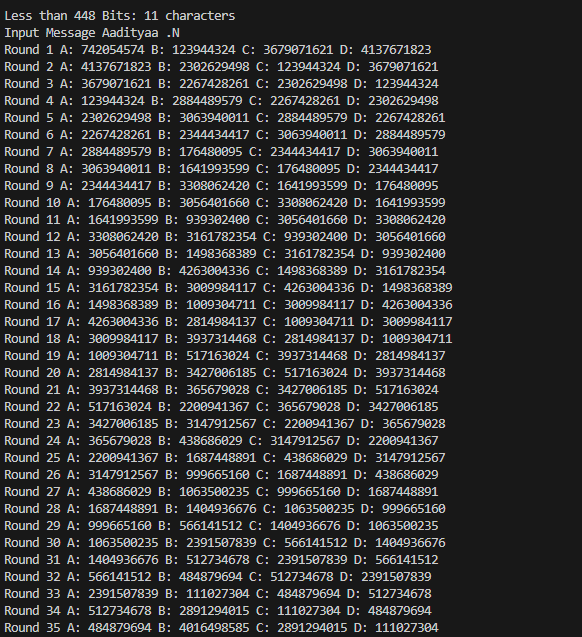
**Output1:**

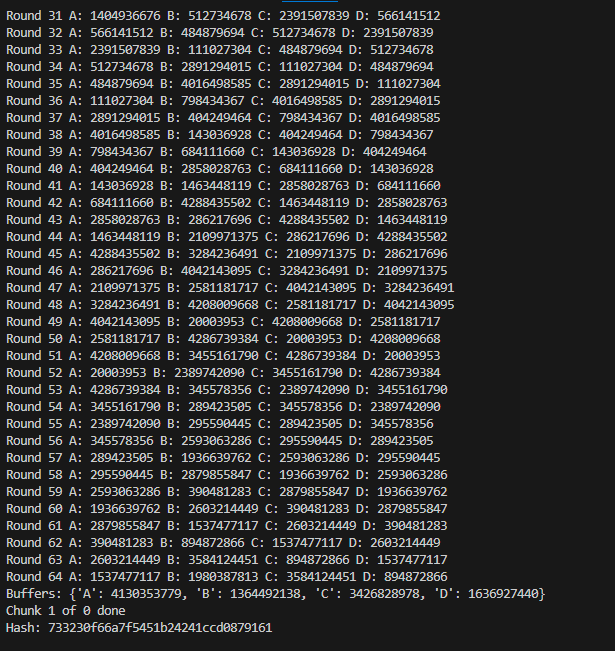


****

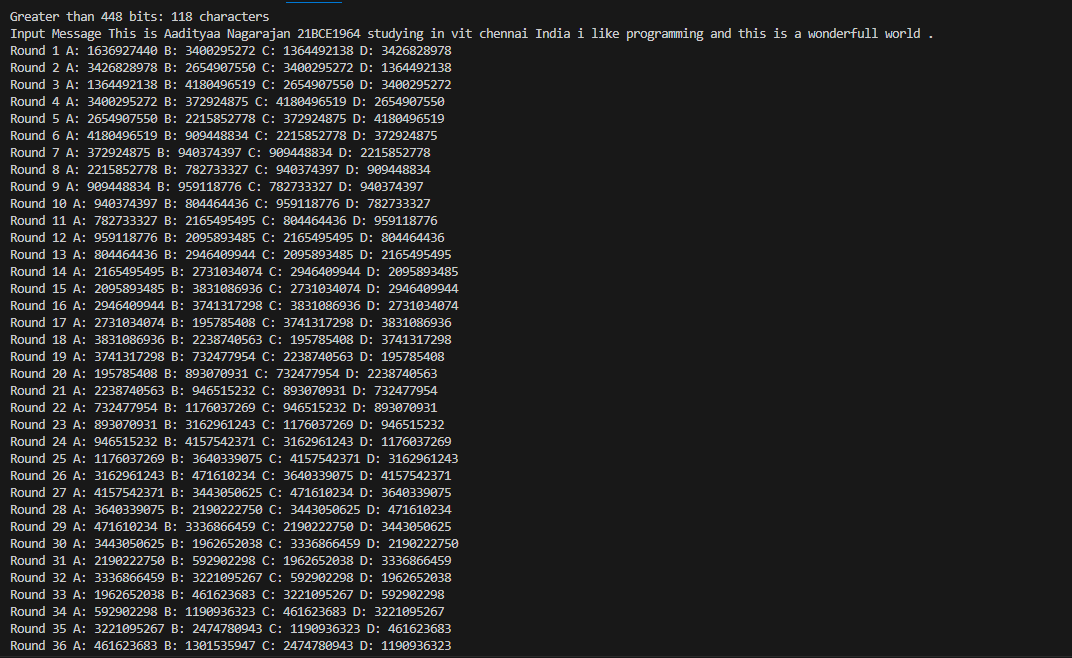
****

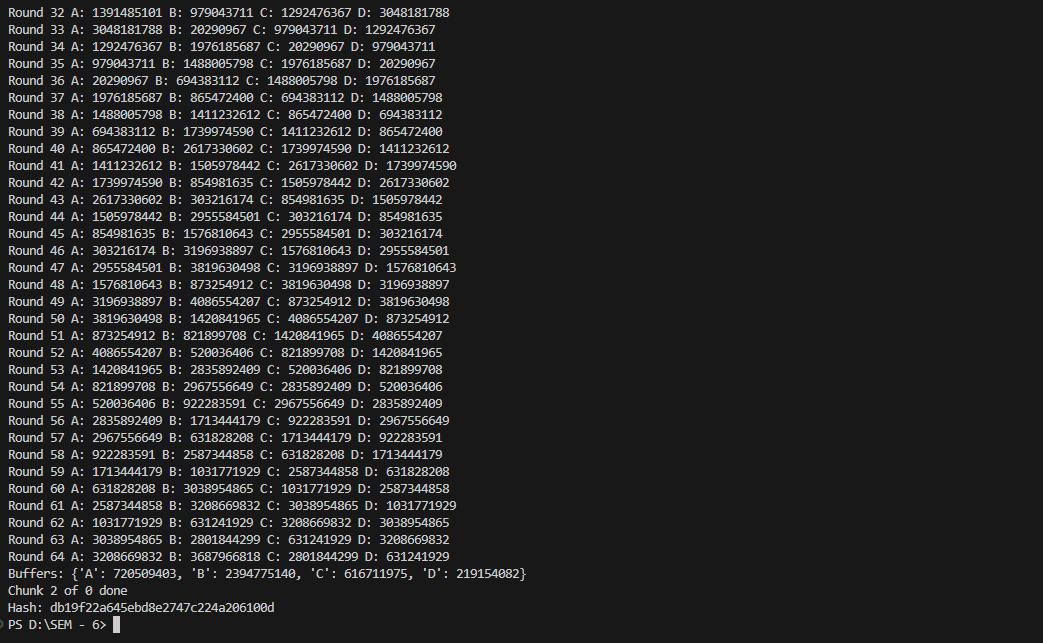
**Output2:**

****

****

**Output 3:**

****

****