

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Date : 30/03/2022 | | | | | | | |
|  | CSPC63: **Principles of Cryptography**  **Assignment - 1** | | | | | |  |
|  |  | | | | |  | |
|  | | |  |  | | | |
|  | | | Roll no. : 106119100Name : Rajneesh PandeySection : CSE-B |  | | | |
|  | |  | | |  | | |

Write a program to determine if a number is quadratic residue to the modulus m using Jacobi.

**Explanation:**

Basic Definitions

**Text

Description automatically generated**

**Text

Description automatically generated**

**Graphical user interface, text, application, email

Description automatically generated**

**Graphical user interface, text, application

Description automatically generated**

**Text

Description automatically generated**

Now, I implemented the message encryption using the above property.

In this Code generates a random message and encrypts it using the **Goldwasser-Micali-cryptosystem**.

We are given the public key (N,a), where N=p\*q and a=−1.

In order to decrypt the encoded message (c1,c2,…,cn), we require the private key (p,q). If we have the private key, we can decrypt ci by checking if ci is a quadratic residue modulo N, i.e., if there exists an integer x such that:

**x2≡ ci mod N**

If ci is a quadratic residue, then we set bit mi=1. Otherwise, mi=0. Doing this for all bits gives us the original message (m1,m2,…,mn)

We can check if ci is a quadratic residue by calculating

**ci(p−1)/2≡1 mod p** and  **c(q−1)/2≡1modq**.

However, since ci, p and q are all large integers, this will likely give us an overflow error.

So,

An alternative method to check if it is a ci is a quadratic residue is by calculating the Jacobi symbol. The Jacobi symbol **(a/p)** is the product of Legendre symbols for each prime factorization of p.

The Legendre symbol is defined as follows:

If the Jacobi symbol for an encrypted bit ci is 1, then we know that the decrypted bit mi is 0

If the Jacobi symbol for an encrypted bit ci is -1, then we know that the decrypted bit mi is 1

Code :

from pwn import \*

import pwn

import json

from cypari import pari

# Connect to server

pwn.context.log\_level = 'error'

sh = pwn.remote('localhost', 8000)

# Receive N

N = int(sh.recvuntil(b'\n'))

print("N: ", N)

# Compute the two primfactors using cypari

factors = pari.factor(N)

p = int(factors[0][0])

q = int(factors[0][1])

print("Prime factors are p={} and q={}".format(p,q))

# The jacobi symbol is a generalization of the Legendre symbol which we could also use here

# For the jacobi symbol (a,p) we have the definition:

# 0 - if a = 0 mod(p)

# 1  if a != 0 mod(p) and a is a quadratic residue

# -1 if a 1= 0 mod(p) and a is a quadratic non-residue

# Now that we have p and q, we can decrypt the bits as using the jacobi symbol to check if the encoded bit is a quadratic

# residue of mod n.

# If the jacobi symbol for an encrypted bit is 1, then we know that the decrypted bit is 0

# If the jacobi symbol for an encrypted bit is -1, then we know that the decrypted bit is 1

# It will never be 0 due to the way that we calculate the encryption

def jacobi(a, n):

    if a == 0:

        return 0

    if a == 1:

        return 1

    e = 0

    a1 = a

    while a1%2==0:

        e += 1

        a1 = a1 // 2

    assert 2 \*\* e \* a1 == a

    s = 0

    if e%2==0:

        s = 1

    elif n % 8 in {1, 7}:

        s = 1

    elif n % 8 in {3, 5}:

        s = -1

    if n % 4 == 3 and a1 % 4 == 3:

        s \*= -1

    n1 = n % a1

    if a1 == 1:

        return s

    else:

        return s \* jacobi(n1, a1)

# we compute both strings and throw away the empty one

p\_string = ""

q\_string = ""

# From the source code, we know that we expect a message of length 20

for i in range(20):

    p\_list = []

    q\_list = []

    # Receive the token from the server and turn into a list of encoded bits

    token = sh.recvuntil(b'\n').decode('utf-8')

    print(token)

    j\_text = token.replace(' ', ',')

    bit\_enc\_list = json.loads(j\_text)

    # Compute the Jacobi symbol for each bit

    for bit\_enc in bit\_enc\_list:

        # Encoded bit is 0 if jacobi(b, q) == 1 if it is -1, it is 0

        # Basically this is checking if c\*\*((p-1)/2) is congruent to 1 mod p (and c\*\*((q-1)/2) is congruent to 1 mod q)

        bit\_p = 1 if jacobi(bit\_enc, p) == -1 else 0

        bit\_q = 1 if jacobi(bit\_enc, q) == -1 else 0

        p\_list.append(bit\_p)

        q\_list.append(bit\_q)

    # Turn the bit array into an int

    p\_int = int("".join(str(i) for i in p\_list),2)

    q\_int = int("".join(str(i) for i in q\_list),2)

    # and the int into a char which we append to the string

    p\_string = p\_string + chr(p\_int)

    q\_string = q\_string + chr(q\_int)

# Throw away the empty string and send the decoded string to the server

if not p\_string[0] == '\x00':

    msg = p\_string.format()

else:

    msg = q\_string.format()

print('Decoded string: {}'.format(msg))

sh.sendline(msg.encode('utf-8'))

# Receive empty line before our flag

sh.recvuntil(b'\n')

flag = sh.recvuntil(b'\n')

print(flag.decode('utf-8'))

Output

N:  259100079009838173106812091958653713911

Prime factors are p=15357312123475845169 and q=16871447094818545319

[39259593559709653362645902811241921654 121892299866367682713290812445544344622 8437315095678660677025386562013249073 751152165047126381287736864484418462 220489950644924241273479761042188937432 23858823754328283148232156800602776959 68812489863720154817707081138299449141 78216044892772585735835429201467583125]

