Laboratory 2

Tiutiu Natan-Gabriel,937

Introduction

I will implement an algorithm for determining all bases b with respect to which a composite odd number n is pseudoprime (1st part) and an algorithm for determining all bases b with respect to which a composite odd number n is strong pseudoprime(2nd part). In this implementation I will use repeated squaring modular exponentiation method (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.

Theoretical support

Theorem

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

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

Definition

An odd composite natural number n is called pseudoprime to the base b if (b, n) = 1 and (1) holds.

Observations

Because for $\forall b \ in \ Z$ with |b| $\geq$ 2 there are infinitely-many pseudoprimes to the base b,we will find bases b<n.

We also have that every odd natural number is pseudoprime to the bases b = $\pm$ 1 (3)

We will shrink our search space (for pseudoprimality algorithm) using the first part of (i) from the following theorem:

Theorem

Let n in N be an odd composite.

(i) n pseudoprime to b $\implies$ n pseudoprime to -b and b^{-1}, where b^{-1} is the inverse modulo n of b. (2)

(ii) n pseudoprime to b1 and b2 $\implies$ n pseudoprime to b1b2.

(iii) If n fails (1) for a single base b < n, then n fails (1) for at least half of the possible bases b < n.

Observations

So,using these improvements ((1) and (2)), our algorithm will compute the correct bases with respect to which the given number is pseudoprime only if the given number is an odd composite number!

We didn't use the 2nd part from (i) and (ii) because it would add more complexity to our program (in understanding not in efficiency) , and the benefits are few(if any) ,at least for numbers relatively small. By the way,for finding the inverse you can use Extended Euler's GCD algorithm having complexity O(Log n) - works when b and n are coprime.

Definition

Let n in N be odd composite and write n - 1 = 2^st for some odd t. Let b in Z with (b, n) = 1. If n and b satisfy the condition

$$b^t = 1 \text{ (mod n) or exists } 0 \leq \text{j} < \text{s} : b^{2^j t} = \text{-1 (mod n)}$$

then n is called strong pseudoprime to the base b.

Theorem

Strong pseudoprime to the base b $\implies$ pseudoprime to the base b.

Pseudoprimality

Our first function is a wrapper function having the following mathematical model:

$$getBasesWrapper(n) = [1] \bigcup [n-1] \bigcup getBases(n, 2...n/2+1)$$

```
<<getBasesWrapper>>=
getBasesWrapper n=[1]++[n-1]++(getBases n  ([x | x <- [2..((quot n 2)+1)]]) )
@
```

The getBases function which is called from the wrapper function, calls function rsme with corresponding parameters for each number in the given interval(2…n/2+1 in our case) and also check if that number is coprime with n. Using (1) and (2) from theoretical support we automatically add 1 and -1 to the result (see function getBasesWrapper) and for each base b satisfying the conditions we will also add -b to the result. In the mathematical model we will replace the call of rsme(l1,n-1,n, n in binary form) with $l1^{n-1}$ mod n.

$$getBases(n; l1, l2..lm) = \begin{cases} [l1]\bigcup[n-l1], if\ m = 1\ and\ (l1, n) = 1\ and\ l1^{n-1} = 1(mod\ n) \\ [l1]\bigcup[n-l1]\bigcup getBases(n, l2..lm), elif\ m > 1\ and\ (l1, n) = 1\ and\ l1^{n-1} = 1(mod \\ [], elif\ m = 1 \\ getBases(n, l2..lm), else \end{cases}$$

```
<<getBases>>=
getBases n (hl:tl)=
    if ((euclidean hl n)==1 && (rsme hl (n-1) n (reverseList (generateBinary [] (n-1))))==1)
        then if ((length tl)==0 ) then [hl] ++ [n-hl]
        else [hl] ++ [n-hl] ++ (getBases n tl)
    else
        if ((length tl)==0 ) then []
        else (getBases n tl)
@
```

As long as b,which is the second element is not equal to 0,we will call recursively euclidean(b,a%b) .We can observe that always second element becomes the first element in the new recursive call.If we would not inverse the position of the elements,we would get stuck(in some cases).

