AN TOÀN BẢO MẬT THÔNG TIN

[Buoi01.py 2](#_Toc43229243)

[Buoi02\_03.py 4](#_Toc43229244)

[pydes.py 7](#_Toc43229245)

[Buoi04.py 14](#_Toc43229246)

[Create\_Certificate\_OpenSSL 19](#_Toc43229247)

# Buoi01.py

# Mat ma Affine

# -\*- coding: utf-8 -\*-

from tkinter import \*

window = Tk()

window.title("Welcome to Demo An Toan Bao Mat Thong Tin")

lb0 = Label(window, text=" ", font=("Arial Bold", 10))

lb0.grid(column=0, row=0)

lbl = Label(window, text="CHƯƠNG TRÌNH DEMO", font=("Arial Bold", 20))

lbl.grid(column=1, row=1, columnspan=4)

lb2 = Label(window, text="MẬT MÃ AFFINE", font=("Arial Bold", 15))

lb2.grid(column=0, row=2)

plainlb3 = Label(window, text="PLANT TEXT", font=("Arial", 14))

plainlb3.grid(column=0, row=3)

plaintxt = Entry(window, width=20)

plaintxt.grid(column=1, row=3)

KEYlb4 = Label(window, text="KEY PAIR", font=("Arial", 14))

KEYlb4.grid(column=2, row=3)

KEYA = Entry(window, width=4)

KEYA.grid(column=3, row=3)

KEYB = Entry(window, width=4)

KEYB.grid(column=4, row=3)

lb5 = Label(window, text="CIPHER TEXT", font=("Arial", 14))

lb5.grid(column=0, row=4)

ciphertxt3 = Entry(window, width=20)

ciphertxt3.grid(column=1, row=4)

denctxt3 = Entry(window, width=20)

denctxt3.grid(column=3, row=4)

def Char2Num(c): return ord(c) - 65

def Num2Char(n): return chr(n + 65)

def xgcd(a, b): # Extended Euclidean Algorithm

tmp = b

x0, x1 = 1, 0

while b > 0:

q, a, b = a // b, b, a % b

x0, x1 = x1, x0 - q \* x1

if x0 < 0:

x0 += tmp

return x0

# def xgcd(a, m): # a^-1 % m

# if isPrime(m):

# return a\*\*(m-2) % m

# else: # Any value of m

# return a\*\*(phi(n)-1) % m

def encryptAF(txt, a, b, m):

r = ""

for c in txt:

e = (a \* Char2Num(c) + b) % m

r += Num2Char(e)

return r

def decryptAF(txt, a, b, m):

r = ""

a1 = xgcd(a, m)

for c in txt:

e = (a1 \* (Char2Num(c) - b)) % m

r += Num2Char(e)

return r

def matma():

a, b, m = int(KEYA.get()), int(KEYB.get()), 26

cirphertext = encryptAF(plaintxt.get(), a, b, m)

ciphertxt3.delete(0, END)

ciphertxt3.insert(INSERT, cirphertext)

def giaimat():

a, b, m = int(KEYA.get()), int(KEYB.get()), 26

decryptext = decryptAF(ciphertxt3.get(), a, b, m)

denctxt3.delete(0, END)

denctxt3.insert(INSERT, decryptext)

AFbtn = Button(window, text="Mã Hóa", command=matma)

AFbtn.grid(column=5, row=3)

DEAFbtn = Button(window, text="Giải Mã ", command=giaimat)

DEAFbtn.grid(column=2, row=4)

window.geometry('700x200')

window.mainloop()

# Buoi02\_03.py

# Mat ma Khoa doi xung (SKC)

# -\*- coding: utf-8 -\*-

import sys

from tkinter import \*

from Crypto.Cipher import AES, DES3

from Crypto import Random

from pydes import des

window = Tk()

window.title("CHƯƠNG TRÌNH DEMO")

lb0 = Label(window, text=" ", font=("Arial Bold", 10))

lb0.grid(column=0, row=0)

lbl = Label(window, text="MẬT MÃ KHOÁ ĐỐI XỨNG (SKC)", font=("Arial Bold", 20))

lbl.grid(column=0, row=1, columnspan=6, pady=16)

lbAlgo = Label(window, text="ALGO", font=("Arial", 14),)

lbAlgo.grid(column=0, row=2, sticky="E")

lbMode = Label(window, text="MODE", font=("Arial", 14),)

lbMode.grid(column=2, row=2, sticky="E")

plainlb3 = Label(window, text="PLAIN TEXT", font=("Arial", 14))

plainlb3.grid(column=0, row=3, sticky="E")

plaintxt = Entry(window, width=20)

plaintxt.grid(column=1, row=3)

