Homework10

December 5, 2020

[50]:
```
########## 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 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 also need probably prime and all of 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):
    #Return 2 and 3 as prime
    if p==2 or p==3:
```

```
        return True
    for i in range(0,20):
        a = ZZ.random_element(2,p-1)
        if millerRabin(a,p):
            return False
    return True
```

[12]:
```
########## Problem 1

def mixingFunction(x,a,b,g,h,p):
    #giving x = g^a*h^b runs the mixing function
    if x < p/3:
        x = (g*x) % p
        a = (a+1) % (p-1)
    elif x < 2*p/3:
        x = (x*x) % p
        a = (a+a) % (p-1)
        b = (b+b) % (p-1)
    else:
        x = (x*h) % p
        b = (b+1) % (p-1)
    return x,a,b




def PollardRhoLog(g,h,p):
    #Looking for a collision here
    x = 1
    y = 1

    #here are the exponents in particular x = g^a*h^b and y = g^c*h^d
    a = 0
    b = 0
    c = 0
    d = 0

    #loop through this until we find a collision
    i = 0
    while True:
        #run the mixing function once on x and twice on y
        x,a,b = mixingFunction(x,a,b,g,h,p)
        y,c,d = mixingFunction(y,c,d,g,h,p)
        y,c,d = mixingFunction(y,c,d,g,h,p)
        #if we find a collision we break
        if x==y:
            break
        i = i+1
```

```
    #then we know g^a*h^b = g^c*h^d so that log_g h^(d-b) = a-c
    #therefore our loslution solves the congrunece (d-b)x = a-c mod p-1
    candidates = solveLinearCongruence(d-b,a-c,p-1)

    #one of these is our log
    for l in candidates:
        if fastPowerSmall(g,l,p)==h:
            return l
```

[38]:
```
########## Problem 2
def pollardRhoLogTesting(g,h,p):
    log = PollardRhoLog(g,h,p)
    print("log base",g,"of",h,"mod",p,"is:",log)

##### Part(a)
g = 3
h = 5
p = 17
pollardRhoLogTesting(g,h,p)

##### Part(b)
g = 19
h = 24717
p = 48611
pollardRhoLogTesting(g,h,p)

##### Part(c)
g = 29
h = 5953042
p = 15239131
pollardRhoLogTesting(g,h,p)

##### Part(3)
g = 2
h = 2598854876
p = 2810986643
pollardRhoLogTesting(g,h,p)
```

```
log base 3 of 5 mod 17 is: 5
log base 19 of 24717 mod 48611 is: 37869
log base 29 of 5953042 mod 15239131 is: 2528453
log base 2 of 2598854876 mod 2810986643 is: 1470450926
```

