In [1]:
```
#Ning Tientso
#25-Nov-2019
#SEC TP2 - RSA and SHA256

import numpy as np
import math
import random
import time
import struct
```

In [2]:
```
def Fast_Exponentiation(a, k, n):
    '''
    compute a**k mod n for big values k and n.
    '''

    #compute every exponent that is a power of 2
    x = a
    y = a if (k%2==1) else 1 #confirm that k is a power of 2, otherwise start
 at 1
    k_ = math.floor(k/2)

    while k_ > 0: #run while there is still powers to compute
        #x = x**2 % n
        x = pow(x,2,n)
        if (k_%2==1):
            y = x if y==1 else y*x%n #we either want y=a**n_i-1 or y=a**n_i
        k_ = math.floor(k_/2) #running count

    return y
```

In [3]:
```
def Faster_Exponentiation(a, k, n):
    '''
    The fast exponentiation implementation is above to show understanding,
    But due to uncertainty for which part is causing slowdown, used python's p
ow method instead,
    Please understand my situation
    '''
    return pow(a,k,n)
```

In [4]:
```python
def Prime_Number_Generator():
    '''
    generate a big prime number n
    '''

    while(1): #keep running until we find what we want

        #generate a big random number n
        n = random.getrandbits(1000)

        #fermat prime test
        a = random.randint(2,n-2) #non inclusive n-1
        if Faster_Exponentiation(a, n-1, n) == 1:
            return n #for big numbers, passing the test once is enough, don't
 you think?
```

In [5]:
```python
a = Prime_Number_Generator()
print(a)
```

```
2449171020217963040628537465076145996263165790980931761853413230635853615472l
1681841005088408146139773991571957264441421232487638144003223160064844045l184
4725377628080455194717573334217709837691375091495061053950575783486839621248 7
91283982962934121697832074652138722307107543442254989255518505858859 27
```

Yea... that's probably a prime...

In [6]:
```python
def egcd(a, b):

    x = 0
    y = 1
    u = 1
    v = 0

    while a != 0:
        q = b//a
        r = b%a
        m = x-u*q
        n = y-v*q
        b = a
        a = r
        x = u
        y = v
        u = m
        v = n

    return b, x, y
```

In [7]:

```
#reminders on how to deal with strings, bit-blocks, and ints in python
rep_int = 2
rep_str = format(rep_int, "064b")
rep_bit = ord("a")
print(rep_int, rep_str, rep_bit)

#example convert message "hello" to bit blocks of size 16
def test_convert(message, block):
    bits = []
    for ch in message:
        bits.append(int(format(ord(ch), "0{0}b".format(block)),2))
    return bits

#example convert message from bits back to string
def test_revert(bits, block):
    message = ""
    for item in bits:
        message = message + chr(item)

    return message

a = test_convert("This is a test",32)
print(a)
print(test_revert(a, 32))
```

```
2 0000000000000000000000000000000000000000000000000000000000000010 97
[84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 116, 101, 115, 116]
This is a test
```