KEYlb4 = Label(window, text="KEY", font=("Arial", 14))

KEYlb4.grid(column=2, row=3, sticky="E")

SecretKey = Entry(window, width=20)

SecretKey.grid(column=3, row=3)

lb5 = Label(window, text="CIPHER TEXT", font=("Arial", 14))

lb5.grid(column=0, row=4, sticky="E")

ciphertxt3 = Entry(window, width=20)

ciphertxt3.grid(column=1, row=4)

denctxt3 = Entry(window, width=20)

denctxt3.grid(column=3, row=4)

d = des()

algo = AES

mode = algo.MODE\_ECB

ciphered = ""

iv = Random.new().read(algo.block\_size)

description = {

"AES (pycrypto)": '\* Độ dài KEY 16 (AES-128), 24 (AES-192), 32 (AES-256)\n\* Độ dài PLAINTEXT là bội của 16',

'DES (pycrypto)': '\* Độ dài KEY 16 hoặc 24\n\* Độ dài PLAINTEXT là bội của 8',

"DES (pydes)": '\* Độ dài KEY >= 8\n\* Độ dài PLAINTEXT là bội của 8'

}

def bin2txt(bin\_val):

return bin\_val.decode('utf-8')

def change\_algo\_option(option):

global algo, iv

algo = AES if option == 'AES (pycrypto)' else DES3

iv = Random.new().read(algo.block\_size)

lbDes['text'] = description[option]

mode\_options['state'] = 'disabled' if option == 'DES (pydes)' else 'normal'

ciphertxt3.delete(0, END)

denctxt3.delete(0, END)

def change\_mode\_option(option):

global mode

mode = algo.MODE\_CBC if option == 'CBC' else algo.MODE\_ECB

ciphertxt3.delete(0, END)

denctxt3.delete(0, END)

def matma():

print('Mật Mã clicked !!!')

msg = plaintxt.get()

key = SecretKey.get()

ok = True

global ciphered

if "pycrypto" in var\_algo.get():

obj = algo.new(key, mode, iv) # Cipher-Block Chaining

ciphered = obj.encrypt(msg)

else:

ciphered = d.encrypt(key, msg)

print('KEY =', key, 'PLAINTEXT =', msg)

print("Ciphered: %r" % ciphered)

ciphertxt3.delete(0, END)

ciphertxt3.insert(INSERT, ciphered)

def giaimat():

print('Giải Mật clicked !!!')

key = SecretKey.get()

if "pycrypto" in var\_algo.get():

obj = algo.new(key, mode, iv)

deciphered = obj.decrypt(ciphered)

deciphered = bin2txt(deciphered)

else:

deciphered = d.decrypt(SecretKey.get(), ciphered)

print('KEY =', key, 'DECRYPTEXT =', deciphered)

denctxt3.delete(0, END)

denctxt3.insert(INSERT, deciphered)

lbDes = Label(window, text=description["AES (pycrypto)"], font=(

"Arial", 14), fg="red")

lbDes.grid(column=1, row=5, columnspan=4)

EncrypBtn = Button(window, text="Mật Mã", command=matma)

EncrypBtn.grid(column=5, row=3)

DecryptBtn = Button(window, text="Giải Mật", command=giaimat)

DecryptBtn.grid(column=2, row=4)

var\_algo = StringVar(window)

var\_algo.set("AES (pycrypto)") # default value

algo\_options = ["AES (pycrypto)", "DES (pycrypto)", "DES (pydes)"]

algo\_options = OptionMenu(

window, var\_algo, \*(algo\_options), command=change\_algo\_option)

algo\_options.grid(column=1, row=2)

var\_mode = StringVar(window)

var\_mode.set("ECB") # default value

mode\_options = ["ECB", "CBC"]

mode\_options = OptionMenu(window, var\_mode, \*(mode\_options),

command=change\_mode\_option)

mode\_options.grid(column=3, row=2)

dim = '600x220' if sys.platform == 'darwin' else '700x240'

window.geometry(dim)

window.mainloop()

# pydes.py

# -\*- coding: utf8 -\*-

# Initial permut matrix for the datas

PI = [58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7]

# Initial permut made on the key

CP\_1 = [57, 49, 41, 33, 25, 17, 9,

1, 58, 50, 42, 34, 26, 18,

10, 2, 59, 51, 43, 35, 27,

19, 11, 3, 60, 52, 44, 36,

63, 55, 47, 39, 31, 23, 15,

7, 62, 54, 46, 38, 30, 22,

14, 6, 61, 53, 45, 37, 29,

21, 13, 5, 28, 20, 12, 4]

