Takehome2Public

December 21, 2020

[1]:
```python
########## 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 getBinary(A):
    binaryList = []
    while A>0:
        if A%2 == 0:
            binaryList.append(0)
        else:
            binaryList.append(1)
        A = math.floor(A/2)
    return binaryList

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 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

### We will need findPrime and all its dependencies
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

#####Part (b)
# 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

#####Part (c)
def findPrime(lowerBound,upperBound):
    while True:
        candidate = ZZ.random_element(lowerBound,upperBound)
        if probablyPrime(candidate):
            return candidate

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
```

[2]:
```
#This just checks that the discriminant is nonzero
def isCurve(E,p=0):
    A,B = E
    Delta = 4*A**3 + 27*B**2
    if p!=0:
        Delta = Delta % p
    if Delta!=0:
        return True
    else:
        return False
```

```python
#This just checks if the point is on the curve
def onCurve(E,P,p=0):
    if P=='O':
        return True
    A,B = E
    x,y = P
    LHS = y**2
    RHS = x**3 + A*x + B
    if p!=0:
        LHS = LHS % p
        RHS = RHS % p
    if LHS==RHS:
        return True
    else:
        return False

primeList = [3,5,7,11,13,17,19]
E = [3,2]
P = [3,5]
for p in primeList:
    print("E a curve over",p,":",isCurve(E,p))
    if(isCurve(E,p)):
        print("P is on E over",p,":",onCurve(E,P,p))
        print("O is on E over",p,":",onCurve(E,'O',p))

pointList = ['O']
for i in range(0,6):
    for j in range(0,6):
        if onCurve(E,[i,j],7):
            pointList.append([i,j])
print(pointList)
```

```
E a curve over 3 : False
E a curve over 5 : True
P is on E over 5 : False
O is on E over 5 : True
E a curve over 7 : True
P is on E over 7 : False
O is on E over 7 : True
E a curve over 11 : True
P is on E over 11 : False
O is on E over 11 : True
E a curve over 13 : True
P is on E over 13 : True
O is on E over 13 : True
E a curve over 17 : True
```

```
P is on E over 17 : False
O is on E over 17 : True
E a curve over 19 : True
P is on E over 19 : False
O is on E over 19 : True
['O', [0, 3], [0, 4], [2, 3], [2, 4], [4, 1], [5, 3], [5, 4]]
```

[3]:

```
def addPoints(E,P,Q,p):
    #First see if you're adding O
    if P=='O':
        return Q
    if Q=='O':
        return P
    #Otherwise let's extract some data
    A,B = E
    x1,y1 = P
    x2,y2 = Q
    #make sure everything is reduced mod p
    x1 = (x1 % p)
    x2 = (x2 % p)
    y1 = (y1 % p)
    y2 = (y2 % p)

    #If the points are inverses we just return the point at infinity
    if y1!=y2 and x1==x2:
        return 'O'

    #Otherwise we begin by computing the slope of the line
    if(x1==x2):
        L = ((3*x1**2 + A)*findInverse(2*y1,p)) % p
    else:
        L = ((y2-y1)*findInverse(x2-x1,p)) % p

    #Finally compute coords of the new points
    x3 = (L**2 - x1 - x2) % p
    y3 = (L*(x1-x3) - y1) % p
    return [x3,y3]

print("Curve: y^2 = x^3 + 3x + 8 over F_{13}")
print("P = (9,7) and Q= (1,8)")
E = [3,8]
p = 13
P = [9,7]
Q = [1,8]
print("P+Q=",addPoints(E,P,Q,p))
print("2P=",addPoints(E,P,P,p))
print("O+Q=",addPoints(E,'O',Q,p))
```

```
print("")
print("Curve: y^2 = x^3 + 3x + 2 over F_7: Multiplication Table")
print("")
for P in pointList:
    for Q in pointList:
        R = addPoints(E,P,Q,7)
        if R=='O':
            print("[OOOO]",end='')
        else:
            print(R,end=''),
    print("")
print("")
print("Curve: y^2 = x^3 + 231x + 473 over F_{17389}")
print("P =  (11259, 11278) and Q = (11017,14673)")
E = [231,473]
p = 17389
P = [11259,11278]
Q = [11017,14673]
print("P+Q=",addPoints(E,P,Q,p))
print("2Q=",addPoints(E,Q,Q,p))
print("3P=",addPoints(E,P,addPoints(E,P,P,p),p))
print("")

print("Mistake Q = (11017,14673)")
E = [231,473]
p = 17389
P = [11259,11278]
Q = [11017,14637]
print("P+Q=",addPoints(E,P,Q,p))
print("2Q=",addPoints(E,Q,Q,p))
print("3P=",addPoints(E,P,addPoints(E,P,P,p),p))
print("")
```

```
Curve: y^2 = x^3 + 3x + 8 over F_{13}
P = (9,7) and Q= (1,8)
P+Q= [2, 10]
2P= [9, 6]
O+Q= [1, 8]

Curve: y^2 = x^3 + 3x + 2 over F_7: Multiplication Table

[OOOO][0, 3][0, 4][2, 3][2, 4][4, 1][5, 3][5, 4]
[0, 3][2, 3][OOOO][5, 4][0, 4][5, 3][2, 4][4, 6]
[0, 4][OOOO][2, 4][0, 3][5, 3][4, 6][4, 1][2, 3]
[2, 3][5, 4][0, 3][4, 6][OOOO][2, 4][0, 4][4, 1]
[2, 4][0, 4][5, 3][OOOO][4, 1][5, 4][4, 6][0, 3]
[4, 1][5, 3][4, 6][2, 4][5, 4][0, 3][2, 3][0, 4]
```

```
[5, 3][2, 4][4, 1][0, 4][4, 6][2, 3][5, 4][0000]
[5, 4][4, 6][2, 3][4, 1][0, 3][0, 4][0000][5, 3]

Curve: y^2 = x^3 + 231x + 473 over F_{17389}
P =  (11259, 11278) and Q = (11017,14673)
P+Q= [12613, 2831]
2Q= [522, 6187]
3P= [13395, 14468]

Mistake Q = (11017,14673)
P+Q= [6978, 16749]
2Q= [13720, 11471]
3P= [13395, 14468]
```

