# Homework 7

November 2, 2020

[7]:
```
########## Preamble:
def fastPowerSmall(g,A,N):
    a = g
    b = 1
    while A>0:
        if A % 2 == 1:
            b = b * a % N
        A = A//2
        a = a*a % N
    return b

def extendedEuclideanAlgorithm(a,b):
    u = 1
    g = a
    x = 0
    y = b
    while true:
        if y == 0:
            v = (g-a*u)/b
            return [g,u,v]
        t = g%y
        q = (g-t)/y
        s = u-q*x
        u = x
        g = y
        x = s
        y = t

def findInverse(a,p):
    inverse = extendedEuclideanAlgorithm(a,p)[1] % p
    return inverse

def textToInt(words):
    number = 0
    i = 0
    for letter in words:
        number += ord(letter)*(256**i)
```

```
        i+=1
    return number

def intToText(number):
    words = ""
    while number>0:
        nextLetter = number % 256
        words += chr(nextLetter)
        number = (number-nextLetter)/256
    return words

def millerRabin(a,n):

    #first throw out the obvious cases
    if n%2 == 0 or extendedEuclideanAlgorithm(a,n)[0]!=1:
        return True

    #Next factor n-1 as 2^k m
    m = n-1
    k = 0
    while m%2 == 0 and m != 0:
        m = m//2
        k = k+1
    #Now do the test:
    a = fastPowerSmall(a,m,n)
    if a == 1:
        return False

    for i in range(0,k):
        if (a + 1) % n == 0:
            return False
        a = (a*a) % n

    #If we got this far a is not a witness
    return True

# This function runs the Miller-Rubin test on 20 random numbers between 2 and p-1.  If it returns true there is a probability of (1/4)^20 that p is prime.
def probablyPrime(p):
    for i in range(0,20):
        a = ZZ.random_element(2,p-1)
        if millerRabin(a,p):
            return False
    return True

def findPrime(lowerBound,upperBound):
    while True:
```

```
        candidate = ZZ.random_element(lowerBound,upperBound)
        if probablyPrime(candidate):
            return candidate
```

[8]:
```
########## Problem 1

#Helper Function
def factorOut2(n): #returns m,k such that n = m2^k and m is odd
    k = 0
    while n%2 == 0:
        n = n//2
        k = k+1
    return n,k

#####Part (a)
def legendreSymbol(a,p):
    #Make sure the base is even
    if p == 2:
        print("The base of a Legendre symbol must be odd!")
        return

    #Then use Euler's criterion
    else:
        a = a % p
        m = (p-1)//2
        return fastPowerSmall(a,m,p)


#####Part (b)
def jacobiSymbol(a,b):
    #print("(",a,",",b,")")
    #Make sure the base is even
    if b%2 == 0:
        print("The base of a Jacobi symbol must be odd!")
        return

    #The value only depends on the class of a modulo b:
    a = a%b
    #If a=-1, 0, 1, or 2 we can compute this directly using quadratic␣
 ↪reciprocity
    if a==b-1:
        #This depends on the congruence class of b modulo 4
        if (b%4)==1:
            return 1
        else:
            return -1
```

```
    if a==0:
        return 0

    if a==1:
        return 1

    if a==2:
        #This depends on the congruence class of b modulo 8
        if (b%8 == 1) or (b%8 == 7):
            return 1
        else:
            return -1

    #If the base is prime we can just compute a Legendre symbol using Euler's␣
 ↪formula
    if probablyPrime(b):
        return legendreSymbol(a,b)

    #If not we use quadratic reciprocity to flip the jacobi symbol.  Since the␣
 ↪base of the jacobi symbol must be even, we need to factor all powers of 2␣
 ↪out of a.
    m,k = factorOut2(a)

    #since a = m2^k then (a/b) = (m/b)(2/b)^k.  The even part is easy to␣
 ↪compute:
    evenPart = jacobiSymbol(2,b)**k

    #Using quadratic reciprocity we compute that the odd part is either (b/m)␣
 ↪or -(b/m) depending on the residues of b and m modulo 4
    if (m%4==1) or (b%4==1):
        oddPart = jacobiSymbol(b,m)
    else:
        oddPart = -jacobiSymbol(b,m)
    return evenPart*oddPart

#####Part (c): Testing
tests = [
    [8,15],
    [11,15],
    [12,15],
    [171337608,536134436237]
]

for pairs in tests:
    a = pairs[0]
    b = pairs[1]
    print("(",a,"/",b,") =",jacobiSymbol(a,b))
```

```
( 8 / 15 ) = 1
( 11 / 15 ) = -1
( 12 / 15 ) = 0
( 171337608 / 536134436237 ) = -1
```

