Final

December 21, 2020

[1]:
```
##########Preamble
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 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 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
```

```
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]

def invertPoint(P,p):
    if P=='O':
        return P
    else:
        x,y = P
        return [x,p-y]

def getBinary(A):
    binaryList = []
    while A>0:
        if A%2 == 0:
            binaryList.append(0)
        else:
            binaryList.append(1)
        A = (A//2)
    return binaryList

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

### 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
```

[3]:
```
##########Problem 1

#####Here is an adjusted version of addPoints which keeps in mind that the␣
↪modulus may not be prime and returns the discovered factorization if␣
↪addition fails.
def addPointsAdjusted(E,P,Q,N):
    #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 % N)
    x2 = (x2 % N)
    y1 = (y1 % N)
    y2 = (y2 % N)

    #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):
        gcdPlus = extendedEuclideanAlgorithm(2*y1,N)
        #We make sure we can divide by y1 first.  If not we're happy!
        if gcdPlus[0]!=1:
            return ["Factored",gcdPlus[0]]
        else:
            L = ((3*x1**2 + A)*(gcdPlus[1]%N)) % N
    else:
        gcdPlus = extendedEuclideanAlgorithm(x2-x1,N)
        #We make sure we can divide by x2-x1 first
        if gcdPlus[0]!=1:
```

```
            return ["Factored",gcdPlus[0]]
        else:
            L = ((y2-y1)*(gcdPlus[1]%N)) % N

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

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

    Q = 'O'
    for i in nTernary:
        if i==1:
            Q = addPointsAdjusted(E,P,Q,N)
        elif i==-1:
            Q = addPointsAdjusted(E,invertPoint(P,N),Q,N)
        #Let's check if the addition step factored p
        if Q[0]=="Factored":
            return Q
        P = addPointsAdjusted(E,P,P,N)
        #Same check for doubling P
        if P[0]=="Factored":
            return P
    return Q

def LenstraFactor(N,upperBound = -1,numberOfCurves = -1):
    n = 0
    #Loop around various elliptic curves and points
    while True:
        print("Trying a new curve")
        E,P = generateEllipticCurveAndPoint(N)
        for j in range(2,upperBound):
            P = doubleAndAddTernarySmallAdjusted(P,j,E,N)
            if P[0]=="Factored":
                if P[1] < N:
                    print("Found a factor, j=",j)
                    return P[1]
                else:
                    break
        if n==numberOfCurves:
            print("TEST FAILED: Reached upper limit on number of curves to try.
↪")
            return -1
```

```
        n = n+1
```

[4]:
```
########## Problem 2
#First I have to get my pollard algorithms from previous assignments

def PollardRhoFactorQuiet(N,upperBound = -1,f = lambda z:z^2+1,x=2,):
    y = x
    n = 0
    while True:
        x = f(x) % N
        y = f(f(y)) % N
        if (y>x):
            g = extendedEuclideanAlgorithm(y-x,N)[0]
        else:
            g = extendedEuclideanAlgorithm(x-y,N)[0]
        if g>1:
            print("Found a factor for n=",n)
            return g
        if n==upperBound:
            print("TEST FAILED: Reached upper bound without finding factors")
            return -1
        n += 1
```

[6]:
```
def PollardFactor(N, a=2, n=-1):
    i = 1
    while true:
        #print(i)
        p = extendedEuclideanAlgorithm(a-1,N)[0]
        if p == N and a!=2:
            print("TEST FAILED: Found GCD of N, try another value of a")
            return -1
        elif p !=1 and a!=2:
            q = N//p
            print("Found a factor, i=",i)
            return [p,q]
        elif i==n:
            print("TEST FAILED: Reached upper bound without finding factors")
            return -1
        a = fastPowerSmall(a,i,N)
        i = i+1
```

[7]:
```
#TESTING:

def runTests(N):
    print("Trying to factor",N)
    print("Lenstra:",LenstraFactor(N,15000,3))
    print("Pollard p-1:",PollardFactor(N,2,100000))
```

```
    print("Pollard rho:",PollardRhoFactorQuiet(N,1000000))
    print("")

runTests(25992521)
runTests(70711569293)
runTests(508643544315682693)
runTests(253770427990634017760356738З)
```

```
Trying to factor 25992521
Trying a new curve
Found a factor, j= 11
Lenstra: 2797
Found a factor, i= 102
Pollard p-1: [9293, 2797]
Found a factor for n= 30
Pollard rho: 2797

Trying to factor 70711569293
Trying a new curve
Found a factor, j= 233
Lenstra: 294167
Found a factor, i= 40064
Pollard p-1: [240379, 294167]
Found a factor for n= 735
Pollard rho: 294167

Trying to factor 508643544315682693
Trying a new curve
Found a factor, j= 443
Lenstra: 702291341
TEST FAILED: Reached upper bound without finding factors
Pollard p-1: -1
Found a factor for n= 13916
Pollard rho: 702291341

Trying to factor 2537704279906340177603567383
Trying a new curve
Trying a new curve
Trying a new curve
Trying a new curve
TEST FAILED: Reached upper limit on number of curves to try.
Lenstra: -1
Found a factor, i= 4524
Pollard p-1: [52725024492661, 48130926525403]
TEST FAILED: Reached upper bound without finding factors
Pollard rho: -1
```

[0]:

```

```