# Permut applied on shifted key to get Ki+1

CP\_2 = [14, 17, 11, 24, 1, 5, 3, 28,

15, 6, 21, 10, 23, 19, 12, 4,

26, 8, 16, 7, 27, 20, 13, 2,

41, 52, 31, 37, 47, 55, 30, 40,

51, 45, 33, 48, 44, 49, 39, 56,

34, 53, 46, 42, 50, 36, 29, 32]

# Expand matrix to get a 48bits matrix of datas to apply the xor with Ki

E = [32, 1, 2, 3, 4, 5,

4, 5, 6, 7, 8, 9,

8, 9, 10, 11, 12, 13,

12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21,

20, 21, 22, 23, 24, 25,

24, 25, 26, 27, 28, 29,

28, 29, 30, 31, 32, 1]

S\_BOX = [

[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

[0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

[4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],

],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],

],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],

],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],

],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],

],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],

],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],

],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],

]

]

# Permut made after each SBox substitution for each round

P = [16, 7, 20, 21, 29, 12, 28, 17,

1, 15, 23, 26, 5, 18, 31, 10,

2, 8, 24, 14, 32, 27, 3, 9,

19, 13, 30, 6, 22, 11, 4, 25]

# Final permut for datas after the 16 rounds

PI\_1 = [40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25]

# Matrix that determine the shift for each round of keys

SHIFT = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

def string\_to\_bit\_array(text): # Convert a string into a list of bits

array = list()

for char in text:

binval = binvalue(char, 8) # Get the char value on one byte

# Add the bits to the final list

array.extend([int(x) for x in list(binval)])

return array

def bit\_array\_to\_string(array): # Recreate the string from the bit array

res = ''.join([chr(int(y, 2)) for y in [''.join([str(x)

for x in \_bytes])

for \_bytes in nsplit(array, 8)]])

return res

def binvalue(val, bitsize): # Return the binary value as a string of the given size

binval = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]

if len(binval) > bitsize:

raise "binary value larger than the expected size"

while len(binval) < bitsize:

binval = "0"+binval # Add as many 0 as needed to get the wanted size

return binval

def nsplit(s, n): # Split a list into sublists of size "n"

return [s[k:k+n] for k in range(0, len(s), n)]

ENCRYPT = 1

DECRYPT = 0

class des():

def \_\_init\_\_(self):

self.password = None

self.text = None

self.keys = list()

def run(self, key, text, action=ENCRYPT, padding=False):

if len(key) < 8:

raise "Key Should be 8 bytes long"

elif len(key) > 8:

key = key[:8] # If key size is above 8bytes, cut to be 8bytes long

self.password = key

self.text = text

if padding and action == ENCRYPT:

self.addPadding()

elif len(self.text) % 8 != 0: # If not padding specified data size must be multiple of 8 bytes

raise "Data size should be multiple of 8"

self.generatekeys() # Generate all the keys

# Split the text in blocks of 8 bytes so 64 bits

text\_blocks = nsplit(self.text, 8)

result = list()

for block in text\_blocks: # Loop over all the blocks of data

# Convert the block in bit array

block = string\_to\_bit\_array(block)

block = self.permut(block, PI) # Apply the initial permutation

g, d = nsplit(block, 32) # g(LEFT), d(RIGHT)

tmp = None

for i in range(16): # Do the 16 rounds

d\_e = self.expand(d, E) # Expand d to match Ki size (48bits)

if action == ENCRYPT:

tmp = self.xor(self.keys[i], d\_e) # If encrypt use Ki

else:

# If decrypt start by the last key

tmp = self.xor(self.keys[15-i], d\_e)

tmp = self.substitute(tmp) # Method that will apply the SBOXes

tmp = self.permut(tmp, P)

tmp = self.xor(g, tmp)

g = d

d = tmp

# Do the last permut and append the result to result

result += self.permut(d+g, PI\_1)

final\_res = bit\_array\_to\_string(result)

if padding and action == DECRYPT:

# Remove the padding if decrypt and padding is true

return self.removePadding(final\_res)

else:

return final\_res # Return the final string of data ciphered/deciphered

def substitute(self, d\_e): # Substitute bytes using SBOX

subblocks = nsplit(d\_e, 6) # Split bit array into sublist of 6 bits

result = list()

for i in range(len(subblocks)): # For all the sublists

block = subblocks[i]

# Get the row with the first and last bit

row = int(str(block[0])+str(block[5]), 2)

