Experiment No 4

Aim: Implement MD5 algorithm

Theory:

A cryptographic hash function is a hash function which takes an input (or 'message') and returns a fixed-size alphanumeric string. The string is called the 'hash value', 'message digest', 'digital fingerprint', 'digest' or 'checksum'.

The ideal hash function has three main properties:

1. It is extremely easy to calculate a hash for any given data.
2. It is extremely computationally difficult to calculate an alphanumeric text that has a given hash.
3. It is extremely unlikely that two slightly different messages will have the same hash.

Functions with these properties are used as hash functions for a variety of purposes, not only in cryptography. Practical applications include message integrity checks, digital signatures, authentication, and various information security applications.

A hash function takes a string of any length as input and produces a fixed length string which acts as a kind of "signature" for the data provided. In this way, a person knowing the "hash value" is unable to know the original message, but only the person who knows the original message can prove the "hash value" is created from that message.

A cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable. A cryptographic hash function is considered "insecure" from a cryptographic point of view, if either of the following is computationally feasible:

1. Finding a (previously unseen) message that matches a given hash values.
2. Finding "collisions", in which two different messages have the same hash value.

An attacker who can find any of the above computations can use them to substitute an authorized message with an unauthorized one.

Ideally, it should be impossible to find two different messages whose digests ("hash values") are similar. Also, one would not want an attacker to be able to learn anything useful about a message from its digest ("hash values"). Of course the attacker learns at least one piece of information, the digest itself, by which the attacker can recognize if the same message occurred again.

In various standards and applications, the two most commonly used hash functions are MD5 and SHA-1.

What is MD5?

The MD5 hashing algorithm is a one-way cryptographic function that accepts a message of any length as input and returns as output a fixed-length(128 bits) digest value to be used for authenticating the original message. The MD5 hash function was originally designed for use as a secure cryptographic hash algorithm for authenticating digital signatures. MD5 has been deprecated for uses other than as a non-cryptographic checksum to verify data integrity and detect unintentional data corruption. Although originally designed as a cryptographic message authentication code algorithm for use on the internet, MD5 hashing is no longer considered reliable for use as a cryptographic checksum because researchers have demonstrated techniques capable of easily generating MD5 collisions on commercial off-the-shelf computers. Ronald Rivest , founder of RSA Data Security and institute professor at MIT, designed MD5 as an improvement to a prior message digest algorithm, MD4. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA. MD5 is considered one of the most efficient algorithms currently available and being used widely today.

How MD5 works ?

The MD5 message digest hashing algorithm processes data in 512-bit blocks, broken down into 16 words composed of 32 bits each. The output from MD5 is a 128-bit message digest value.
Computation of the MD5 digest value is performed in separate stages that process each 512-bit block of data along with the value computed in the preceding stage. The first stage begins with the message digest values initialized using consecutive hexadecimal numerical values. Each stage includes four message digest passes which manipulate values in the current data block and values processed from the previous block. The final value computed from the last block becomes the MD5 digest for that block.

MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted A, B, C, and D. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of

four similar stages, termed rounds; each round is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation.