[4]:
```
#Computes nP applying the double and add algorithm
def doubleAndAdd(P,n,E,p):
    #First find the binary expansion of n
    nBinary = getBinary(n)
    r = len(nBinary)

    #compute doubles of p and add them to a list
    multiplesOfP = [P]
    for i in range(0,r):
        Qi = addPoints(E,multiplesOfP[i],multiplesOfP[i],p)
        multiplesOfP.append(Qi)

    #Start with Q as the identity
    Q = 'O'
    for i in range(0,r):
        if nBinary[i]==1:
            Q = addPoints(E,multiplesOfP[i],Q,p)
    return Q

#An O(1) storage variant
def doubleAndAddSmall(P,n,E,p):
    Q = 'O'
    while n>0:
        if n%2 == 1:
            Q = addPoints(E,Q,P,p)
        n = n//2
        P = addPoints(E,P,P,p)
    return Q

#Building a ternrary variant.
def getTernary(n):
    nTernary = getBinary(n)
```

```
    nTernary.append(0)
    r = len(nTernary)

    #now spread out the nonzero elements by putting in minus signs.
    s = 0
    t = 0
    for i in range(0,r):
        if nTernary[i] == 1:
            t = t + 1
        if nTernary[i] == 0:
            if t-s>1:
                nTernary[s] = -1
                for j in range(s+1,t):
                    nTernary[j] = 0
                nTernary[t] = 1
                s = i
                t = i+1
            else:
                t = s = i+1
    #make sure the most significant digit is a 1
    if nTernary[-1]==0:
        nTernary.pop()
    return nTernary

#Inverting a point just inverts the y coordinate
def invertPoint(P,p):
    if P=='O':
        return P
    else:
        x,y = P
        return [x,p-y]

def doubleAndAddTernary(P,n,E,p):
    #First find the ternary expansion of n
    nTernary = getTernary(n)
    r = len(nTernary)

    #compute doubles of p and add them to a list
    multiplesOfP = [P]
    for i in range(0,r):
        Qi = addPoints(E,multiplesOfP[i],multiplesOfP[i],p)
        multiplesOfP.append(Qi)

    #Start with Q as the identity
    Q = 'O'
    for i in range(0,r):
        if nTernary[i]==1:
```

```
            Q = addPoints(E,multiplesOfP[i],Q,p)
        elif nTernary[i]==-1:
            #In this case me must subract the multiple of P
            Q = addPoints(E,invertPoint(multiplesOfP[i],p),Q,p)
    return Q

def doubleAndAddTernarySmall(P,n,E,p):
    #First find the ternary expansion of n
    nTernary = getTernary(n)
    r = len(nTernary)

    Q = 'O'
    for i in nTernary:
        if i==1:
            Q = addPoints(E,P,Q,p)
        elif i==-1:
            Q = addPoints(E,invertPoint(P,p),Q,p)
        P = addPoints(E,P,P,p)
    return Q
E = [14,19]
p = 3623
n = 947
P = [6,730]

print(doubleAndAdd(P,n,E,p))
print(doubleAndAddSmall(P,n,E,p))
print(doubleAndAddTernary(P,n,E,p))
print(doubleAndAddTernarySmall(P,n,E,p))
print("")
E = [143,367]
p = 613
P = [195,9]
n = 23

print(doubleAndAdd(P,n,E,p))
print(doubleAndAddSmall(P,n,E,p))
print(doubleAndAddTernary(P,n,E,p))
print(doubleAndAddTernarySmall(P,n,E,p))
```

```
[3492, 60]
[3492, 60]
[3492, 60]
[3492, 60]

[485, 573]
[485, 573]
[485, 573]
```

```
[485, 573]
```