In [8]:
```
from collections import deque

def rotate(seq, n):

    #rotate involves replacement of the value, so you join it to the very end
    seq.rotate(n)

    return "".join(seq)

def sum0(bits):

    #x2 according to assignment
    seq = deque(bits)
    x = int(rotate(seq.copy(), 2), 2) ^ int(rotate(seq.copy(), 13), 2) ^ int(r
otate(seq.copy(), 22), 2)
    return x

def sum1(bits):

    #x1 according to assignmnet
    seq = deque(bits)
    x = int(rotate(seq.copy(), 6), 2) ^ int(rotate(seq.copy(), 11), 2)^int(rot
ate(seq.copy(),25),2)
    return x

def choose(x, y, z):

    #choosing x y or z
    return z^(x & (y^z))

def majority(x, y, z):

    #whichever one is the majority after x&z or x&y or y&z
    return ((x|y)&z)|(x&y)

def padding(string):

    #padding by adding 1, and the requisite number of 0s to the string
    return string + "1" + "0"*(512-len(string)+1-len(format(len(string), '032
b'))) + "".join(format(len(string),'032b'))

def sig0(bits):

    #s0 from the assignment
    seq = deque(bits)

    return int(rotate(seq.copy(), 7), 2)^int(rotate(seq.copy(),18),2)^int(bits
,2) >> 3 #shift 3

def sig1(bits):


    #s1 from the assignment
    seq = deque(bits)

    return int(rotate(seq.copy(), 17), 2)^int(rotate(seq.copy(),19),2)^int(bit
```

```
s,2) >> 10 #shift 10

def SHA_256(string):

    #SHA256 proper?

    #first pad
    binary_string = padding("".join(format(ord(x), '08b') for x in string))
    split = []

    #break into many bits and pieces of 32 from 512
    for i in range(32, 513, 32):
        split.append(binary_string[i-32:i]) #block from beginning, and every b
lock of size 32

    #IV
    h0 = 0x6a09e667 #a
    h1 = 0xbb67ae85 #b
    h2 = 0x3c6ef372 #c
    h3 = 0xa54ff53a #d
    h4 = 0x510e527f #e
    h5 = 0x9b05688c #f
    h6 = 0x1f83d9ab #g
    h7 = 0x5be0cd19 #h
    a,b,c,d,e,f,g,h = h0, h1, h2, h3, h4, h5, h6, h7

    #K constants
    k = (0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f
1, 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 )

    #for 64 iterations as per the assignment
    for i in range(0, 64):

        if i > 15:
            split.append(format((sig1(split[i-2])+sig0(split[i-15])+int(split[
i-7],2)+int(split[i-16],2))%2**32,'032b'))

        #The compression proper, temp1 and temp2 given from assignment, follow
ing use of MAJ and CH
        t1 = (h + sum1(format(e, '032b')) + choose(e,f,g) + k[i] + int(split[i
],2))%2**32
        t2 = (sum0(format(a, '032b')) + majority(a,b,c))%2**32
```

```python
        #set each a->h values
        h = g
        g = f
        f = e
        e = (d + t1) % 2 ** 32
        d = c
        c = b
        b = a
        a = (t1 + t2) % 2 ** 32

        #update each value mod 32
        h0 = (h0 + a) % 2 ** 32
        h1 = (h1 + b) % 2 ** 32
        h2 = (h2 + c) % 2 ** 32
        h3 = (h3 + d) % 2 ** 32
        h4 = (h4 + e) % 2 ** 32
        h5 = (h5 + f) % 2 ** 32
        h6 = (h6 + g) % 2 ** 32
        h7 = (h7 + h) % 2 ** 32

        return (hex(h0), hex(h1), hex(h2), hex(h3), hex(h4), hex(h5), hex(h6),
hex(h7))
```

In [9]:

```python
message = "Is this real life?"
print(message)
print(SHA_256(message))
```

```
Is this real life?
('0xaf858f28', '0x257194ec', '0xf7d6a1f7', '0xe1bee8ac', '0x33495595', '0xec1
3bb0b', '0xba894237', '0x7b64a6c4')
```

In [10]:
```
def gcd(a,b):
    #calculate the gcd between inputs a,b

    while b > 0: #note that if a<b then a%b == 0, while ends
        a,b = b, a%b

    return a

#def EEA already done

def inverse_mod(m, a):
    #calculates the inverse of a%m

    #get values from euclid's extended alg
    g, x, y = egcd(a, m)
    if g!=1:
        return None #inverse does not exist, account for this!!
    else:
        return x%m

def div_mod(m, a,b):
    #computes a/b mod m

    inverse = inverse_mod(m,b) #get inverse
    try:
        return (a*inverse)%m #division is just multiplying the inverse
    except:
        print("inverse did not exist")

#def Faster_Exponentiation(a,k,n) already done

def factors(x):
    #calculate the factors of x

    factors = []
    count = 2

    while x > 1:

        if x%count == 0: #everytime it is divisible by 2
            factors.append(count) #it is a factor
            x = x/count
        else:
            count+=1 #keep check

    return factors

def choose_e(totient):
    #calculate the 1 < e < totient, see if gcd(e, totient) == 1

    while(1):
        e = random.randint(2, totient)

        if gcd(e, totient) == 1:
            return e
```

```
#def Prime_Number_Generator() already done

def rsa_key():
    #p and q are BIG primes
    p = Prime_Number_Generator()
    q = Prime_Number_Generator()

    #n is the multiplication of these two BIG primes
    n = p*q

    #define phi,e, and d for decrypt
    phi=(p-1)*(q-1)
    e = choose_e(phi)
    d = inverse_mod(phi,e)

    return n,e,d

def rsa_enc(n,e,m):

    return Faster_Exponentiation(m, e, n)

def rsa_dec(n,d,c):

    return Faster_Exponentiation(c, d, n)

def RSA(message):
    #goes through RSA encrypt decrypt to test implementation

    #print original message value to test
    print("Original message: ",message)

    #convert message to numerical values
    BLOCK_SIZE = 32
    message_enc = test_convert(message, BLOCK_SIZE)
    print("Message encoded: ",message_enc)

    #gen keys
    n,e,d = rsa_key()

    #encryption for each chunk
    cipher_chunks=[]
    for block in message_enc:
        cipher = rsa_enc(n,e,block)
        cipher_chunks.append(cipher)

    print("Encrypted Text: ")
    print(cipher_chunks)

    #decryption
    plain_chunks = []
    for chunk in cipher_chunks:

        try:
            plain = rsa_dec(n,d,chunk) #for each chunk of the cipher, decrypt
        except:
            plain = -1
```

```
        plain_chunks.append(plain)

    #convert back to text
    plain_text = test_revert(plain_chunks, BLOCK_SIZE)
    print("Decrypted Text: ", plain_text)


    return plain_text == message #returns TRUE if encryption decryption worked
```