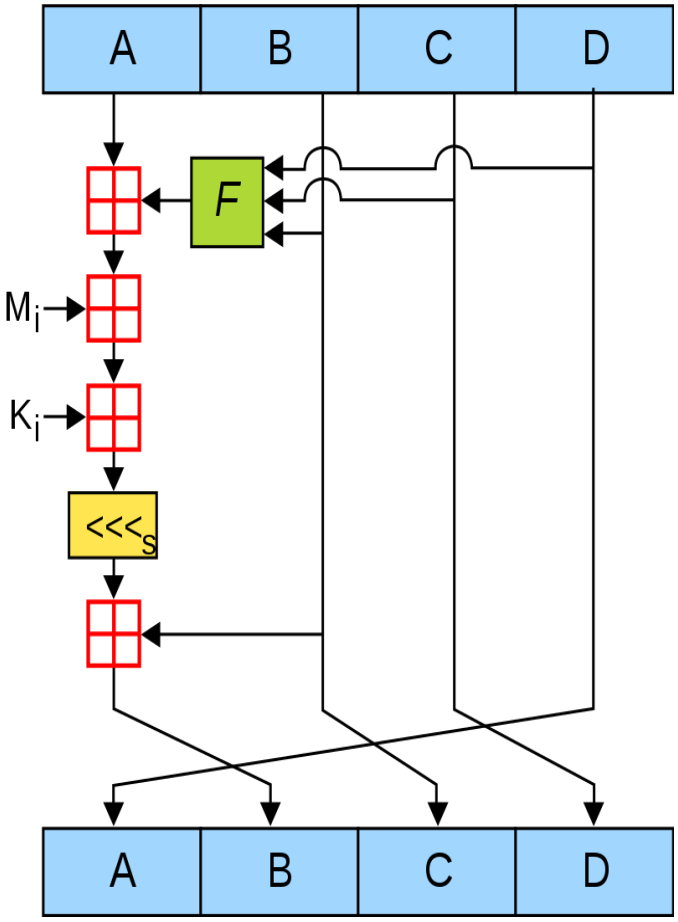

Figure 1. One MD5 operation.

As shown in Figure 1 MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. *F* is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $\lll_s$ denotes a left bit rotation by *s* places; *s* varies for each operation. $\boxplus$ denotes addition modulo 2^{32}.

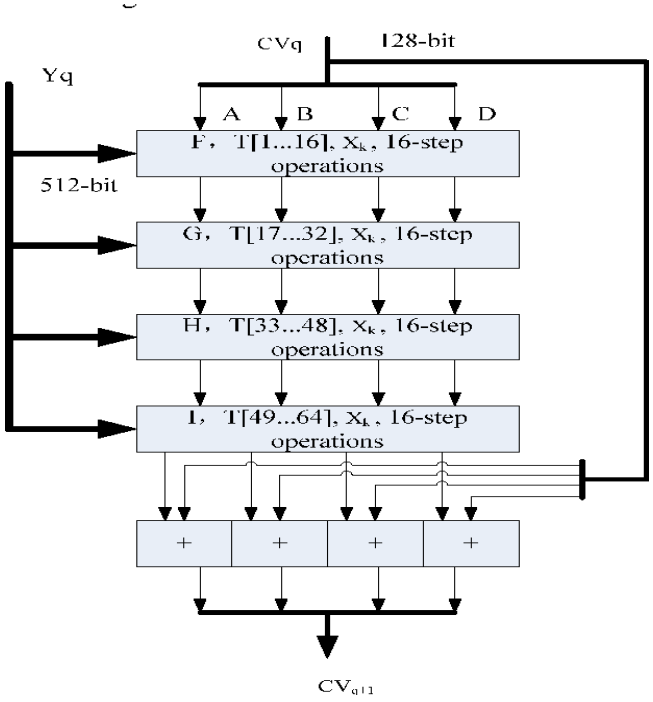

Figure 2:-One round of MD5 consisting of 64 steps.T[1...64] are the 64 constants and Xk is a part of the plain text.

As we see Figure 2 shows all 4 rounds each doing 16 iterations and how operations takes place on each block. We also define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word as shown in Figure 3.

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Y) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

Figure 3; Represents the Functions F,G,H and I

Weakness of MD5:-The MD5 algorithm is reported prone to a hash collision weakness. This weakness reportedly allows attackers to create multiple, differing input sources that, when the MD5 algorithm is used, result in the same output fingerprint.

It has been demonstrated that attackers can create multiple input sources to MD5 that result in the same output fingerprint. Reportedly, at this time, attackers cannot generate arbitrary

collisions. At this time, it is also reported that only a very limited number of individual bits in an input message may be altered while maintaining an identical output fingerprint.

This weakness may allow attackers to create two messages, or executable binaries such that their MD5 fingerprints are identical. One of these messages or binaries would be innocent, and the other malicious. The innocent message or binary may be digitally signed, and then later would have the malicious file substituted into its place. This attack may allow malicious code to be executed, or non-repudiation properties of messages to be broken. At this time, preimage attacks are not reportedly possible.

It is recommended that cryptosystems that utilize the MD5 algorithm should be reviewed, and the measures should be taken to protect against this weakness. Other hashing algorithms may possibly be utilized in replacement to, or in conjunction with MD5 to decrease the likelihood of a successful attack.

ALGORITHM:

1. Initialize the random left shift values s[], the 64 constants k[] and the values of a0,b0,c0 and d0.
2. Choose a plain text by selecting a random number between 1 and 2**512.
3. Determine the number of blocks to be used.
4. Add the padding bits to make the length of plain text as multiple of 512-64.
5. Add the 64 bits that represent the length of the message.
6. If there are multiple blocks, divide the plain text into multiple blocks of 512 bits and iterate over each 512 block.
7. Perform the MD5 algorithm of 64 steps for the 1st block.
8. If there are multiple blocks, use the final values of A,B,C and D as an input in the next round and repeat those 64 steps again.
9. Depending upon the number of iteration use the Function F,G,H,I which is represented by Fig 3 earlier.
10. Get the final values of A,B,C and D of 32 bit each,
11. Concatenate those and make it a 128 bit message digest and store it in md.
12. Print the final MD value.