[112334069051379613428555977526235174720 184212529387103308590636967000174097098 113740189312256781198955523449298058909 100139832470861551784621537646843408972 224262741903501195530916944370104893314 9866392024488510262419687887760181403 251902026551808042030796200881795197847 19251612716260270418662134326518949476]

[105882953965448588332394425477142882515 89031799047847151294790281930568200271 141864320124580557641474026006489365952 137194620160428643738305358302423719797 144896053069467966971006739094495876878 231739504999638159558862450301768809289 33946676506134688367193051887836400451 173712248780348905518968349109947842484]

[162341696134423705660325717445728669898 134536164576681218651035433032303373927 242602481140138420757162472888838154490 125703758031070240115664665661685497739 177323814810323621601833694566033023065 175950809364654832731232446433350130920 84401063636158356934126070421991488197 158189347871465316512082892818078015312]

[241062520112189506879500053094968476375 223149695440611347110726460097964670541 204519334637362170074852207412241454668 89974297694868741147220882740262936939 107759078594138943331433909500110142588 205824666481119559031061552796918172386 246127435028985457572873988198354520918 141650919665478271187047783134307666345]

[255805588183900627856467714352203098072 239541184521383640315636501386393948693 251242948835680370529926833750503022553 23588340787897503705873167288442569859 207584059663350246448337349883781703860 115558040446671862830401211862122363779 128584946844024265460286313174178472019 186170473261262891403760811677556238373]

[12579549411248946263122718661896278131 12757854185623984071959167906104014385 163407955436721339725235573266085957993 65001330048122979552333359964638254335 225770116515090813289034480040961228445 233978315005913608356174784126296094783 201895101773340631703967421655208585691 87152247531157499600243260323348901113]

[243238350037051792385531390328888770694 194919900821597496530898599127733793068 29873791238100781634122901571439248282 31103222119666969463199590695087104932 5014704544603436431931705039994479599 237084754469832041334229815026068157850 125048628169929371618971097537227383050 146269164168794607050572486730557236159]

[17176562466645879670762010581110743699 247582506571707374738895468417791185681 62959420570786751553149764947578641239 192410088111112227985507852432285824011 33220255709563728424441330377924502215 84910348535670969097080773612022081942 159734644184402716967677846308503156781 19281349982337778486314116064288035816]

[105900250696366213841349568086627897618 159397060670957694936306900960136148331 210906428034465628397400567292030366304 90945137226617465880779105637278017137 46759979675556882528583907524676264715 116836037258569271314190363493611184996 58287376755503082761864602305612237001 76153177017329226924410074831456798242]

[49775566445608823337277798110884255521 137762148157827654608445248685782956136 229835495480870991692265513809323358984 117220432214131404858080435135867718517 79157750567303707415372924127778293185 55839316623741860738799636293953562183 217628430731130514215591951425927927332 32833455945970831389114508707765573695]

[220395930980183375043043909765601510031 245814712267306870201008429679885669081 237025488376788306879032717679975021123 197625542039064591899294380902986424899 100175419575080215578875988438234338769 22351417423769319136900000371562006274 175094146828796325740399237379240525502 52650292432672715161085987624674513690]

[23959089326239610003666148525522687442 30310427391781824727972262683128440137 70303448095233396907020614044493638344 256910050389849475317548484642057447439 111760636336157949894868123975398610235 237765040524201178521896683618967255428 85477720490959060307682181629999073448 166332597873817222787716227473768494794]

[146009003897366202491524785750639611693 161727140175860418956915019630304427998 40280203003385227611499413984759210105 140242322639860428645384929329961670585 4398470761175630212103715878267366980 247012293580502060404909040870016088430 17773704406684770115718874737375893297 234700597159290547377549769633217929077]

[153050321492869564948991046477849307809 99400615742914036623646722430033535407 112728723140011199998126087314746111147 23134307122963421382395370742505429968 188986674931231752417375633429485374030 104823221320627209756254666000815160764 4001544543016435901549932334498919962 155550839058617310446813171264024843842]

[161980211647707657203092742290516658806 111803325918196090823857735024807917692 192542431523802866617279138972930457135 160540042563093284334921465291446924191 203690360964165439707094279876038688891 139415413417719251036038947845867489194 26343382927053502874563531957730201535 9628115518891506471866405275545532158]

[9368857789248627963735955700233870023 209900203983811128782447210588595937430 215671348511730629300380522751622354684 236781633392716839728262590077268024636 216102005244313147909573987889487156356 141107995679835178598448313942540159821 206992234398155708478109593559308670305 132139793578690514642130464614014267134]

[54764619342848485365068363375072771014 105892991469353215521498592547600561343 151426117639959014232014996058378628565 27501599412772669402141522327652478748 63978307048326887578899447930393357348 168567115728400050827486584733670092590 136101968758241940265571506841486267417 119661689577316952303797701365682231627]

[205071547519221612076367186718270130198 70546778005184954465094362986900861475 48085943458860192498985351411156300936 60200575903545078787887735441963236174 214836625580038766163794286221250923061 38599872384671320838622426342430199032 217244589224248854904512479985268642715 244697986685504838495367689455153569024]

[179396166086294255072359490601179544285 226512771116773522077333711264103121598 118663477489157121611468629786227439416 171475020890511543684842623444746327108 229228392088061823996493863012639786890 88914741445736416139938400553337296995 143046239008564465159707347637928231493 216853660128160819202531088515160129212]

Decoded string: btzwMg4QrZlIBJBXawyX

b'flag{Oh\_NO\_aT\_LEast\_mY\_AlGORithM\_is\_ExpanDiNg}\n'

Process finished with exit code 0