

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.

Repeated Squaring Modular Exponentiation & helper functions

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

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

```
<<generateBinary>>=
generateBinary l n=
```

```

    if (n==0 && (length l)==0)
    then [0]
    else
    if (n==0)
    then 1
    else (generateBinary ( [(mod n 2)] ++ 1 ) (quot n 2))
@

```

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]
@

```

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: $a = b^k \bmod n$.

a=1

if k=0 then write(a)

c=b

if $k_0 = 1$ then a=c

for i=1 to t do

$c = c^2 \bmod n$

 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 $b^{2^i} \bmod n$, starting from i=1 (from right in binary representation). Using $b^{2^i} \bmod n$, we compute $b^{2^{i+1}} \bmod n = (b^{2^i})^2 \bmod n = (b^{2^i} \bmod n)^2$. If the k_i is 1 then we compute : (new a= new c* old a mod n). This holds because $b^{\sum_{i=0}^t k_i * 2^i} = b^{k_0 * 2^0} * b^{k_1 * 2^1} * \dots * b^{k_t * 2^t}$ and then $b^k \bmod n = b^{\sum_{i=0}^t k_i * 2^i} \bmod n = (b^{k_0 * 2^0} \bmod n) * (b^{k_1 * 2^1} \bmod n) * \dots * (b^{k_t * 2^t} \bmod n)$, where $(b^{k_i * 2^i} \bmod n)$ 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

$c = c^2 \bmod n$

if $k_i = 1$ then $a = c * a \bmod n$

```

<<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))
@

```

Euclidean algorithm

Description:

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

Lowest Common Multiple (lcm)

Observation

We have the following relation:

$$a * b = \text{lcm}(a, b) * \text{gcd}(a, b)$$

If we apply this relation to our program, we obtain the following relation:

$$\text{computeLCMFor2Numbers}(a, b) = \frac{a*b}{\text{euclidean}(a,b)}$$

The above relation only holds for two numbers. For computing lcm of a list we will define the function `computeLCMForAList`.

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

The function `computeLCMForAList` computes the lcm of a list. This is the mathematical model:

```
computeLCMForAList(l1,l2..lm;result)=
```

$$\begin{cases} \text{computeLCMFor2Numbers}(l1, \text{result}), & \text{if } m = 1 \\ \text{computeLCMForAList}(l2..lm; \text{computeLCMFor2Numbers}(l1, \text{result})), & \text{else} \end{cases}$$

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

The function `computeLCMForAListWrapper` is a wrapper function. Using it, instead calling **`computeLCMForAList l 1`**, you just need to call **`computeLCMForAList l`**. It also handle the situation in which the given list is empty (in this case it will return 1).

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

Pollard's p-1

Pollard's p-1 algorithm is efficient 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 $\forall b \in \mathbb{Z}$ (enough $b < n$) with $(b, n) = 1$ we have:

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

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 $k = \text{lcm}\{1, \dots, B\}$.

Proof of correctness

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 $k = (p-1) \cdot q$. (As an observation, if $k < \text{bound}$, then $k = (p-1) \cdot q$ is always true)

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

So, we obtained $a^k = 1 \pmod{p} \implies p \mid a^k - 1$.

Now, if $p \mid n \implies p \mid (a^k - 1, n)$. Even more, if $n \mid a^{p-1}$ 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 a composite odd n for which $p-1$ has only small primes (or, for which $p-1$ divides $k = \text{lcm}\{1, \dots, B\}$, more precisely)

Important observation (borderline case)

Please pay attention at picking a suitable bound B , that is, avoid bounds for which $a^k = 1 \pmod{n} \forall a \text{ in } Z$.

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

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

Algorithms

The function `pollardFunction` implements the algorithm for Pollard's $p-1$ Method. This is the algorithm:

Input: an odd composite number n , and a bound B

Output: a non-trivial factor d of n

1. Let $k = \text{lcm}\{1, \dots, B\}$ or $k = \prod \{q^i \mid q \text{ prime}, i \text{ is a non-zero natural number}, q^i \leq B\}$

2. Randomly choose $1 < a < n - 1$.

3. $a = a^k \pmod{n}$.

4. $d = (a-1, n)$

5. If $d=1$ or $d=n$ then output FAILURE (we output 0), else output d .

And this is the implementation:

```
<<pollardFunction>>=
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
@
```

Remark

If the algorithm ends with a failure, it is repeated for another value $1 < a < n-1$ or for another bound B . (we use another value a , because we should use the bound given by the user, if any).

That's what pollard does. At each iteration, we get the result of calling pollardFunction. As long as result is FAILURE (that is, 0), we recursively call pollard

.

Input: an odd composite number n , and a bound B

Output: a non-trivial factor d of n

Algorithm:

1. Generate a random number r , such that $2 \leq r \leq n-2$
2. Compute $a = \text{pollardFunction}(n, b, r)$
3. If $a=0$ then call $\text{pollard}(n, b)$, else return a .