Code:

```
from bitarray import bitarray
T=[
  3614090360  ,  3905402710  ,  606105819    , 3250441966   , 4118548399  ,
1200080426   , 2821735955  ,  4249261313 ,
  1770035416  ,  2336552879  ,  4294925233   , 2304563134   , 1804603682  ,
4254626195   , 2792965006  ,  1236535329,
  4129170786  ,  3225465664  ,  643717713    , 3921069994   , 3593408605  ,
38016083     , 3634488961  ,  3889429448,
  568446438   ,  3275163606  ,  4107603335   , 1163531501   , 2850285829  ,
4243563512   , 1735328473  ,  2368359562,
  4294588738  ,  2272392833  ,  1839030562   , 4259657740   , 2763975236  ,
1272893353   , 4139469664  ,  3200236656,
  681279174   ,  3936430074  ,  3572445317   , 76029189     , 3654602809  ,
3873151461   , 530742520   ,  3299628645,
  4096336452  ,  1126891415  ,  2878612391   , 4237533241   , 1700485571  ,
2399980690   , 4293915773  ,  2240044497,
  1873313359  ,  4264355552  ,  2734768916   , 1309151649   , 4149444226  ,
3174756917   , 718787259   ,  3951481745
]
def function_F(X,Y,Z):
  Xb=bitarray('{:032b}'.format(X))
  Yb=bitarray('{:032b}'.format(Y))
  Zb=bitarray('{:032b}'.format(Z))
  Ans=bitarray((Xb & Yb) | (~(Xb) & Zb ))
  return(int(Ans.to01(),2))

def function_G(X, Y, Z):
  Xb=bitarray('{:032b}'.format(X))
  Yb=bitarray('{:032b}'.format(Y))
  Zb=bitarray('{:032b}'.format(Z))
  Ans=bitarray((Xb & Zb) | (Yb & ~(Zb)))
  return(int(Ans.to01(),2))

def function_H(X, Y, Z):
  Xb=bitarray('{:032b}'.format(X))
  Yb=bitarray('{:032b}'.format(Y))
  Zb=bitarray('{:032b}'.format(Z))
  Ans=bitarray(Xb ^ Yb ^ Zb)
  return(int(Ans.to01(),2))
```

```
def function_I(X, Y, Z):
  Xb=bitarray('{:032b}'.format(X))
  Yb=bitarray('{:032b}'.format(Y))
  Zb=bitarray('{:032b}'.format(Z))
  Ans=bitarray(Yb ^ (Xb | ~(Zb)))
  return(int(Ans.to01(),2))



def analyze_block(block,ABCD):
    new_ABCD=ABCD.copy()
    M=[]
    t=0
    start = 0
    end = 32
    i=0
    k=0
    while start != len(block):
        M.append( int(block[start:end],2))
        start = end
        end += 32
        i+=1
    print("Message : ",M)
    #Round 1
    i=0
    s=7
    print("Round 1")
    for f in range(16):
        print((k+1),end=" ")
        print(new_ABCD)
        a=new_ABCD[0]
        b=new_ABCD[1]
        c=new_ABCD[2]
        d=new_ABCD[3]
        temp=(a+function_F(b,c,d)+M[i]+T[k])%(2**32)
        a=(b+int((format(temp, '032b')+format(temp, '032b')[0:s])[s:],2))%(2**32)
        new_ABCD[1]=new_ABCD[0]
        new_ABCD[2]=b
        new_ABCD[3]=c
        new_ABCD[0]=a
        i+=1
        s=7 if s==22 else s+5
        k+=1
    #Round 2
```

```
    i = 1
    s = 5
    print("Round 2")
    for g in range(16):
        print((k+1),end=" ")
        print(new_ABCD)
        a = new_ABCD[0]
        b = new_ABCD[1]
        c = new_ABCD[2]
        d = new_ABCD[3]
        temp = (a + function_G(b, c, d) + M[i] + T[k])%(2**32)
        a = (b + int((format(temp, '032b') + format(temp, '032b')[0:s])[s:],
2))%(2**32)
        new_ABCD[1] = new_ABCD[0]
        new_ABCD[2] = b
        new_ABCD[3] = c
        new_ABCD[0] = a

        i =(i+5)%16
        s = 5 if s == 20 else (9 if s==5 else (14 if s==9 else 20))
        k += 1
    #Round 3
    i = 5
    s = 4
    print("Round 3")
    for h in range(16):
        print((k+1),end=" ")
        print(new_ABCD)
        a = new_ABCD[0]
        b = new_ABCD[1]
        c = new_ABCD[2]
        d = new_ABCD[3]
        temp = (a + function_H(b, c, d) + M[i] + T[k])%(2**32)
        a = (b + int((format(temp, '032b') + format(temp, '032b')[0:s])[s:],
2))%(2**32)
        new_ABCD[1] = new_ABCD[0]
        new_ABCD[2] = b
        new_ABCD[3] = c
        new_ABCD[0] = a
        i = (i + 3) % 16
        s = 4 if s == 23 else (11 if s == 4 else (16 if s == 11 else 23))
        k += 1
    # Round 4
    i = 0
    s = 6
```

```
    print("Round 4")
    for m in range(16):
        print((k+1),end=" ")
        print(new_ABCD)
        a = new_ABCD[0]
        b = new_ABCD[1]
        c = new_ABCD[2]
        d = new_ABCD[3]
        temp = (a + function_I(b, c, d) + M[i] + T[k])%(2**32)
        a = (b + int((format(temp, '032b') + format(temp, '032b')[0:s])[s:],
2))%(2**32)
        new_ABCD[1] = new_ABCD[0]
        new_ABCD[2] = b
        new_ABCD[3] = c
        new_ABCD[0] = a
        i = (i + 7) % 16
        s = 6 if s == 21 else (10 if s == 6 else (15 if s == 10 else 21))
        k += 1
    latest_ABCD=[0,0,0,0]
    print("Old ABCD ",ABCD)
    print("New ABCD ",new_ABCD)
    for i in range(4):
        latest_ABCD[i]=(ABCD[i]+new_ABCD[i])%(2**32)
    print("ABCD to next block ",latest_ABCD)
    return latest_ABCD

message=input("Enter your message : ").strip()
plain_text=str(message)
textnum=""
for c in plain_text:
    textnum = textnum + str(ord(c))
print(plain_text)
text_num = int(textnum)
print(text_num)
bin_text="{0:b}".format(text_num)
print(bin_text)
print(len(bin_text))
length=len(bin_text)
padding=1
while True:
    if (length + padding) % 512 == 448 % 512:
        print("Padding bits are : ", padding)
        break
    padding += 1
padding_str=''.join("0" for x in range(padding))
```

```
padding_str="1"+padding_str[1:]
bin_text=bin_text+padding_str

print(bin_text)
print(len(bin_text)%512)
length_bin="{0:b}".format(length)

if len(length_bin)>64:
    length_bin = "{0:b}".format(length%(2**64))

if len(length_bin)==64:
    bin_text+=length_bin
elif len(length_bin)<64:
    length_bin=''.join("0" for x in range(64-len(length_bin)))+length_bin
    bin_text+=length_bin

print(bin_text)
print(len(bin_text))

ABCD=[1732584193, 4023233417, 2562383102, 271733878]
start=0
end=512
l=0
while start!=len(bin_text):
    block=bin_text[start:end]
    print("Block ",(l+1))
    print(block)
    l=l+1
    ABCD=analyze_block(block,ABCD)
    start=end
    end+=512
print(ABCD)
hashcode="0x"
for item in ABCD:
  hashcode+=hex(item).replace('0x','')
print(hashcode)
```