In [11]:
```
RSA("This is a test")
```

```
Original message:  This is a test
Message encoded:  [84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 116, 101, 11
5, 116]
Encrypted Text:
[23004365765317820541953179650809751236221220868561062345018825422873958628754
0764309691989854837371591419683949369106311111386999268692166016167150955897
82083109208044078807812978357378555701083914105766068697460015465259951991907
12252589369346257253011925384489602171013680991686321875756689362166083433913
44050805556393456078360847712529482690773670473528272692027162225120477650613
44607599077479608172924665787629095249448253367877058864721385313023220187949
03233798449103490707481694699035090099633371249131349820507529461725201516520
0136912301809794242959007525288056889581829402194724137529338504, 71933412471
82012041023571609476236639203607617789652170037501624427180124650719272237906
88321062689372461898659002229559891289030673254375713284167196232283901879002
45319659278193322630818265570628425477548909110592635085945497349576374668246
13558418538368304989043456440748448144636647534858385348588290319387444249058
65087066882117314587154276969197400431742896254023271010865072865332424808881
25417602097978064431174793136388459431101954186928156662807469847282089937219
19822570434375123710397133084606110219365145122013849000813485053150760711206
608789043832134937634946622072035121947970278217380, 296566829856946416398969
93770385359129840768621016752376357865570371801328558687680498665394830427523
94110869684835266878435983016418505814586285531111886826014289612702549355325
05805484520478555375579187488259941437543367633792120642411748786034773137202
18952482152419317642590560195401305896416349274519551380716770054250906405722
73087951633960883759618210110130932664480649749939362132872762935546304555702
03759895074107713105898038723353034520606892573948485514907606624177973493728
20336340686936084449824343935396712344521339734206698157592092309954148436966
525234731259312328409511123089060785098, 149972994390689730957675397701531809
41011210944377338100287906729275860288762965583571367244225977019255705381898
95501538276801301864493617192663343988004677357292617168472287623817194609114
94557926446459670162509699485818986874495087876796077978905770268176764923318
03467999630830677957053164720619035362205560888385227938637383961388649998266
03574830648454644423689998339573442359396694915063983689674825178190335607621
06719377108026023350841068566186018780884208377287670951472175865610453989605
25904402394827778425492676231497113123569947277338662917520874277301001472738
85306173709199991431465784, 297195067650198552796500951421947081885852359118
04814060800066967719229259033474170832066106461739999582798240364853140512876
51260453570479677375316352287101918563990818169533461658758677978418184096097
29546080929257998445463656879923715972006302831437548160040500134443098371890
83474308944528546062296312592063254914851716730293261518884686544052831283875
96972491756818665503509709496353907054425076148035375061175592856200479436548
15313343583342470672938811149736398355959906425663311597050699580658479781591
50079750783964736674284484494150646420300046208068108237648885324790159342560
220828895340439, 296566829856946416398969937703853591298407686210167523763578
65570371801328558687680498665394830427523941108696848352668784359830164185058
14586285531111886826014289612702549355325058054845204785553755791874882599414
37543367633792120642411748786034773137202189524821524193176425905601954013058
96416349274519551380716770054250906405722730879516339608837596182101101309326
64480649749939362132872762935546304555702037598950741077131058980387233530345
20606892573948485514907606624177973493728203363406869360844498243439353967123
44521339734206698157592092309954148436966525234731259312328409511123089060785
098, 149972994390689730957675397701531809410112109443773381002879067292758602
88762965583571367244225977019255705381898955015382768013018644936171926633439
88004677357292617168472287623817194609114945579264464596701625096994858189868
74495087876796077978905770268176764923318034679996308306779570531647206190353
62205560888385227938637383961388649998266035748306484546444236899983395734423
59396694915063983689674825178190335607621067193771080260233508410685661860187
```

