Project2ImplementationSolutions

December 19, 2021

[1]:
```
########## Problem 0: Preamble

########## 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

#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

#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

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
```

```python
    y3 = (L*(x1-x3) - y1) % p
    return [x3,y3]
```

[2]:
```python
########## Problem 1: Double and Add

##### Part (a)

#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

##### Part(b)

#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

##### Part (c)(i)
E = [14,19]
p = 3623
P = [6,730]
n = 947

print("Comuting 947*(6,730) on y^2 = x^3 + 14x + 19 over the prime 3623 using...
 ↪")
print("doubleAndAdd:", doubleAndAdd(P,n,E,p))
```

```
print("doubleAndAddSmall:", doubleAndAddSmall(P,n,E,p))
print("============================")

##### Part (c)(ii)
E = [143,367]
p = 613
P = [195,9]
n = 23

print("Computing 23*(195,9) on y^2 = x^3 + 143x + 367 over the prime 613 using..
 ↪.")
print("doubleAndAdd:", doubleAndAdd(P,n,E,p))
print("doubleAndAddSmall:", doubleAndAddSmall(P,n,E,p))
```

```
Comuting 947*(6,730) on y^2 = x^3 + 14x + 19 over the prime 3623 using…
doubleAndAdd: [3492, 60]
doubleAndAddSmall: [3492, 60]
============================
Computing 23*(195,9) on y^2 = x^3 + 143x + 367 over the prime 613 using…
doubleAndAdd: [485, 573]
doubleAndAddSmall: [485, 573]
```

[3]:
```
########### Problem 2

##### Part (a)

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]

##### Part(b)(i)
E,P = generateEllipticCurveAndPoint(13)
print("Generated E over F_13 given by ((( y^2 = x^3 + ",E[0],"x + ",E[1],"))),␣
 ↪together with the point P = ",P)
print("Checking that P is on E...",onCurve(E,P,13))
```

```
print("==============================================")

##### Part (b)(ii)
E,P = generateEllipticCurveAndPoint(1999)
print("Generated E over F_1999 given by ((( y^2 = x^3 + ",E[0],"x +␣
 ↪",E[1],"))), together with the point P = ",P)
print("Checking that P is on E...",onCurve(E,P,1999))
```

```
Generated E over F_13 given by ((( y^2 = x^3 +  8 x +  10 ))), together with the
point P =  [11, 8]
Checking that P is on E… True
==============================================
Generated E over F_1999 given by ((( y^2 = x^3 +  192 x +  1146 ))), together
with the point P =  [1055, 258]
Checking that P is on E… True
```

[4]:
```
########## Problem 3

##### Part 3(a)
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]

##### Part 3(b)
def MVKeyCreation(pubParams):
    E,P,p = pubParams
    while True:
        n = ZZ.random_element(2,p-1)
        Q = doubleAndAddSmall(P,n,E,p)
        if Q!='O' and Q[1]!=0:
            return[n,Q]

##### Part 3(c)
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 = doubleAndAddSmall(P,k,E,p)
        S = doubleAndAddSmall(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:
                #Use the point S to mask encrypt message with a symmetric␣
↪cipher.  (S is the DH shared secret).
                c1 = (x*m1) % p
                c2 = (y*m2) % p
                return [R,c1,c2]

##### Part 3(d)
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 = doubleAndAddSmall(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]
```

[0]:

```
```