Infonet Security HW3

Harris Ransom

10/30/2024

In [110…

```
import numpy as np
import random
import sympy
import math
```

Problem 1

In [111…

```
def exponentiation(m,e,n):
    return((m**e)%n)

print(exponentiation(2000,2020,2023))
```

```
72
```

In [112…

```
def extendedGCD(a, b):
    x0 = 1
    x1 = 0
    y0 = 0
    y1 = 1

    while b != 0:
        q = a // b
        a, b = b, a % b
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1

    return a, x0

def inverseFinder(a,n):
    gcd, x = extendedGCD(a, n)
    if gcd == 1:
        return x % n

print(inverseFinder(2000,2023))
```

```
1935
```

Problem 2

In [113…

```python
def rsaKey(e):
    while True:
        while True:
            p = random.getrandbits(512)
            if sympy.isprime(p):
                break


        while True:
            q = random.getrandbits(512)
            if sympy.isprime(q) is True:
                if q != p:
                    break
        n = p*q
        phi = (p-1) * (q-1)

        if math.gcd(e, phi) == 1 and e<phi:
            publicKey = [e,n]
            privateKey = [inverseFinder(e,phi),n]
            return(publicKey,privateKey)

publicKeys = []
privateKeys = []
for i in range(10):
    publicKey, privateKey = rsaKey(3)
    publicKeys.append(publicKey)
    privateKeys.append(privateKey)

for i in range(10):
    print(f"Public Key {i}: {publicKeys[i]}")
    print(f"Private Key {i}: {privateKeys[i]}")
    print(f"\n")
```

```
Public Key 0: [3, 10544538975581158699755595701918428380780111749278593142720021899 2
6180316874638505086656629074279975686974815654424884677433157093206639639109703518 51
336415660009218786883882381343730170655996969777624039492452846729130555804382806979
3953836573281750332712916837414724516793021437105849697068252539206297 9541]
```

```
Public Key 4: [3, 682601086045711564698713998099197848498145357160126639778027891982
```

```
Public Key 4: [3, 682601086045711564698713998099197848498145357160126639778027891982
706249046294732753445903282840212653673311483536864732417875521846687892297097018998
903023374616211755672008032679049580860493494709066424758155507373962636904247132470
1331662086719562143776210421784178355316649419439386110547791571794725541]
Private Key 4: [45506739069714104313247599873279856566543023810675109318535192798847
```

```
Public Key 8: [3, 3205303822028620082598992149370634550572065675757332400927063834667785414135505204059733441934096158194662822481958572926106766516555397434763723952072430893268665141981599723773934044832988111116429946702756832261308715678917554964562015417907706187143563972674976545315136062991574478551265595811768881 6497]
```

```
Private Key 8: [2136869214685746721732661432913756367048043783838221600618042556445190276090336802706488961289397438796441881654639048617404511011036931623175815968048286321358483001668620870134482584399558898623539700421663592048571589105896717259291344381949755231302015921620064283108665494564431890801939042723420570 6827, 3205303822028620082598992149370634550572065675757332400927063834667785414135505204059733441934096158194662822481958572926106766516555397434763723952072430893268665141981599723773934044832988111116429946702756832261308715678917554964562015417907706187143563972674976545315136062991574478551265595811768881 6497]
```

```
Public Key 9: [3, 1294976201342267241667182141743351481601730633245664541216148553608030559903651537093686466346992296448750155853158885936995601468157492475340961656481239682163126008714069547043322339295529423812271562041852589633488977281770718493574325916710715257847083793502888314195749021749889211230956589184778343 69587]
```

```
Private Key 9: [8633174675615114944447880944955676544011537554971096941440990357386870399357676913957909775646615309658334372354392572913304009787716616502273077709874929694074668131328041457135417564806104144068622336140999076262957801761704204067732046184754808937629237507748915491504315954242627192354060854644620803 8891, 1294976201342267241667182141743351481601730633245664541216148553608030559903651537093686466346992296448750155853158885936995601468157492475340961656481239682163126008714069547043322339295529423812271562041852589633488977281770718493574325916710715257847083793502888314195749021749889211230956589184778343 69587]
```