8088420837728767095147217586561045398960525904402394827778425492676231497113 1
2356994727733866291752087427730100147273885306173709199999143146578 4, 2971950
6765019855279650095142194708188585235911804814060800066967719229259033474170 8
3206610646173999958279824036485314051287651260453570479677375316352287101918 5
6399081816953346165875867797841818409609729546080929257998445463656879923715 9
7200630283143754816004050013444309837189083474308944528546062296312592063254 9
1485171673029326151888468654405283128387596972491756818665503509709496353907 0
5442507614803537506117559285620047943654815313343583342470672938811149736398 3
5595990642566331159705069958065847978159150079750783964736674284484494150646 4
203000462080681082376488853247901593425602208288953404 39, 265550141699530224 1
3055619058593055644364586182800918706235399658224211060229477735976830875236 4
9562100368359390693300996643756081783283677033269844462279787638432203198244 8
8832634431532210528994320589572963538491512044898083895232802365653302436366 8
4752370451274107019509506467253399583371713327631036217345817750158145790227 0
7570873081612591093356962223050194555719713217670467734378661460746077545083 7
6885829871260730755597161558184827127205415317993525817692520460283201890904 0
4999960725375630506372866223049299247223450598348300080466386607162414171567 3
877838916156869320979423857282015337421758 28, 297195067650198552796500951421 9
4708188585235911804814060800066967719229259033474170832066106461739999582798 2
4036485314051287651260453570479677375316352287101918563990818169533461658758 6
7797841818409609729546080929257998445463656879923715972006302831437548160040 5
0013444309837189083474308944528546062296312592063254914851716730293261518884 6
8654405283128387596972491756818665503509709496353907054425076148035375061175 5
9285620047943654815313343583342470672938811149736398355959906425663311597050 6
9958065847978159150079750783964736674284484494150646420300046208068108237648 8
85324790159342560220828895340439, 654895309892791017261455406606199194443037 9
6946856683848563671511482624063838703306795201552127039586166781836751109617 8
8795424430022192271097274757184947542901952374551315813273199149727470832220 7
6297131688631218920897566248034783981429087175598613508904881184037981804151 7
5085083134471010669709159836462357515389558393716244150242991068199052228727 2
6083022536350061378618858140431096048184617609335818488244788106530497296735 3
5465563691616459802536590308989395724332027153331544936233021631781257805331 9
2720863280906042284873710428535394119987327024089687213342325555797082972498 1
02921346292781012 8, 350538472140489439998239729195280851497012153530550461532
5615211610420168600938016652708118886509009273967063967334579179280850522370 9
6971890410836314809121910345953829505725417138159629353976012800481096809318 1
3258056566186949371435043850758324841469867544845920866771418032827261589982 5
0566400631879114721568461346710472660286227907706295354044826948839330505899 7
5114121380419329366731728850015836585670696555785490852319187064517470359434 1
2813768289103790617553207371623939336884538358663946131078898896886380403844 5
3084043258657014857771099796921359143901123129857053828066116598097145784435 5
013498, 149972994390689730957675397701531809410112109443773381002879067292758
6028876296558357136724422597701925570538189895501538276801301864493617192663 3
4398800467735729261716847228762381719460911494557926446459670162509699485818 9
8687449508787679607797890577026817676492331803467999630830677957053164720619 0
3536220556088838522793863738396138864999826603574830648454644423689998339573 4
4235939669491506398368967482517819033560762106719377108026023350841068566186 0
1878088420837728767095147217586561045398960525904402394827778425492676231497 1
131235699472773386629175208742773010014727388530617370919999143146578 4, 6548
9530989279101726145540660619919444303796946856683848563671511482624063838703 3
0679520155212703958616678183675110961788795424430022192271097274757184947542 9
0195237455131581327319914972747083222076297131688631218920897566248034783981 4
2908717559861350890488118403798180415175085083134471010669709159836462357515 3
8955839371624415024299106819905222872726083022536350061378618858140431096048 1
8461760933581848824478810653049729673535465563691616459802536590308989395724 3
3202715333154493623302163178125780533192720863280906042284873710428535394119 9

```
873270240896872133423255557970829724981029213462927810128]
Decrypted Text:  This is a test
```

Out[11]:

```
True
```

In [ ]:

```

```