Output:

```
# Python 3.6.1 (default, Dec 2015, 13:05:11)
# [GCC 4.8.2] on linux

# Repl.it: Installing fresh packages
# Repl.it:
```

```
# Collecting bitarray
#   Downloading
https://files.pythonhosted.org/packages/e2/1e/b93636ae36d08d0ee3aec40b08731cc9721
7c69db9422c0afef6ee32ebd2/bitarray-0.8.3.tar.gz
# Building wheels for collected packages: bitarray
#   Running setup.py bdist_wheel for bitarray: started
#   Running setup.py bdist_wheel for bitarray: finished with status 'done'
#   Stored in directory:
/home/runner/.cache/pip/wheels/5f/33/3c/1370189fedb8fbb2c3297388cc8cdbe2af0efb489
41cf8d526
# Successfully built bitarray
# Installing collected packages: bitarray
# Successfully installed bitarray-0.8.3
# You are using pip version 9.0.1, however version 19.0.3 is available.
# You should consider upgrading via the 'pip install --upgrade pip' command.


# Repl.it: package installation success

# Enter your message : hi guys
# hi guys
# 10410532103117121115
# 1001000001111001101000111100101001011001000110001010011001011011011
# 64
# Padding bits are :  384
#
1001000001111001101000111100101001011001000110001010011001011011100000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000
# 448
#
1001000001111001101000111100101001011001000110001010011001011011100000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000001000000
# 512
# Block  1
```