In [114…

```python
def rsaEncrypt(msg, pubKey):
    e = int(pubKey[0])
    n = int(pubKey[1])
    msg = int.from_bytes(msg.encode('utf-8'),'big')

    ciypherText = pow(msg,e,n)
    return(ciypherText)

def rsaDecrypt(msg, privateKey):
    d = int(privateKey[0])
    n = int(privateKey[1])

    cipherText = pow(msg,d,n)

    cipherText = cipherText.to_bytes((cipherText.bit_length() + 7) // 8, 'big')
    recoveredPlaintext = cipherText.decode('utf-8')

    return(recoveredPlaintext)

pubKey,privateKey = rsaKey(65537)
plainText = "123456789"
print(f"Plain Text Message: {plainText}")
cipher = rsaEncrypt(plainText,pubKey)
recoveredText = rsaDecrypt(cipher,privateKey)
print(f"Recovered Message: {recoveredText}")
```

```
Plain Text Message: 123456789
Recovered Message: 123456789
```

In [115…

```
plainText = "hello world!"
print(f"Plain Text Message: {plainText}")
cipher = rsaEncrypt(plainText,pubKey)
recoveredText = rsaDecrypt(cipher,privateKey)
print(f"Recovered Message: {recoveredText}")
```

```
Plain Text Message: hello world!
Recovered Message: hello world!
```

Problem 3

Problem 3a

In [116…

```
def primitiveRoot(a,n):
    result = []
    for i in range(n-1):
        result.append((a**(i+1))%n)
    if len(result) != len(set(result)):
        print(f"{a} is not a primitive root mod {n}")
        print(result)
    if len(result) == len(set(result)):
        print(f"{a} is a primitive root mod {n}")
        print(result)
    return(result)

_ = primitiveRoot(2,13)
_ = primitiveRoot(3,13)
```

```
2 is a primitive root mod 13
[2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1]
3 is not a primitive root mod 13
[3, 9, 1, 3, 9, 1, 3, 9, 1, 3, 9, 1]
```

Problem 3b

$$2^x \equiv 12(mod 13)$$

$g = 2, h = 12, p = 13, N = 13 - 1 = 12 = 2^2 - 1$

q	e	$g^{\frac{p-1}{q^e}}$	$h^{\frac{p-1}{q^e}}$
2	2	8	1728
3	1	16	20736

$x \equiv (x_0 + 2x_1)(mod 2^2)$

$(8^{2^1})^{x_0}(mod13) = (1728^{2^1})(mod13) \Rightarrow 12^{x_0} = 1 \Rightarrow x_0 = 0$
$(8^{2^1})^{x_1}(mod13) = (1728 * 8^{-x_0})^{2^0}(mod13) \Rightarrow 12^{x_1} = 12 \Rightarrow x_1 = 1$
$x \equiv 0 + 2(1)(mod2^2) \Rightarrow x \equiv 2(mod2^2)$

$x \equiv x_0(mod3^1)$
$16^{x_0}(mod13) = 20736(mod13) \Rightarrow 16^{x_0} = 1 \Rightarrow x_0 = 0$
$x \equiv 0(mod3)$

We can use Chinese Remainder Theorem to solve for x $3 * 2(mod4) \Rightarrow 2(mod4)$
$4 * 0(mod3) \Rightarrow 0(mod3)$
$x = 3 * 2 + 0 = 6$

$$x = 6$$

Problem 3c

We can find the following using the result from 3b

$$2^x \equiv 3(mod13)$$

$$2^6 \equiv 12(mod13)$$

$$\frac{2^6}{4} \equiv \frac{12}{4}(mod13)$$

$$2^4 \equiv 3(mod13)$$

$$x = 4$$

In [117…

```
discreteLogs = primitiveRoot(2,13)
solutions = []
for i in range(len(discreteLogs)):
    if discreteLogs[i] == 12:
        solutions.append(i+1)
print(f"Solutions to 2B:")
print(solutions)

solutions = []
for i in range(len(discreteLogs)):
    if discreteLogs[i] == 3:
        solutions.append(i+1)
print(f"Solutions to 2C:")
print(solutions)
```

```
2 is a primitive root mod 13
[2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1]
Solutions to 2B:
[6]
Solutions to 2C:
[4]
```