Ex.:a =1000,b=100 euclidean(1000,100)= euclidean(1000,100)= …= euclidean(1000,100)= ….

Here is the mathematical model:

$$euclidean(a, b) = \begin{cases} euclidean(b, a\%b), if b > 0 \\ a, else \end{cases}$$

The proof of corectness is based on the following lemma:

If a=c(mod b) (1), then (a,b)=(c,b) (3)

Proof of this lemma:

From (1) we have b|a-c,so there is a y such that by=a-c.If there is a d such that d divides a and b ,then it will also divide c=a-by. $\implies$ any divisor of a and b is a divisor of c and b. (2) Suppose (a,b)=x and (c,b)=y.Using (2) we have that x|y and y|x ,so we have that (a,b)=(c,b).

For our problem we use this:

We have a=a%b(mod b) $\implies$ using lemma (3) we have that (a,b)=(a%b,b)

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

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

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

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

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

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

```
<<generateBinary>>=
generateBinary l n=
    if (n==0)
        then l
    else (generateBinary ( [(mod n 2)] ++ l ) (quot n 2))
@
```

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
    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)
@
```

Strong pseudoprimality

Our first function is a wrapper function having the following mathematical model:

$$getStrongPseudoprimeBasesWrapper(n) = getStrongPseudoprimeBasesWrapper(n, 1...n-1)$$

```
<<getStrongPseudoprimeBasesWrapper>>=
getStrongPseudoprimeBasesWrapper n=(getStrongPseudoprimeBases n  ([x | x <- [1..(n-1)]]) )
@
```

The getStrongPseudoprimeBases function which is called from the wrapper function, calls euclidean and strongPseudoprime functions with corresponding parameters for each number in the given interval(1. . . n-1 in our case).

In the mathematical model we will replace the call of strongPseudoprime(hl, n, getS(n-1,0), (n-1)/$2^{getS(n-1,0)}$) with isPseudoprime(hl,n) for simplicity. Also we will use gSPB as a shortcut for getStrongPseudoprimeBases.

$$gSPB(n; l1, l2..lm) = \begin{cases} [l1], if\ m = 1\ and\ (l1, n) = 1\ and\ isPseudoprime(l1, n) \\ [l1] \bigcup gSPB(n, l2..lm), elif\ m > 1\ and\ (l1, n) = 1\ and\ isPseudoprime(l1, n) \\ [], elif\ m = 1 \\ gSPB(n, l2..lm), else \end{cases}$$

```
<<getStrongPseudoprimeBases>>=
getStrongPseudoprimeBases n (hl:tl)=
    if ((euclidean hl n)==1 && (strongPseudoprime hl n (getS (n-1) 0) (quot (n-1) (2^(getS (
        then if ((length tl)==0 ) then [hl]
        else [hl] ++ (getStrongPseudoprimeBases n tl)
    else
        if ((length tl)==0 ) then []
        else (getStrongPseudoprimeBases n tl)
@
```

Function strongPseudoprime returns 1 if the given number n is pseudoprime to the given bases b. s and t define the exponent n-1.

This is the mathematical model:

$$strongPseudoprime(b, n, s, t) = \begin{cases} 1, if\ b^t\ mod\ n = 1\ or\ exists\ 0 \leq j < s : b^{2^j t} = -1 (mod\ n) \\ 0, else \end{cases}$$

```
<<strongPseudoprime>>=
strongPseudoprime b n s t=

    if( (rsme b t n (reverseList (generateBinary [] t)))==1 || (existsIntermediateJ 0 s b n
        then 1
    else 0
@
```

Function existsIntermediateJ returns 1 if exists $0 \leq$ j $<$ s : $b^{2^j t}$ = -1 (mod n) and 0 else.

This is the mathematical model:

$$existsIntermediateJ(j, s, b, n, t) = \begin{cases} 0, if\ j = s \\ 1, else\ if\ b^{2^j t} = -1 (mod\ n) \\ existsIntermediateJ(j + 1, s, b, n, t), else \end{cases}$$