#
10010000011110011010001111001010010110010001100010100110010110111000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000001000000
Message : [2423890890, 1494787675, 2147483648, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0,
0, 64]
Round 1
1 [1732584193, 4023233417, 2562383102, 271733878]
2 [3790720060, 1732584193, 4023233417, 2562383102]
3 [2313413950, 3790720060, 1732584193, 4023233417]
4 [3576104670, 2313413950, 3790720060, 1732584193]
5 [2353240037, 3576104670, 2313413950, 3790720060]
6 [2933977071, 2353240037, 3576104670, 2313413950]
7 [1309872037, 2933977071, 2353240037, 3576104670]
8 [1782355716, 1309872037, 2933977071, 2353240037]
9 [1221623959, 1782355716, 1309872037, 2933977071]
10 [4211164740, 1221623959, 1782355716, 1309872037]
11 [3852598748, 4211164740, 1221623959, 1782355716]
12 [3491996159, 3852598748, 4211164740, 1221623959]
13 [1722987083, 3491996159, 3852598748, 4211164740]
14 [2188986077, 1722987083, 3491996159, 3852598748]
15 [3239552606, 2188986077, 1722987083, 3491996159]
16 [2636692068, 3239552606, 2188986077, 1722987083]
Round 2
17 [163252692, 2636692068, 3239552606, 2188986077]
18 [3778524487, 163252692, 2636692068, 3239552606]
19 [1891434833, 3778524487, 163252692, 2636692068]
20 [2761243514, 1891434833, 3778524487, 163252692]
21 [886897030, 2761243514, 1891434833, 3778524487]
22 [431792689, 886897030, 2761243514, 1891434833]
23 [1858098983, 431792689, 886897030, 2761243514]
24 [3789797440, 1858098983, 431792689, 886897030]
25 [3804221607, 3789797440, 1858098983, 431792689]
26 [1916152493, 3804221607, 3789797440, 1858098983]
27 [85304537, 1916152493, 3804221607, 3789797440]
28 [3103261111, 85304537, 1916152493, 3804221607]
29 [400984917, 3103261111, 85304537, 1916152493]
30 [2557946805, 400984917, 3103261111, 85304537]
31 [3196638459, 2557946805, 400984917, 3103261111]
32 [829005072, 3196638459, 2557946805, 400984917]
Round 3

```
# 33 [1732159315, 829005072, 3196638459, 2557946805]
# 34 [509463623, 1732159315, 829005072, 3196638459]
# 35 [3838487496, 509463623, 1732159315, 829005072]
# 36 [558655162, 3838487496, 509463623, 1732159315]
# 37 [3056146807, 558655162, 3838487496, 509463623]
# 38 [520559474, 3056146807, 558655162, 3838487496]
# 39 [1679958123, 520559474, 3056146807, 558655162]
# 40 [4019322545, 1679958123, 520559474, 3056146807]
# 41 [736087897, 4019322545, 1679958123, 520559474]
# 42 [2190687023, 736087897, 4019322545, 1679958123]
# 43 [4238574665, 2190687023, 736087897, 4019322545]
# 44 [1109939102, 4238574665, 2190687023, 736087897]
# 45 [1844565049, 1109939102, 4238574665, 2190687023]
# 46 [2149673542, 1844565049, 1109939102, 4238574665]
# 47 [640851468, 2149673542, 1844565049, 1109939102]
# 48 [2303156244, 640851468, 2149673542, 1844565049]
# Round 4
# 49 [2625603831, 2303156244, 640851468, 2149673542]
# 50 [1295388114, 2625603831, 2303156244, 640851468]
# 51 [48020619, 1295388114, 2625603831, 2303156244]
# 52 [1221940970, 48020619, 1295388114, 2625603831]
# 53 [845891336, 1221940970, 48020619, 1295388114]
# 54 [986718925, 845891336, 1221940970, 48020619]
# 55 [2658082062, 986718925, 845891336, 1221940970]
# 56 [3120576709, 2658082062, 986718925, 845891336]
# 57 [585029289, 3120576709, 2658082062, 986718925]
# 58 [2886579521, 585029289, 3120576709, 2658082062]
# 59 [2290699449, 2886579521, 585029289, 3120576709]
# 60 [2229121686, 2290699449, 2886579521, 585029289]
# 61 [657579532, 2229121686, 2290699449, 2886579521]
# 62 [65165301, 657579532, 2229121686, 2290699449]
# 63 [1167190239, 65165301, 657579532, 2229121686]
# 64 [2728918675, 1167190239, 65165301, 657579532]
# Old ABCD  [1732584193, 4023233417, 2562383102, 271733878]
# New ABCD  [1264547441, 2728918675, 1167190239, 65165301]
# ABCD to next block  [2997131634, 2457184796, 3729573341, 336899179]
# [2997131634, 2457184796, 3729573341, 336899179]
# 0xb2a499729275aa1cde4cc5dd1414ac6b
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Enter your message : hi guys how are you guys hope you all are doing well and i
am great here
hi guys how are you guys hope you all are doing well and i am great here
```