[5]:
```
def generateEllipticCurveAndPoint(p):
    while True:
        #randomly choose a point and an A value
        x = ZZ.random_element(1,p-1)
        y = ZZ.random_element(1,p-1)
        A = ZZ.random_element(1,p-1)

        #Generate B from the elliptic curve equation y^2 = x^3 + Ax + B
        B = (y**2 - x**3 - A*x) % p
        P = [x,y]
        E = [A,B]

        #Double check that the discriminant is nonzero
        if isCurve(E,p):
            return [E,P]

E,P = generateEllipticCurveAndPoint(13)
print(E,P)
E,P = generateEllipticCurveAndPoint(1999)
print(E,P)
```

```
[6, 4] [11, 7]
[1147, 1993] [745, 1350]
```

[6]:
```
def MVParameterCreation(b):
    while True:
        p = findPrime(2**(b-1),2**b)
        E,P = generateEllipticCurveAndPoint(p)
        #Let's make sure the P isn't an order 2 point
        if P[1]!=0:
            return [E,P,p]

def MVKeyCreation(pubParams):
    E,P,p = pubParams
    while True:
        n = ZZ.random_element(2,p-1)
        Q = doubleAndAddTernarySmall(P,n,E,p)
        if Q!='O' and Q[1]!=0:
            return[n,Q]

def MVEncrypt(pubParams,m1,m2,publicKey):
    #unpack everything we've been passed
    E,P,p = pubParams
    Q = publicKey
```

```
    #Do the elliptic curve computations
    while True:
        k = ZZ.random_element(2,p-1)
        R = doubleAndAddTernarySmall(P,k,E,p)
        S = doubleAndAddTernarySmall(Q,k,E,p)
        if R!='O' and S!='O':
            x,y = S

            #These coordinates form the basis of a symmetric cipher, so they␣
↪should be nonzero
            if x!=0 and y!=0:
                c1 = (x*m1) % p
                c2 = (y*m2) % p
                return [R,c1,c2]

    #Use the point S to mask encrypt message with a symmetric cipher.  (S is␣
↪the DH shared secret).


def MVDecrypt(pubParams,cipherText,privateKey):
    #unpack everything we've been passed
    E,P,p = pubParams
    R,c1,c2 = cipherText
    n = privateKey

    #First compute the Diffie-Hellman shared secret.
    T = doubleAndAddTernarySmall(R,n,E,p)
    x,y = T

    #The coords of T is the symmetric key.  We must invert it.
    xinv = findInverse(x,p)
    yinv = findInverse(y,p)

    #The message is then easily retrieved
    m1 = (xinv*c1)%p
    m2 = (yinv*c2)%p
    return[m1,m2]

######Testing
pubParams = MVParameterCreation(32)
print("Parameters for an elliptic curve, [E,P,p] =",pubParams)
privateKey,publicKey = MVKeyCreation(pubParams)
print("My private key is:",privateKey)
print("My public key is:",publicKey)
m1 = 314159
m2 = 8675309
cipherText = MVEncrypt(pubParams,m1,m2,publicKey)
```

```python
print("Message encrypted to:",cipherText)
decryption = MVDecrypt(pubParams,cipherText,privateKey)
print("Decrypted to",decryption)
```

```
Parameters for an elliptic curve, [E,P,p] = [[506843209, 930402926],
[1668314663, 1216312496], 2179382729]
My private key is: 1954708708
My public key is: [783819120, 1409060046]
Message encrypted to: [[1556511718, 423662511], 1481438261, 921709318]
Decrypted to [314159, 8675309]
```

[0]:

```python

```