

# 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. In this implementation I will use repeated squaring modular exponentiation method (we will use rsme as a shortcut).

## Theoretical support

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

$$b^{n-1} = 1 \pmod{n} \quad (1)$$

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

Because for  $\forall b \in \mathbb{Z}$  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 using the first part of (i) from the following theorem:

### Theorem

Let  $n$  in  $\mathbb{N}$  be an odd composite.

- (i)  $n$  pseudoprime to  $b \Rightarrow n$  pseudoprime to  $-b$  and  $b^{-1}$ , where  $b^{-1}$  is the inverse modulo  $n$  of  $b$ . (2)
- (ii)  $n$  pseudoprime to  $b_1$  and  $b_2 \Rightarrow n$  pseudoprime to  $b_1 b_2$ .
- (iii) If  $n$  fails (1) for a single base  $b < n$ , then  $n$  fails (1) for at least half of the possible bases  $b < n$ .

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.

## Implementation

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

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

```
<<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  $(2 \dots n/2 + 1)$  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  $l1^{n-1} \bmod n$ .

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

```
<<getBases>>=
getBases n (hl:t1)=
  if ((euclidean hl n)==1 && (rsme hl (n-1) n (reverseList (generateBinary [] (n-1))))==1)
    then if ((length t1)==0 ) then [hl] ++ [n-hl]
    else [hl] ++ [n-hl] ++ (getBases n t1)
  else
    if ((length t1)==0 ) then []
    else (getBases n t1)
@
```

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:

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

The proof of correctness is based on the following lemma:

*If  $a = c \pmod{b}$  (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  $c = a - by$ .  $\Rightarrow$  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  $a = a \% b \pmod{b} \Rightarrow$  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:

$$reverseList(x_1, x_2, \dots, x_n) = \begin{cases} [], & \text{if } n = 0 \\ reverseList(x_2, \dots, x_n) \cup x_1, & \text{else} \end{cases}$$

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

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

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

```
<<generateBinary>>=
generateBinary l n=
  if (n==0)
    then 1
  else (generateBinary ( [(mod n 2)] ++ 1 ) (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:  $a = b^k \text{ mod } n$ .

```
a=1
if k=0 then write(a)
c=b
if  $k_0 = 1$  then a=b
for i=1 to t do
   $c = c^2 \text{ mod } n$ 
  if  $k_i = 1$  then a=c*a mod n
write(a)
<<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
    (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
   $c = c^2 \text{ mod } n$ 
  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)
@

<<*>>=

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

@

```