1041053210311712111532104111119329711410132121111117321031171211153210411111121013
2121111117329710810832971141013210011110511010332119101108108329711010032105329 71
093210311410197116321041011141 01

1001001000001000110110011001011101111111111010100110110000100100000110001111100111
1000001011101111110010111111010111111000110011000110101100001001011001110110001101
1100001101101000100101001000001111001110010000100101001101010111101010111011111101
1011110001011001000111011110001101001011101001000001000000100000100000010110010001
1000011110001100100001010011010111010010010110010111111110100011000101110110100 10
0000101101101110001110111100010010100101111110100010111000111101011101000011100 11
1010101111001101100011110100110010101110110011001100010000111011000011001110110 10
1101100100101110011110100100011110000111110100010010100000111110000111101 01
642
Padding bits are : 318
1001001000001000110110011001011101111111111010100110110000100100000110001111100111
1000001011101111110010111111010111111000110011000110101100001001011001110110001101
1100001101101000100101001000001111001110010000100101001101010111101010111011111101
1011110001011001000111011110001101001011101001000001000000100000100000010110010001
1000011110001100100001010011010111010010010110010111111110100011000101110110100 10
0000101101101110001110111100010010100101111110100010111000111101011101000011100 11
1010101111001101100011110100110010101110110011001100010000111011000011001110110 10
1101100100101110011110100100011110000111110100010010100000111110000111101011000 00
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
448
1001001000001000110110011001011101111111111010100110110000100100000110001111100111
1000001011101111110010111111010111111000110011000110101100001001011001110110001101
1100001101101000100101001000001111001110010000100101001101010111101010111011111101
1011110001011001000111011110001101001011101001000001000000100000100000010110010001
1000011110001100100001010011010111010010010110010111111110100011000101110110100 10
0000101101101110001110111100010010100101111110100010111000111101011101000011100 11
1010101111001101100011110100110010101110110011001100010000111011000011001110110 10
1101100100101110011110100100011110000111110100010010100000111110000111101011000 00
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000001010000010
1024
Block 1
1001001000001000110110011001011101111111111010100110110000100100000110001111100111
1000001011101111110010111111010111111000110011000110101100001001011001110110001101
1100001101101000100101001000001111001110010000100101001101010111101010111011111101
1011110001011001000111011110001101001011101001000001000000100000100000010110010001

