Reid Nagurka

HW 5

Problem 1

```
#main
l = lfsr([2,4,5,7,11,14])
l.seed(int_to_bin(1234))
print (l.gen(10,2000))
```

Search | Stack Data | Python Shell

Commands execute without debug. Use arrow keys for history.

```
2.7.16 (v2.7.16:413a49145e, Mar  4 2019, 01:30:55) [MSC v.1500 32 bit (Intel)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate code.py]
1111100000
>>>
```

Problem 2

```
H(str_to_bin('My name is Reid Nagurka and I like pho.'))
```

Search | Stack Data | Python Shell

Commands execute without debug. Use arrow keys for history.

```
2.7.16 (v2.7.16:413a49145e, Mar  4 2019, 01:30:55) [MSC v.1500 32 bit (Intel)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate code.py]
111101101111000000110101000010110010000000000010011101101
>>>
```

Problem 3

The most significant output is the encrypted sequence of characters, followed by the plaintext (generated by decrypting with same pad). These are the last two lines of the output

```
s = "Hello, my name is Reid Nagurka"
bs = str_to_bin(s)
l = lfsr([2,4,5,7,11,14])
l.seed(int_to_bin(77))
pad = l.gen(len(bs), 1000)
print pad
cipher = enc_pad(bs, pad)
print cipher

print bin_to_str(cipher)
print bin_to_str (enc_pad(cipher,pad))
```

Search | Stack Data | Python Shell

Commands execute without debug. Use arrow keys for history. Options

```
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate code.py]
0011111010011100000110110011000111000101001000110000111011101001110010101010111110100 1
0111011011111001011101110101110110101010000011110010111010000100101100111000111111001 0
v□w]□SI.□□□□-□VTR0e>JSISTX□~□□□DC2□FS□
Hello, my name is Reid Nagurka
>>>
```

Problem 4

```
(n, e, d) = GenRSA()
#print (n, e , d)
print n
print e
print d
```

Search | Stack Data | Python Shell

Commands execute without debug. Use arrow keys for history. Options

```
2.7.16 (v2.7.16:413a49145e, Mar  4 2019, 01:30:55) [MSC v.1500 32 bit (Intel)]
Python Type "help", "copyright", "credits" or "license" for more information.
>>> [evaluate code.py]
746505067986373517228497126909458457472674438766651531276820917053500500166360774968676093905286432855332844157830731782765735055165390555323484419653476190578501671540346324027608954065614375081737742800902851727924458060266773576964983070175257251197471772385217929846886781261384906869707429501527858089415
298602027194549406891398850763783382989069775506660612510728366821400200066544309987470437562114573142133137663132292713106294022066156222129393767861390401342494698249600796100815823895427296812087392622431342524750801780926563768597545218780344480723845536852944000819178315478183968481071538655869042438 13
```

Problem 5

The hash of the plaintext message is used because it can be used to verify the signature, just as the message would be. This ensures that if the verification fails, the message is not decrypted in the process of verification, only the hash is seen. In practice, the signature is attached to the message and the signature is verified first before a user can decrypt the actual message.

```
(n, e, d) = GenRSA()
#if encrypted with private, can only be decrypted with public
#if encrypted with public, can only be decrypted with private
#both parties have plaintext and same hashing algorithm
plaintext = 'Reid Nagurka rnagurka@email.wm.edu'
#Alice hashes plaintext
hashed = H(plaintext)
print "hashed: ", hashed
plain_bin = str_to_bin(hashed)
plain_int = bin_to_int(plain_bin)

#Alice encrypts hash with her private key d
signature = pow(plain_int, d, n)
print "signature: ", signature

#Bob decrypts signature with public key:
message = pow(signature, e, n)
print "decrypted with public key: ", message

#compare the resulting message with the hashed computed using the hashing function:

#the hashed message with e is the same as the original hash, meaning verification is good
message_bin = int_to_bin(message)
message_str = bin_to_str(message_bin)
print "newly computed hash: ", message_str
#in practice, the signature would be attached to the message
```

Search | Stack Data | Python Shell

Commands execute without debug. Use arrow keys for history. Options

```
hashed:  yCgq<EOT>}:5>3`Ml;\0002_8~(<NAK><DC1>*"wD%

io      ,sz'Nh <ETX>`F
;^-~*q ddMD-<ETB>
signature:
10195582918871834050818590519207249952169928065007622472908287273044438173280401106627250434488927914826350732301123560829911268360227380
56795578382323304120909381714473201132810846476038029918001819138956149609834951678749492010711255701210162925464124506942558490594329910
74931758568952325488702753584877992
decrypted with public key:
34429242264965345584308670312641206016364604443821766481475149320203198388176587424980279357306506241575140451673221831202243682003688 7
newly computed hash:  yCgq<EOT>}:5>3`Ml;\0002_8~(<NAK><DC1>*"wD%

io      ,sz'Nh <ETX>`F
;^-~*q ddMD-<ETB>
>>>
```