Laboratory 3

Tiutiu Natan-Gabriel,937

Introduction

I will implement an algorithm for finding prime factors using Pollard's p-1 Method. This algorithm is very efficient in finding any prime factor p for a given number n(composite),for which p-1 has only small prime numbers. I also defined some functions for computing lcm for 2 numbers and also for a list of numbers. You can also find in my implementaion an algorithm for repeated squaring modular exponentiation (we will use rsme as a shortcut).

In order to generate the documentation written using markdown format , run **pandoc -t latex -o main.pdf main.nw** .

In order to generate the source code file, run **notangle main.nw>main.hs** .

Once you have the source code file,start the interpreter using command **ghci** ,then load the file using **:l main.hs** and finally run **getBasesWrapper YOURNUMBER** , for example **getBasesWrapper 1725** in order to find all bases b with respect to which the given YOURNUMBER is pseudoprime. Also you can run **getStrongPseudoprimeBasesWrapper YOURNUMBER** , for example **getStrongPseudoprimeBasesWrapper 65** in order to find all bases b with respect to which the given YOURNUMBER is strong pseudoprime.

Function generateBinary transforms a decimal number into a binary number. This is the mathematical model:

$$generateBinary(l,n) = \begin{cases} [0], if\ n = 0 and l = [] \\ l, if\ n = 0 \\ generateBinary([n\%2] \bigcup l, n/2), else \end{cases}$$

```
<<generateBinary>>=
generateBinary l n=
    if (n==0 && (length l)==0)
        then [0]
    else
        if (n==0)
            then l
```

```
        else (generateBinary ( [(mod n 2)] ++ l ) (quot n 2))
@
```

Function reverseList reverses a given list.The mathematical model is:

$$reverseList(x1, x2, ...xn) = \begin{cases} [], ifn = 0 \\ reverseList(x2, ...xn) \bigcup x1, else \end{cases}$$

```
<<reverseList>>=
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]
@
```

Let define Repeated Squaring Modular Exponentiation (rsme) function:

Input: b, k, n in N with b < n and k = $\sum_{i=0}^{t} k_i * 2^i$

Output: a = b^k mod n.

a=1

if k=0 then write(a)

c=b

if k_0 =1 then a=b

for i=1 to t do

  c= c^2 mod n

  if k_i =1 then a=c*a mod n

write(a)

Proof of corectness for Repeated Squaring Modular Exponentiation:

We have k = $\sum_{i=0}^{t} k_i * 2^i$

if k=0 => we return 1 because any integer to the power 0 is 1.

At each step we have b^{2^i} mod n,starting from i=1(from right in binary representation). Using b^{2^i} mod n, we compute $b^{2^{i+1}}$ mod n = $(b^{2^i})^2$ mod n = $(b^{2^i} mod\ n)^2$. If the k_i is 1 then we compute : (new a= new c* old a mod n).This holds because $b^{\sum_{i=0}^{t} k_i * 2^i} = b^{k_0 * 2^0} * b^{k_1 * 2^1} * \ldots * b^{k_t * 2^t}$ and then b^k mod n = $b^{\sum_{i=0}^{t} k_i * 2^i}$ mod n = ($b^{k_0 * 2^0}$ mod n) * ($b^{k_1 * 2^1}$ mod n) * ... * ($b^{k_t * 2^t}$ mod n) , where ($b^{k_i * 2^i}$ mod n) is our c computed at i-th iteration(for k_i=1).

```
<<rsme>>=
rsme b k n (ht:tt)= do
    if (n==0)
        then 0
    else do
```

```
        let a = 1
        if (k==0)
            then a
            else do
                let c=b
                let aa = if (ht==1)
                    then b
                    else a
                if (length(tt)==0)
                    then aa
                else (forLoopRsme aa c n k tt)
@
```

The forLoopRsme function represents the for loop of rsme function written in a functional style. More precisely,this loop:

for i=1 to t do

c= c^2 mod n

if k_i =1 then a=c*a mod n

```
<<forLoopRsme>>=
forLoopRsme :: Integer->Integer->Integer->Integer->[Integer]->Integer
forLoopRsme a c n k (ht:tt) = do
    let cc =(mod (c*c) n)

    let aa = if (ht==1)
            then
                (mod (cc*a) n)
            else
                a
    if ( (length tt)==0 )
        then aa
    else (forLoopRsme aa cc n k tt)
@
```

```
<<rsmeWrapper>>=
-- rsmeWrapper::Integer -> Integer -> Integer -> Integer
rsmeWrapper b k n=rsme b k n (reverseList (generateBinary [] k))
@
```

```
<<getMod>>=
getMod b k n=
    if (n==0)
        then 0
    else
        if (k==0) --this also trats the case 0^0
            then 1
```

```
        else (mod (b^k) n)
@

<<euclidean>>=
euclidean::Integer -> Integer -> Integer
euclidean a b =
    if (b>0)
        then (euclidean b (mod a b) )
        else a
@
```

The above relation only holds for two numbers, The idea here is to extend our relation for more than 2 numbers

```
<<computeLCMFor2Numbers>>=
computeLCMFor2Numbers a b= (quot (a*b) (euclidean a b) )
@

<<computeLCMForAList>>=
computeLCMForAList (x:xs) result=
    if ((length xs)==0)
        then (computeLCMFor2Numbers x result)
    else (computeLCMForAList xs (computeLCMFor2Numbers x result) )
@

<<computeLCMForAListWrapper>>=
computeLCMForAListWrapper l=
    if ((length l)==0)
        then 1
    else (computeLCMForAList l 1)
@
```

Pollard's p-1

Pollard's p-1 algorithm is efficiently in finding any prime factor p of an odd composite number for which p-1 has only small prime divisors. We will find then a multiple k of p-1 without knowing p-1, as a product of powers of small primes.