10000111100011001000010100110101110100100101100101111111101000110001011101101001000001011011011100011101111000100101001011111101000101110001111010111010000111001110101011110011011000111101
Message : [2450053527, 2144655432, 838058359, 3421894860, 1795778403, 1893344544, 4086338773, 3941578635, 599550324, 2181304409, 410568787, 1562744826, 829853787, 1910383919, 3513904033, 3467589181]
Round 1
1 [1732584193, 4023233417, 2562383102, 271733878]
2 [2844570301, 1732584193, 4023233417, 2562383102]
3 [4218078243, 2844570301, 1732584193, 4023233417]
4 [3967427653, 4218078243, 2844570301, 1732584193]
5 [3279196934, 3967427653, 4218078243, 2844570301]
6 [3771370699, 3279196934, 3967427653, 4218078243]
7 [792214560, 3771370699, 3279196934, 3967427653]
8 [1323809694, 792214560, 3771370699, 3279196934]
9 [4215150277, 1323809694, 792214560, 3771370699]
10 [2693948601, 4215150277, 1323809694, 792214560]
11 [364148357, 2693948601, 4215150277, 1323809694]
12 [347250177, 364148357, 2693948601, 4215150277]
13 [3027102305, 347250177, 364148357, 2693948601]
14 [657713540, 3027102305, 347250177, 364148357]
15 [1717216550, 657713540, 3027102305, 347250177]
16 [3420458835, 1717216550, 657713540, 3027102305]
Round 2
17 [2751546356, 3420458835, 1717216550, 657713540]
18 [2003429214, 2751546356, 3420458835, 1717216550]
19 [1517753765, 2003429214, 2751546356, 3420458835]
20 [2338551989, 1517753765, 2003429214, 2751546356]
21 [1421655664, 2338551989, 1517753765, 2003429214]
22 [1930431753, 1421655664, 2338551989, 1517753765]
23 [1468562734, 1930431753, 1421655664, 2338551989]
24 [4246430374, 1468562734, 1930431753, 1421655664]
25 [2268394814, 4246430374, 1468562734, 1930431753]
26 [327407386, 2268394814, 4246430374, 1468562734]
27 [129594348, 327407386, 2268394814, 4246430374]
28 [668618245, 129594348, 327407386, 2268394814]
29 [14941743, 668618245, 129594348, 327407386]
30 [3018372841, 14941743, 668618245, 129594348]
31 [1612253238, 3018372841, 14941743, 668618245]
32 [303705217, 1612253238, 3018372841, 14941743]
Round 3
33 [2229355639, 303705217, 1612253238, 3018372841]
34 [2351792124, 2229355639, 303705217, 1612253238]
35 [1994837480, 2351792124, 2229355639, 303705217]
36 [1782307963, 1994837480, 2351792124, 2229355639]

```
37 [3166874851, 1782307963, 1994837480, 2351792124]
38 [2393487970, 3166874851, 1782307963, 1994837480]
39 [1245291536, 2393487970, 3166874851, 1782307963]
40 [10436247, 1245291536, 2393487970, 3166874851]
41 [1071757642, 10436247, 1245291536, 2393487970]
42 [3880198368, 1071757642, 10436247, 1245291536]
43 [34844691, 3880198368, 1071757642, 10436247]
44 [1743297554, 34844691, 3880198368, 1071757642]
45 [3568372558, 1743297554, 34844691, 3880198368]
46 [2621974573, 3568372558, 1743297554, 34844691]
47 [4169068157, 2621974573, 3568372558, 1743297554]
48 [4251663887, 4169068157, 2621974573, 3568372558]
Round 4
49 [2102474766, 4251663887, 4169068157, 2621974573]
50 [3195838673, 2102474766, 4251663887, 4169068157]
51 [3268348363, 3195838673, 2102474766, 4251663887]
52 [939893394, 3268348363, 3195838673, 2102474766]
53 [2775796389, 939893394, 3268348363, 3195838673]
54 [462450895, 2775796389, 939893394, 3268348363]
55 [3418136213, 462450895, 2775796389, 939893394]
56 [1692934182, 3418136213, 462450895, 2775796389]
57 [1973546073, 1692934182, 3418136213, 462450895]
58 [2070515700, 1973546073, 1692934182, 3418136213]
59 [3146491326, 2070515700, 1973546073, 1692934182]
60 [1410302065, 3146491326, 2070515700, 1973546073]
61 [2902866747, 1410302065, 3146491326, 2070515700]
62 [4058855888, 2902866747, 1410302065, 3146491326]
63 [3422708306, 4058855888, 2902866747, 1410302065]
64 [2409757108, 3422708306, 4058855888, 2902866747]
Old ABCD  [1732584193, 4023233417, 2562383102, 271733878]
New ABCD  [1619520201, 2409757108, 3422708306, 4058855888]
ABCD to next block  [3352104394, 2138023229, 1690124112, 35622470]
Block  2
00110010101110110011001100010000111011000011001110110101101100100101110011110100
10001111000011111010001001010000011111000011110101100000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000001010000010
Message :  [851129104, 3962811826, 1559531279, 2723183677, 1610612736, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 642]
Round 1
1 [3352104394, 2138023229, 1690124112, 35622470]
2 [2541352792, 3352104394, 2138023229, 1690124112]
```