# Column is the 2,3,4,5th bits

column = int(''.join([str(x) for x in block[1:][:-1]]), 2)

# Take the value in the SBOX appropriated for the round (i)

val = S\_BOX[i][row][column]

bin = binvalue(val, 4) # Convert the value to binary

# And append it to the resulting list

result += [int(x) for x in bin]

return result

def permut(self, block, table): # Permut the given block using the given table (so generic method)

return [block[x-1] for x in table]

# Do the exact same thing than permut but for more clarity has been renamed

def expand(self, block, table):

return [block[x-1] for x in table]

def xor(self, t1, t2): # Apply a xor and return the resulting list

return [x ^ y for x, y in zip(t1, t2)]

def generatekeys(self): # Algorithm that generates all the keys

self.keys = []

key = string\_to\_bit\_array(self.password)

key = self.permut(key, CP\_1) # Apply the initial permut on the key

g, d = nsplit(key, 28) # Split it in to (g->LEFT),(d->RIGHT)

for i in range(16): # Apply the 16 rounds

# Apply the shift associated with the round (not always 1)

g, d = self.shift(g, d, SHIFT[i])

tmp = g + d # Merge them

# Apply the permut to get the Ki

self.keys.append(self.permut(tmp, CP\_2))

def shift(self, g, d, n): # Shift a list of the given value

return g[n:] + g[:n], d[n:] + d[:n]

def addPadding(self): # Add padding to the datas using PKCS5 spec.

pad\_len = 8 - (len(self.text) % 8)

self.text += pad\_len \* chr(pad\_len)

# Remove the padding of the plain text (it assume there is padding)

def removePadding(self, data):

pad\_len = ord(data[-1])

return data[:-pad\_len]

def encrypt(self, key, text, padding=False):

return self.run(key, text, ENCRYPT, padding)

def decrypt(self, key, text, padding=False):

return self.run(key, text, DECRYPT, padding)

if \_\_name\_\_ == '\_\_main\_\_':

key = "secret\_k"

text = "Hello wo"

d = des()

r = d.encrypt(key, text)

r2 = d.decrypt(key, r)

print("Ciphered: %r" % r)

print("Deciphered: ", r2)

# Buoi04.py

# RSA

# -\*- coding: utf-8 -\*-

from tkinter import \*

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

window = Tk()

window.title("CHƯƠNG TRÌNH DEMO")

lb0 = Label(window, text=" ", font=("Arial Bold", 10))

lb0.grid(column=0, row=0)

lbl = Label(window, text="GIẢI THUẬT RSA", font=("Arial Bold", 20))

lbl.grid(column=0, row=1, columnspan=6, pady=16)

plainlb3 = Label(window, text="PLAIN TEXT", font=("Arial", 14))

plainlb3.grid(column=0, row=3, sticky="E")

lb5 = Label(window, text="CIPHER TEXT", font=("Arial", 14))

lb5.grid(column=0, row=4, sticky="E")

ciphertxt3 = Entry(window, width=20)

ciphertxt3.grid(column=1, row=4)

denctxt3 = Entry(window, width=20)

denctxt3.grid(column=3, row=4)

lbSTT = Label(window, text="", font=("Arial", 14), fg="red")

lbSTT.grid(column=1, row=5, columnspan=4, sticky="W")

key = RSA.generate(2048) # Sinh khoá

ciphered = ''

# Lưu khoá cá nhân (để mật mã)

f = open('private.pem', 'wb')

f.write(key.exportKey('PEM'))

f.close()

# Lưu khoá công khai (để mật mã)

f = open('public.pem', 'wb')

f.write(key.publickey().exportKey('PEM'))

f.close()

def onkeypress(event):

ciphertxt3.delete(0, END)

def genkeypair():

key = RSA.generate(2048)

pubkey = key.publickey().exportKey('PEM')

privkey = key.exportKey('PEM')

return (pubkey, privkey)

def matma():

print('Mật Mã clicked !!!')

global ciphered

pubkey = RSA.importKey(open('public.pem').read())

cipher = PKCS1\_OAEP.new(pubkey)

ciphered = cipher.encrypt(plaintxt.get().encode('utf-8'))

print('PLAINTEXT =', plaintxt.get())

print('ciphered =', ciphered)

ciphertxt3.delete(0, END)

ciphertxt3.insert(INSERT, 'Plaintext được mật mã!')