```
<<existsIntermediateJ>>=
existsIntermediateJ j s b n t=
    if (j==s)
        then 0
    else if ( (rsme b (2^j*t) n (reverseList (generateBinary [] (2^j*t))))==(n-1) )
        then 1
    else (existsIntermediateJ (j+1) s b n t)
@
```

Function getS returns s for a given n,where n= $2^s t$.

This is the mathematical model:

$$getS(n, s) = \begin{cases} s, if\ n\ mod\ 2 = 1 \\ getS(n/2, s + 1), else \end{cases}$$

```
<<getS>>=
-- getS :: Integer->Integer->Integer
getS n s=
    if ((mod n 2)==1)
        then s
```

```
    else (getS (quot n 2) (s+1))
@
```

Tests performed

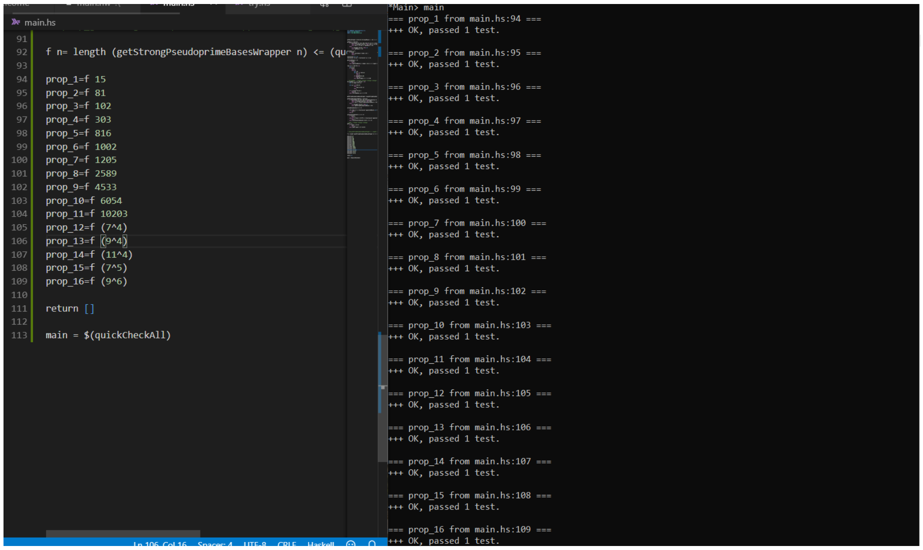

Figure 1: tests

```
<<tests>>=
f n= length (getStrongPseudoprimeBasesWrapper n) <= (quot (n-1) 4)

prop_1=f 15
prop_2=f 81
prop_3=f 102
prop_4=f 303
prop_5=f 816
prop_6=f 1002
prop_7=f 1205
prop_8=f 2589
prop_9=f 4533
prop_10=f 6054
prop_11=f 10203
prop_12=f (7^4)
prop_13=f (9^4)
prop_14=f (11^4)
prop_15=f (7^5)
prop_16=f (9^6)
```

```
return []

main = $(quickCheckAll)

@

<<*>>=
{-# LANGUAGE TemplateHaskell #-}
import Test.QuickCheck
import Test.QuickCheck.All

<<getBasesWrapper>>
<<getBases>>
<<euclidean>>
<<reverseList>>
<<generateBinary>>
<<rsme>>
<<forLoopRsme>>

<<getStrongPseudoprimeBasesWrapper>>
<<getStrongPseudoprimeBases>>
<<strongPseudoprime>>
<<existsIntermediateJ>>
<<getS>>

<<tests>>

@
```