Final

December 19, 2021

[1]:
```python
##########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


### 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
```

[2]:

```
##########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]]
        #Otherweise we just continue as usual
        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 not, we're happy!
        if gcdPlus[0]!=1:
            return ["Factored",gcdPlus[0]]
        #Otherwise we just continue as usual.
        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]

#An O(1) storage variant
def doubleAndAddSmallAdjusted(P,n,E,p):
    Q = 'O'
    while n>0:
        if n%2 == 1:
            Q = addPointsAdjusted(E,Q,P,p)
            #before moving on, let's see if this addition factored N
            if Q[0] == "Factored":
                return Q
        n = n//2
        P = addPointsAdjusted(E,P,P,p)
        #before moving on, let's see if this addition factored N
        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 = doubleAndAddSmallAdjusted(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
```

[3]:
```
########## Problem 2
#First I have to get my other factoring algorithms from previous assignments

def quadraticSieve(a,b,B,N):
    B+=1 #The upper bound shouldn't be sharp
    sieveList = []
    primes = prime_range(3,B)
    primeDataList = [0 for i in range(0,len(primes)+1)]  #This i'th spot␣
 ↪primeDataList keeps track of how many factors of the i'th prime we have
    for t in range(a,b):
        sieveList.append([t*t - N] + primeDataList)
        #factor out powers of 2 right away
        while(sieveList[t-a][0]%2 == 0):
            sieveList[t-a][0] = sieveList[t-a][0]//2
            sieveList[t-a][1] += 1

    #now do the odd primes
    i = 2
    for p in primes:
        if fastPowerSmall(N,(p-1)//2,p) == 1: #First make sure N can even be a␣
 ↪square mod p
            pPower = p                        #We will in fact do this for␣
 ↪prime powers too
            while(pPower < 2*(b-a)):
```

```
                alpha = int(Mod(N,pPower).sqrt())     #First we compute the square roots (casting to an integer)
                beta = pPower-alpha

                #Next we find the smallest number >= a which is congruent to the square roots mod pPower
                if a%pPower < alpha:
                    t1 = a + alpha - (a%pPower)
                else:
                    t1 = a + alpha - (a%pPower) + pPower
                if a%pPower < beta:
                    t2 = a + beta - (a%pPower)
                else:
                    t2 = a + beta - (a%pPower) + pPower

                while(t1<b):
                    sieveList[t1-a][0] = sieveList[t1-a][0]//p  #We divide the associated numbers by p
                    sieveList[t1-a][i] += 1                     #Keeping track of how many factors to remove
                    t1 += pPower
                while(t2<b):
                    sieveList[t2-a][0] = sieveList[t2-a][0]//p
                    sieveList[t2-a][i] += 1
                    t2 += pPower
                pPower *= p
        i+=1
    return sieveList

def sieveFactor(a,b,B,N):
    sieve = quadraticSieve(a,b,B,N) #First run the sieve doing the relation building step.  Next we do the elimination step
    primes = prime_range(0,B)
    A = []
    C = []
    E = []

    for i in range(0,len(sieve)):         #These are the a_i, such that c_i = a_i^2-N is B-smooth, and the e_ij are the exponents of the prime factors p_j
        if sieve[i][0]==1:
            A.append(a + i)
            C.append((a+i)^2 - N)
            nextRow = [sieve[i][j] for j in range(1,len(sieve[i]))]
            E.append(nextRow)
```

```
    M = matrix(GF(2),E)                                       #Use Sage to convert E to␣
↪a matrix over F_2. Dont forget E
    basis = M.kernel().basis()                                #compute the basis of the␣
↪nullspace of this matrix

    for b in range(0,len(basis)):
        #Each entry here will the sum of the column of the E associated to a␣
↪prime, if it appears in the basis  This gives us the exponenets of the c_is
        exponent = [0 for i in range(0,len(primes))]
        a0 = 1
        for i in range(0,len(basis[b])):
            if basis[b][i] == 1:
                a0 = a0 * A[i] % N                            #We're also computing the␣
↪products of the a_i associated to the basis element of the nullspace
                for j in range(0,len(primes)):
                    exponent[j] += E[i][j]
        #Next we compute the product of the square roots off the c_i that␣
↪appear in our factorization using the exponenets we computed in the previous␣
↪loop
        b0 = 1
        for j in range(0,len(primes)):
            b0 = b0 * primes[j]**(exponent[j]//2) % N #since a^2 =␣
↪p_j^(exp[j]), we divide by 2 to take the square root in Z.
        #In this case there's no hope
        if a0==b0:
            continue
        divisor = extendedEuclideanAlgorithm(N,a0-b0)[0] #Here's our candidate!␣
↪ Let's see if it works!
        if(divisor != 1 and divisor !=N and divisor !=-1 and divisor !=-N):
            return[abs(divisor),abs(N//divisor)] #we use absolute values to␣
↪ensure positive factors
    print("none found")

def ell(x):
    return float(e^((ln(x)*ln(ln(x)))^.5))

def sieveLFactor(N):
    L = int(ell(N))
    B = int(L^(.5^.5))
    a = math.floor(sqrt(N))
    b = a + L
    return sieveFactor(a,b,B,N)
```

[4]:
```
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
```

[8]:

```
#TESTING:

def runTests(N):
    print("Trying to factor",N)
    print("Lenstra:",LenstraFactor(N,20000,5))
    print("Pollard p-1:",PollardFactor(N,2,100000))
    print("Quadratic Sieve:",sieveLFactor(N))
    print("")

runTests(25992521)
runTests(70711569293)
runTests(508643544315682693)
runTests(253770427990634017760356738 3)
```

```
Trying to factor 25992521
Trying a new curve
Found a factor, j= 151
Lenstra: 9293
Found a factor, i= 102
Pollard p-1: [9293, 2797]
Quadratic Sieve: [2797, 9293]

Trying to factor 70711569293
Trying a new curve
Found a factor, j= 379
Lenstra: 294167
Found a factor, i= 40064
Pollard p-1: [240379, 294167]
Quadratic Sieve: [294167, 240379]

Trying to factor 508643544315682693
Trying a new curve
```

```
Trying a new curve
Found a factor, j= 1367
Lenstra: 702291341
TEST FAILED: Reached upper bound without finding factors
Pollard p-1: -1
```

[6]:
```
########Quadratic Sieve failed and kille the kernel on 5086435443156826693. ␣
↪Let's try the last one:
def runTests(N):
    print("Trying to factor",N)
    print("Lenstra:",LenstraFactor(N,20000,5))
    print("Pollard p-1:",PollardFactor(N,2,100000))
    print("Quadratic Sieve:",sieveLFactor(N))
    print("")


runTests(2537704279906340177603567383)
```

```
Trying to factor 2537704279906340177603567383
Trying a new curve
Trying a new curve
Trying a new curve
Found a factor, j= 2143
Lenstra: 52725024492661
Found a factor, i= 4524
Pollard p-1: [52725024492661, 48130926525403]
```

[0]:
```
```