# ciphertxt3.insert(INSERT, ciphered.decode('utf-8', errors="ignore"))

def giaimat():

print('Giải Mật clicked !!!')

privkey = RSA.importKey(open('private.pem').read())

decipher = PKCS1\_OAEP.new(privkey)

deciphered = decipher.decrypt(ciphered)

print("Deciphered: %s" % deciphered)

denctxt3.delete(0, END)

denctxt3.insert(INSERT, deciphered.decode('utf-8'))

plaintxt = Entry(window, width=20)

plaintxt.bind('<Key>', onkeypress)

plaintxt.grid(column=1, row=3)

EncrypBtn = Button(window, text="Mật Mã", command=matma)

EncrypBtn.grid(column=2, row=3)

DecryptBtn = Button(window, text="Giải Mật", command=giaimat)

DecryptBtn.grid(column=2, row=4)

window.geometry('600x200')

window.mainloop()

# RSA

# -\*- coding: utf-8 -\*-

from tkinter import \*

window = Tk()

window.title("CHƯƠNG TRÌNH DEMO")

lb0 = Label(window, text=" ", font=("Arial Bold", 10))

lb0.grid(column=0, row=0)

lbl = Label(window, text="GIẢI THUẬT RSA", font=("Arial Bold", 20))

lbl.grid(column=0, row=1, columnspan=6, pady=16)

plainlb3 = Label(window, text="PLAIN TEXT", font=("Arial", 14))

plainlb3.grid(column=0, row=3, sticky="E")

plaintxt = Entry(window, width=20)

plaintxt.grid(column=1, row=3)

lb5 = Label(window, text="CIPHER TEXT", font=("Arial", 14))

lb5.grid(column=0, row=4, sticky="E")

ciphertxt3 = Entry(window, width=20)

ciphertxt3.grid(column=1, row=4)

denctxt3 = Entry(window, width=20)

denctxt3.grid(column=3, row=4)

lbSTT = Label(window, text="", font=("Arial", 14), fg="red")

lbSTT.grid(column=1, row=5, columnspan=4, sticky="W")

def xgcd(a, b): # Extended Euclidean Algorithm ~ a^-1 % b

tmp = b

x0, x1 = 1, 0

while b > 0:

q, a, b = a // b, b, a % b

x0, x1 = x1, x0 - q \* x1

if x0 < 0:

x0 += tmp

return x0

p, q = 3, 5

n = p\*q

phin = (p-1)\*(q-1)

e = 11

d = xgcd(e, phin)

ciphered = ""

def str2char(s):

char\_arr = ''

for c in s:

char\_arr += str(ord(c)) + ' '

return char\_arr.strip()

def char2str(char\_arr, d=10):

char\_arr = char\_arr.split()

s = ''

for hex\_el in char\_arr:

s += chr(int(hex\_el, d))

return s

def encrypt(plaintext):

s = str2char(plaintext)

ciphertext = ''

print("\nplaintext = %s" % s)

for c in s:

if c != ' ':

ciphertext += hex(int(c)\*\*e % n)[2:]

else:

ciphertext += ' '

print("\nciphertext = %s" % ciphertext)

print(char2str(ciphertext, n+1))

return ciphertext

def decrypt(ciphertext):

plaintext = ''

for c in ciphertext:

if c == ' ':

plaintext += ' '

else:

plaintext += hex(int(c, n+1)\*\*d % n)[2:]

print("\nplaintext = %s" % plaintext)

print(char2str(plaintext))

return plaintext

def matma():

print('Mật Mã clicked !!!')

global ciphered

ciphered = encrypt(plaintxt.get())

print('PLAINTEXT =', plaintxt.get())

print("Ciphered: %r" % ciphered)

ciphertxt3.delete(0, END)

ciphertxt3.insert(INSERT, char2str(ciphered, n+1))

def giaimat():

print('Giải Mật clicked !!!')

deciphered = decrypt(ciphered)

print('DECRYPTEXT =', deciphered)

print("Deciphered: %s" % deciphered)

denctxt3.delete(0, END)

denctxt3.insert(INSERT, char2str(deciphered))

EncrypBtn = Button(window, text="Mật Mã", command=matma)

EncrypBtn.grid(column=2, row=3)

DecryptBtn = Button(window, text="Giải Mật", command=giaimat)

DecryptBtn.grid(column=2, row=4)

window.geometry('600x200')

window.mainloop()

# Create\_Certificate\_OpenSSL

openssl req -x509 -newkey rsa:4096 -sha256 -keyout dcongtinh.com.key -out dcongtinh.com.cer -subj "/C=VN/ST=Can Tho/L=Can Tho/O=Dao Cong Tinh, Inc./emailAddress=dcongtinh@gmail.com/OU=IT/CN=dcongtinh.com" -days 600