[9]:
```
########## Problem 2

#####Part(a)

#I'm initializing p and q to -1 here so that we have the option of generating a␣
↪GM key with chosen primes  Notice if you choose primes the choice of b␣
↪doesn't matter
def generateGMKey(b, p=-1, q=-1):
    if p==-1:
        p = findPrime(2**(b-1),2**b)
        q = findPrime(2**(b-1),2**b)
    N = p*q

    #Find an a which is not a square is
    while True:
        a = ZZ.random_element(2,N)
        if (legendreSymbol(a,p) == p-1) and (legendreSymbol(a,q)==q-1):
            break

    publicKey = [N,a]
    privateKey = p
    return [publicKey,privateKey]

#####Part (b)
def GMEncrypt(publicKey,m):
    N = publicKey[0]
    a = publicKey[1]

    r = ZZ.random_element(isqrt(N)+1,N)
    if m:
        return (a*r*r) % N
    else:
        return (r*r) % N

def GMDecrypt(privateKey,c):
    #THe private key is just a prime p.
    if legendreSymbol(c,privateKey) == 1:
        return 0
    else:
        return 1
```

```
##### Part (c)
keys = generateGMKey(16)
m0 = 0
m1 = 1
c0 = GMEncrypt(keys[0],m0)
c1 = GMEncrypt(keys[0],m1)
print("ciphers are",c0,"and",c1)
mm0 = GMDecrypt(keys[1],c0)
mm1 = GMDecrypt(keys[1],c1)

print("0 decrpyted to",mm0,"and 1 decrypted to",mm1)

##### Part(d)
keys = generateGMKey(0,151,233)
cipher = 33482
message = GMDecrypt(keys[1],cipher)
print("The message in part (d) decrypted to",message)
```

```
ciphers are 2071547091 and 3043398832
0 decrpyted to 0 and 1 decrypted to 1
The message in part (d) decrypted to 1
```

[10]:

```
########## Poblem 3
#####Part (a)
def generateRSAKey(b):

    #Generate some primes
    p = findPrime(2^(b-1),2^b)
    q = findPrime(2^(b-1),2^b)
    N = p*q
    M = (p-1)*(q-1)
    #next lets find an encryption exponenet
    while True:
        e = ZZ.random_element(2,M-1)
        gcd = extendedEuclideanAlgorithm(e,M)
        if gcd[0]==1:
            d = gcd[1] % M
            break
    publicKey = [N,e]
    privateKey = [N,d]
    return[publicKey,privateKey]

#####Part (b)

def RSASign(signingKey,document):
    N = signingKey[0]
    d = signingKey[1]
```

```python
    signedDocument = fastPowerSmall(document,d,N)
    return signedDocument

def RSAVerify(verificationKey,document,signedDocument):
    N = verificationKey[0]
    e = verificationKey[1]
    if fastPowerSmall(signedDocument,e,N) == document:
        return True
    else:
        return False

#####Part (c)
print("Part (c)")

keys = generateRSAKey(16)
print("Here are the generated RSA Keys:",keys)
document = 314159
signedDocument = RSASign(keys[1],document)
v1 = RSAVerify(keys[0],document,document)
v2 = RSAVerify(keys[0],document,signedDocument)

print("Verifying the unsigned document:",v1)
print("Verifying the signed document:",v2)


#####Part (d)
print("Part (d)")
#RSA Verification Key
N = 
 ↪10522131111414083920142713395050769617442433657695255146537311896434311144361521056973720088
e = 
 ↪21016836287029986747723759800774101835281275854873180166245543058525887395193963077668602549
verificationKey = [N,e]
#Recieved Documents
D = 4459158569051973444519310560529993353156889234209074860197000813 7
D1 = 
 ↪33737713092926002711997965938686781423985198340985423980678740668540837920165714308759593604

#Let's read them:
DText = intToText(D)
D1Text = intToText(D1)
print("The document D said:",DText)
print("The document D' said:",D1Text)

#Here they are (potentially) signed:
```

```
DSig =
↪842666568816337596459314344145896463734741402996963185688465588713546839926649111577534844
D1Sig =
↪9059280931350999147776789854356125281873028523394670827691831663199736194473764728663367105

#Let's verfify the signatures:
V = RSAVerify(verificationKey,D,DSig)
V1 = RSAVerify(verificationKey,D1,D1Sig)

print("The signature for D is:",V)
print("The signature for D' is:",V1)
```

```
Part (c)
Here are the generated RSA Keys: [[2179698067, 432039971], [2179698067,
1678668171]]
Verifying the unsigned document: False
Verifying the signed document: True
Part (d)
The document D said: I am your Professor Gabriel
The document D' said: Ignore the other message, I am the real professor!
The signature for D is: False
The signature for D' is: True
```