[59]:
```
########## Problem 3
##### Part(a)

#I have pollardRhoFactor defaulting to the function z^2+1
```

```
def pollardRhoFactor(N,f = lambda z:z^2+1,x=2):
    y = x

    #let the counter be k
    k=1
    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 and g!=N:
            print("Data for N=",N)
            print("Nontrivial Factor:",g)
            print("Number of steps:",k)
            print("k/ Sqrt N",k/math.sqrt(N))
            print("k/ Sqrt(new factor)",k/math.sqrt(g))
            print()
            return g
        if g==N:
            print("Data for N=",N)
            print("The nontrivial factor found was",N,"itself. Try a different␣
 ↪mixing function or starting value.")
            print("Number of steps:",k)
            print("k/ Sqrt N",k/math.sqrt(N))
            print("k/ Sqrt(new factor)",k/math.sqrt(g))
            print()
            return g
        k = k+1


##### Part(b)

#REMARK: I am going to use lambda functions to define my functions within the␣
 ↪arguments when calling pollardRhoFactor.  This allows for a bit more␣
 ↪flexibility.  One could also just have various external functions defined␣
 ↪and call them separately.
print("Mixing Function: x^2+1:")
print()
pollardRhoFactor(2201)
pollardRhoFactor(9409613)
pollardRhoFactor(1782886219)

##### Part(c)
print("Mixing Function: x^2+2:")
print()
```

```
pollardRhoFactor(2201,lambda z:z^2+2)
pollardRhoFactor(9409613,lambda z:z^2+2)
pollardRhoFactor(1782886219,lambda z:z^2+2)

##### Part(d)
print("Mixing Function: x^2:")
print()
pollardRhoFactor(2201,lambda z:z^2)
pollardRhoFactor(9409613,lambda z:z^2)
pollardRhoFactor(1782886219,lambda z:z^2)

##### Part(e)
print("Mixing Function: x^2-2:")
print()
pollardRhoFactor(2201,lambda z:z^2-2)
pollardRhoFactor(9409613,lambda z:z^2-2)
pollardRhoFactor(1782886219,lambda z:z^2-2)


print("Mixing Function: x^2-2.  Starting value 3")
print()
pollardRhoFactor(2201,lambda z:z^2-2,3)
pollardRhoFactor(9409613,lambda z:z^2-2,3)
pollardRhoFactor(1782886219,lambda z:z^2-2,3)

##### Part(f)
print("Practicing on primes")
print()
pollardRhoFactor(17)
pollardRhoFactor(29)
pollardRhoFactor(5471)
```

```
Mixing Function: x^2+1:

Data for N= 2201
Nontrivial Factor: 31
Number of steps: 2
k/ Sqrt N 0.04263045563194875
k/ Sqrt(new factor) 0.3592106040535498

Data for N= 9409613
Nontrivial Factor: 541
Number of steps: 34
k/ Sqrt N 0.011083911156224306
k/ Sqrt(new factor) 1.4617741733339824

Data for N= 1782886219
```

```
Nontrivial Factor: 7933
Number of steps: 126
k/ Sqrt N 0.002984068107221476
k/ Sqrt(new factor) 1.414659166430652

Mixing Function: x^2+2:

Data for N= 2201
Nontrivial Factor: 71
Number of steps: 5
k/ Sqrt N 0.10657613907987187
k/ Sqrt(new factor) 0.5933908290969266

Data for N= 9409613
Nontrivial Factor: 541
Number of steps: 5
k/ Sqrt N 0.0016299869347388687
k/ Sqrt(new factor) 0.21496679019617387

Data for N= 1782886219
Nontrivial Factor: 7933
Number of steps: 68
k/ Sqrt N 0.0016104494546909552
k/ Sqrt(new factor) 0.7634668517244789

Mixing Function: x^2:

Data for N= 2201
Nontrivial Factor: 31
Number of steps: 4
k/ Sqrt N 0.0852609112638975
k/ Sqrt(new factor) 0.7184212081070996

Data for N= 9409613
Nontrivial Factor: 541
Number of steps: 36
k/ Sqrt N 0.011735905930119854
k/ Sqrt(new factor) 1.547760889412452

Data for N= 1782886219
Nontrivial Factor: 7933
Number of steps: 660
k/ Sqrt N 0.015630832942588685
k/ Sqrt(new factor) 7.410119443208178

Mixing Function: x^2-2:

Data for N= 2201
```

```
The nontrivial factor found was 2201 itself. Try a different mixing function or
starting value.
Number of steps: 1
k/ Sqrt N 0.021315227815974374
k/ Sqrt(new factor) 0.021315227815974374

Data for N= 9409613
The nontrivial factor found was 9409613 itself. Try a different mixing function
or starting value.
Number of steps: 1
k/ Sqrt N 0.00032599738694777375
k/ Sqrt(new factor) 0.00032599738694777375

Data for N= 1782886219
The nontrivial factor found was 1782886219 itself. Try a different mixing
function or starting value.
Number of steps: 1
k/ Sqrt N 2.368308021604346e-05
k/ Sqrt(new factor) 2.368308021604346e-05

Mixing Function: x^2-2.  Starting value 3

Data for N= 2201
Nontrivial Factor: 31
Number of steps: 4
k/ Sqrt N 0.0852609112638975
k/ Sqrt(new factor) 0.7184212081070996

Data for N= 9409613
Nontrivial Factor: 541
Number of steps: 12
k/ Sqrt N 0.003911968643373284
k/ Sqrt(new factor) 0.5159202964708173

Data for N= 1782886219
Nontrivial Factor: 224743
Number of steps: 1080
k/ Sqrt N 0.025577726633326938
k/ Sqrt(new factor) 2.278141358904614

Practicing on primes

Data for N= 17
The nontrivial factor found was 17 itself. Try a different mixing function or
starting value.
Number of steps: 6
k/ Sqrt N 1.4552137502179978
k/ Sqrt(new factor) 1.4552137502179978
```

```
Data for N= 29
The nontrivial factor found was 29 itself. Try a different mixing function or
starting value.
Number of steps: 8
k/ Sqrt N 1.4855627054164149
k/ Sqrt(new factor) 1.4855627054164149

Data for N= 5471
The nontrivial factor found was 5471 itself. Try a different mixing function or
starting value.
Number of steps: 60
k/ Sqrt N 0.811181230132437
k/ Sqrt(new factor) 0.811181230132437
```

[59]: 5471

[58]:
```
##### Part(g)

##### I'm going to first define a quite version of pollardRhoFactor which does
↪the same thing but doesn't print all the extra data.
def pollardRhoFactorQuiet(N,f = lambda z:z^2+1,x=2):
    y = x

    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:
            return g

def PollardRhoFactorize(N,f = lambda z:z^2+1):
    ###We save a list of known factors to pull from.
    #We pull them from the list one by one, checking if they are prime.
    #If they are we add them to the prime factor list.
    #Otherwise we use the Pollard algorithm to factor them and add the factors
↪to the factor list.

    factors = [N]
    primeFactors = []
    while(factors != []):
        N0 = factors.pop()
        if probablyPrime(N0):
```

```
            primeFactors.append(N0)
        else:
            x = 2
            n = 2
            while(True):
                #Sometimes pollardRhoFactor doesn't work (it returns N0).
                #In this case we try a new starting point (x)
                #Or if we've tried all possible starting points (mod N0)
                #We change the mixing function (x^2+n)
                p0 = pollardRhoFactorQuiet(N0,f,x)
                if p0==N0:
                    if(x>N0):
                        f = lambda z:z^2 + n
                        n = n+1
                        x = 2
                    else:
                        x = x+1
                else:
                    factors.append(p0)
                    factors.append(N0//p0)
                    break
    return(primeFactors)

#Let's run a few examples:
def factorizationLoud(N):
    print(N,"factors as",PollardRhoFactorize(N))

factorizationLoud(15)
factorizationLoud(17)
factorizationLoud(25)
factorizationLoud(2201)
factorizationLoud(9409613)
factorizationLoud(1782886219)
```

```
15 factors as [5, 3]
17 factors as [17]
25 factors as [5, 5]
2201 factors as [71, 31]
9409613 factors as [17393, 541]
1782886219 factors as [224743, 7933]
```

[0]:

```

```