Theorem

By Fermat's Little Theorem,we have that if n is prime, then $\forall b \ in \ Z$ (enough b $<$ n) with (b, n) = 1 we have:

$$b^{n-1} = 1 \text{ (mod n) (1)}$$

Observation

The situation d = n, in which case the algorithm fails, occurs with a negligible probability.

In our implementation,as candidates for k,we will consider k = lcm{1, . . . , B}.

Proof of corectness

We want to find a divisor of n. Consider that k is a multiple of p-1(otherwise the algorithm won't work),that is k=(p-1)*q . *(As an observation,if k < bound,then k=(p-1)*q is always true)*

If p | a doesn't hold,then, $a^k = a^{(p-1)*q} = 1$ (mod p) ,because $a^{p-1} = 1$ (mod p),using Fermat's Little Theorem and of course $1^q = 1$ (mod p).

So,we obtained $a^k = 1$ (mod p) $\implies$ p | a^k - 1 .

Now, if p | n $\implies$ p | (a^k - 1,n) . Even more, if n | a^{p-1} doesn't hold, then d=(a^k - 1,n) is a non-trivial divisor of n.

In conclusion, we can find any prime factor p of an composite odd n for which p-1 has only small primes (or, for which p-1 divides k=lcm{1, . . . , B} ,more precisely)

Important obeservation(borderline case)

Please pay attention at picking a suitable bound B,that is, avoid bounds for which $a^k = 1$ (mod n) $\forall a \ in \ Z$.

Example: Take number n=37^2=1369 and the bound B=37. So,k=lcm{1, . . . , B}. Using Euler's function, we have that φ (37^2)=37^2 *(1-$\frac{1}{37}$)=36*37 So,using the Euler's Theorem,we have $a^{lcm\{1,...,37\}} = a^{36*37*z} = b^{36*37} = b^{\varphi(37^2)} = 1$ (mod 37^2) . We used that lcm{1,. . . ,37}=36*37*z,where z is an integer and we defined b = a^z.

So,for $\forall a \ in \ Z$ we will have that a^k=$a^{lcm\{1,...,37\}} = 1$ (mod 37^2) , that is, we will obtain an FAILURE for each iteration of our algorithm $\implies$ we will have an infinite loop !

Algorithms

```
<<pollardFunction>>=
-- pollardFunction :: Integer-> Integer->Integer->Integer
pollardFunction n b a = do
    let k=computeLCMForAListWrapper [x | x <- [1..b]]
    let aa=(rsmeWrapper a k n)
    let d= euclidean (aa-1) n
    if (d==1 || d==n)
        then  0
    else   d
@

<<pollard>>=
pollard n b=do
    r <- randomRIO(0,10)
    print  "input a valid value for the bound,or input NO if you want to use the default bou
```

```
    inputBound <- getLine
    let bound = if(inputBound=="NO" || inputBound=="no")
        then 17
        else (read inputBound :: Integer)
    print $ pollardFunction n bound r
@
```

```
<<rop_random>>=
rop_random :: Integer-> Integer -> Integer -> Bool
rop_random b k n = (rsmeWrapper b k n) > (getMod b k n)
@
```

– revapp :: Int -> Int -> Int -> Bool – revapp b k n = (rsmeWrapper b k n) == (getMod b k n)

– simpleMathTests :: TestTree – simpleMathTests = testGroup "Simple Math Tests" – [testCase "Small Numbers" . – revapp 3 4 5 @?= 1 –]

– elements :: [a] -> Gen a

– generate=

– prop_1=revapp 15 12 37 – prop_2=revapp 22 11 45 – prop_3=revapp 12 87 34 – prop_4=revapp 3 16 7 – prop_5=revapp 90 6 7 – prop_6=revapp 34 5 7 – prop_7=revapp 34 51 74 – prop_8=revapp 23 45 2 – prop_9=revapp 0 0 0 – prop_10=revapp 1 0 0 – prop_11=revapp 0 1 0 – – prop_12=revapp 0 0 1

test_lcm a b c =(computeLCMFor2Numbers a b)==c test_lcmForList l c =(computeLCMForAListWrapper l)==c

prop_101=test_lcm 8 12 (lcm 8 12) prop_102=test_lcm 26 165 (lcm 26 165) prop_103=test_lcm 135 205 (lcm 135 205)

prop_151=test_lcmForList [165,205,310] (lcm 165 (lcm 205 310)) prop_152=test_lcmForList [132,162,90] (lcm 132 (lcm 162 90)) prop_153=test_lcmForList [192,101,7] (lcm 192 (lcm 101 7)) prop_154=test_lcmForList [72,245,90,83] (lcm 72 (lcm 245 (lcm 90 83)))

return []

mainasdfg = $(quickCheckAll)

– main = quickCheck revapp

```
<<*>>=
{-# LANGUAGE TemplateHaskell #-}
import Test.QuickCheck
import Test.QuickCheck.All
import System.Random
import Data.Time
```

@