[11]:
```
########## Problem 4

#####Part (a)
def generateElgamalKey(p,g):
    a = ZZ.random_element(2,p-1)
    #a = 15140
    A = fastPowerSmall(g,a,p)
    signingKey = [a,p,g]
    verificationKey = [A,p,g]
    return [signingKey,verificationKey]

#####Part (b)
def elgamalSign(signingKey,document):
    a = signingKey[0]
    p = signingKey[1]
    g = signingKey[2]

    #k = 10727
    while True:
        k = ZZ.random_element(1,p-1)
        if extendedEuclideanAlgorithm(k,p-1)[0]==1:
            break
    kinv = findInverse(k,p-1)
    S1 = fastPowerSmall(g,k,p)
```

```python
    S2 = ((document - a*S1)*kinv) % (p-1)
    signedDocument = [S1,S2]
    return signedDocument

def elgamalVerify(verificationKey,document,signedDocument):
    A = verificationKey[0]
    p = verificationKey[1]
    g = verificationKey[2]

    S1 = signedDocument[0]
    S2 = signedDocument[1]

    V1 = (fastPowerSmall(A,S1,p)*fastPowerSmall(S1,S2,p)) % p
    V2 = fastPowerSmall(g,document,p)
    print(V1,V2)
    if V1==V2:
        return True
    else:
        return False

#####Part (c)

#The following helper function is crucial.  It takes as input a,c,N, and
 ↪returns the set of solutions to ax = c mod N.  This could have been written
 ↪back in HW2.
def solveLinearCongruence(a,c,N):
    G,u,v = extendedEuclideanAlgorithm(a,N)
    #First check if there are any solutions using HW2#7
    if c%G!=0:
        print("No solutions to",a,"x =",c,"mod",N)
        return -1

    #Otherwise we find one solution a0
    l = c//G
    a0 = (u*l) % N

    #Now we can iterate through all of them.
    solutionsList = [a0]
    for i in range(0,G-1):
        a0 = (a0 + N//G) % N
        solutionsList.append(a0)

    #now we have our list!
    return solutionsList

def stealElgamalSignature(verificationKey,D,Dsig,D1,D1sig):
    #Let's extract all the info we need
```

```
    A,p,g = verificationKey
    S1,S2 = Dsig
    S1a,S2a = D1sig

    #Make sure the mistake was made:
    if S1 != S1a:
        print("The same signing exponent wasn't used")
        return -1

    #Then let's find candidates for k inverse
    d = (D - D1) % (p-1)
    s = (S2 - S2a) % (p-1)
    kList = solveLinearCongruence(d,s,p-1)

    #One of these is k inverse
    k = 0
    for kinv in kList:
        if fastPowerSmall(S1,kinv,p)==g:
            k = findInverse(kinv,p-1)
            break

    #Now we've found k, we use a similar philosophy to find a.  We will need to␣
 ↪set up an equations
    d1 = S1 % (p-1)
    s1 = (D - k*S2) % (p-1)
    aList = solveLinearCongruence(d1,s1,p-1)

    #One of these is a
    for a in aList:
        if fastPowerSmall(g,a,p) == A:
            return a


#####Part (d)
print("Part (d)")
p = 3700273081
g = 7
keys = generateElgamalKey(p,g)
D = 314159
Dsig = elgamalSign(keys[0],D)
print("Signed document",D,"to recieve",Dsig)
D1 = elgamalVerify(keys[1],D,Dsig)
print("Verification =",D1)


#####Part (e)
print("Part (e)")
```

```
verificationKey = [185149, 348149, 113459]
D = 153405
Dsig = [208913,209176]
D1 = 127561
D1sig = [208913,217800]

print("Attempting to extract signing key...")
print("Returns: ",stealElgamalSignature(verificationKey,D,Dsig,D1,D1sig))
```

```
Part (d)
Signed document 314159 to recieve [2066955618, 2940130549]
912253726 912253726
Verification = True
Part (e)
Attempting to extract signing key…
Returns:  72729
```

[13]:
```
########## Problem 5
def generateDSAKey(p,q,g):
    a = ZZ.random_element(2,q-1)
    A = fastPowerSmall(g,a,p)
    signingKey = [a,p,q,g]
    verificationKey = [A,p,q,g]
    return [signingKey,verificationKey]

def DSASign(signingKey,document):
    a = signingKey[0]
    p = signingKey[1]
    q = signingKey[2]
    g = signingKey[3]
    k = ZZ.random_element(1,q)
    kinv = findInverse(k,q)
    S1 = fastPowerSmall(g,k,p) % q
    S2 = ((document + a*S1)*kinv) % q
    signedDocument = [S1,S2]
    return signedDocument

def DSAVerify(verificationKey,document,signedDocument):
    A = verificationKey[0]
    p = verificationKey[1]
    q = verificationKey[2]
    g = verificationKey[3]
    S1 = signedDocument[0]
    S2 = signedDocument[1]
    S2inv = findInverse(S2,q)
    V1 = (document*S2inv) % q
    V2 = (S1*S2inv) % q
```

```
    W = (fastPowerSmall(g,V1,p)*fastPowerSmall(A,V2,p) % p) % q
    if W == S1:
        return True
    else:
        return False

#####Part (c)
print("Part (c)")
p = 48731
q = 443
gPrimitive = 7
g = fastPowerSmall(g,(p-1)//q,p)
print("By raising the primitive root to the",p-1,"/",q,"we get an␣
 ↪element",g,"of order",q)

print("Part (d)")
D = 314
keys = generateDSAKey(p,q,g)
Dsig = DSASign(keys[0],D)
print("Signed document is",Dsig)
Verified = DSAVerify(keys[1],D,Dsig)
print("Verifying DSig:",Verified)
```

```
Part (c)
By raising the primitive root to the 48730 / 443 we get an element 1024 of order
443
Part (d)
Signed document is [81, 304]
Verifying DSig: True
```

[0]:

```

```