3 [4051999497, 2541352792, 3352104394, 2138023229]
4 [3627727657, 4051999497, 2541352792, 3352104394]
5 [2033811376, 3627727657, 4051999497, 2541352792]
6 [3555858299, 2033811376, 3627727657, 4051999497]
7 [3804655347, 3555858299, 2033811376, 3627727657]
8 [3160399833, 3804655347, 3555858299, 2033811376]
9 [4130068310, 3160399833, 3804655347, 3555858299]
10 [2243915898, 4130068310, 3160399833, 3804655347]
11 [1018004403, 2243915898, 4130068310, 3160399833]
12 [70214315, 1018004403, 2243915898, 4130068310]
13 [4102197910, 70214315, 1018004403, 2243915898]
14 [3960125597, 4102197910, 70214315, 1018004403]
15 [392155635, 3960125597, 4102197910, 70214315]
16 [3162111134, 392155635, 3960125597, 4102197910]
Round 2
17 [2367324185, 3162111134, 392155635, 3960125597]
18 [2512691715, 2367324185, 3162111134, 392155635]
19 [220020767, 2512691715, 2367324185, 3162111134]
20 [610374706, 220020767, 2512691715, 2367324185]
21 [496184090, 610374706, 220020767, 2512691715]
22 [669058770, 496184090, 610374706, 220020767]
23 [1257368009, 669058770, 496184090, 610374706]
24 [3898253107, 1257368009, 669058770, 496184090]
25 [3609697221, 3898253107, 1257368009, 669058770]
26 [329627647, 3609697221, 3898253107, 1257368009]
27 [3890687287, 329627647, 3609697221, 3898253107]
28 [2808241547, 3890687287, 329627647, 3609697221]
29 [3814880112, 2808241547, 3890687287, 329627647]
30 [1014078873, 3814880112, 2808241547, 3890687287]
31 [1407657059, 1014078873, 3814880112, 2808241547]
32 [3594446214, 1407657059, 1014078873, 3814880112]
Round 3
33 [1132679728, 3594446214, 1407657059, 1014078873]
34 [2828887157, 1132679728, 3594446214, 1407657059]
35 [2465837539, 2828887157, 1132679728, 3594446214]
36 [2405802540, 2465837539, 2828887157, 1132679728]
37 [2180860691, 2405802540, 2465837539, 2828887157]
38 [242994792, 2180860691, 2405802540, 2465837539]
39 [1639232968, 242994792, 2180860691, 2405802540]
40 [922429371, 1639232968, 242994792, 2180860691]
41 [3508972626, 922429371, 1639232968, 242994792]
42 [1717803760, 3508972626, 922429371, 1639232968]
43 [449283235, 1717803760, 3508972626, 922429371]
44 [2833362613, 449283235, 1717803760, 3508972626]
45 [4168415969, 2833362613, 449283235, 1717803760]

```
46 [299462335, 4168415969, 2833362613, 449283235]
47 [927061241, 299462335, 4168415969, 2833362613]
48 [2704803567, 927061241, 299462335, 4168415969]
Round 4
49 [3163731855, 2704803567, 927061241, 299462335]
50 [2664170845, 3163731855, 2704803567, 927061241]
51 [3012298892, 2664170845, 3163731855, 2704803567]
52 [1515120155, 3012298892, 2664170845, 3163731855]
53 [843352326, 1515120155, 3012298892, 2664170845]
54 [2135238195, 843352326, 1515120155, 3012298892]
55 [2691056729, 2135238195, 843352326, 1515120155]
56 [1999166774, 2691056729, 2135238195, 843352326]
57 [3503667137, 1999166774, 2691056729, 2135238195]
58 [2380057372, 3503667137, 1999166774, 2691056729]
59 [3442668692, 2380057372, 3503667137, 1999166774]
60 [3619631547, 3442668692, 2380057372, 3503667137]
61 [755239491, 3619631547, 3442668692, 2380057372]
62 [2433859306, 755239491, 3619631547, 3442668692]
63 [682870557, 2433859306, 755239491, 3619631547]
64 [2681987413, 682870557, 2433859306, 755239491]
Old ABCD  [3352104394, 2138023229, 1690124112, 35622470]
New ABCD  [4034048188, 2681987413, 682870557, 2433859306]
ABCD to next block  [3091185286, 525043346, 2372994669, 2469481776]
[3091185286, 525043346, 2372994669, 2469481776]
0xb83fbe861f4b86928d71066d93314d30
```

Conclusion: Thus, I learned about MD5 algorithm and thus, was able to understand and implement in python. We can use this same algorithm in our applications for authenticating of original message. MD5 is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length.