```
<<pollard>>=
pollard n b=do
  r <- randomRIO(2,n-2)
  let a=pollardFunction n b r
  if(a==0)
    then (pollard n b)
    else (return a)
@
```

First of all, if you are not interested in using the default value for the bound (that is 17), you can SKIP the next function.

The function `pollard_wrapper` just let the user to choose whether he wants or not to use the default bound. After it gets the bound, `pollard_wrapper` returns the result of `pollard(n, bound)`.

```

<<pollard_wrapper>>=
pollard_wrapper n=do
  print "input a valid value for the bound,or input NO if you want to use the default bound"
  inputBound <- getLine

  let bound = if(inputBound=="NO" || inputBound=="no")
    then 17
    else (read inputBound :: Integer)
  return $ pollard n bound
@

```

The next function (pollardWithIterations) is very similar to pollard .The only difference is that this function allows you to set a maximum number of iterations. I will use this function in testing section,where I will test that a Mersenne number cannot be decomposed in a given number of iterations.

Input: an odd number n, a bound B, a maximum number of iterations to be performed

Output: a non-trivial factor d of n,or FAILURE(that is,we return 0)

Algorithm:

- 1.Generate a random number r,such that $2 \leq r \leq n - 2$
- 2.Compute $a = \text{pollardFunction}(n,b,r)$
- 3.If $a=0$ and $\text{iterations} > 0$,then call $\text{pollardWithIterations}(n,b,\text{iterations}-1)$,else return a.

```

<<pollardWithIterations>>=
pollardWithIterations n b iterations=do
  r <- randomRIO(2,n-2)
  let a=pollardFunction n b r
  if((a==0) && (iterations>0))
    then (pollardWithIterations n b (iterations-1))
    else (return a)
@

```

First of all,if you are not interested in using the default value for the bound (that is 17),you can SKIP the next function.

The function pollard_with_iterations_wrapper just let the user to choose whether he wants or not to use the default bound. After it gets the bound,pollard_with_iterations_wrapper returns the result of pollardWithIterations(n,bound,iterations).

```

<<pollard_with_iterations_wrapper>>=
pollard_with_iterations_wrapper n iterations=do
  print "input a valid value for the bound,or input NO if you want to use the default bound"
  inputBound <- getLine

```



```

let bound = if(inputBound=="NO" || inputBound=="no")
then 17
else (read inputBound :: Integer)
return $ pollardWithIterations n bound iterations
@

```

Testing

I will test functions `rsmeWrapper`, `computeLCMFor2Numbers`, `computeLCMForAListWrapper` and the variants of `pollard` (`pollard` and `pollardWithIterations`). For the first 3, I use the library *Test.QuickCheck.All*. Using this library, I test all properties in the current module, using Template Haskell. This is the reason why you see `{-# LANGUAGE TemplateHaskell #-}` pragma at the top of root chunk.

The name of the properties must begin with `prop_`.

Returns `True` if all tests succeeded, `False` otherwise.

You can also see that I added the following statements to my program: “`return []`” after properties and “`$quickCheckAll`” in main function.

If you are interested to find more about *Test.QuickCheck.All*, please visit <https://hackage.haskell.org/package/QuickCheck-2.13.2/docs/Test-QuickCheck-All.html>

For the last 2 I simply use prints. If the printed result is `True`, then everything was fine.

The function `rsmeTest` tests if `rsme` (repeated squaring modular exponentiation) function works as expected.

The functions `test_lcm` and `test_lcmForList` test if functions `computeLCMFor2Numbers` and `computeLCMForAListWrapper` work as expected. The last parameter for each function represents the expected result. We will compute the expected results using the pre-defined function `lcm`. (see below)

```

<<quickCheckTests>>=
-- rsmeTest :: Int -> Int -> Int -> Bool
rsmeTest b k n = (rsmeWrapper b k n) == (mod (b^k) n)

```

```

prop_1=rsmeTest 15 12 37
prop_2=rsmeTest 22 11 45
prop_3=rsmeTest 12 87 34
prop_4=rsmeTest 3 16 7
prop_5=rsmeTest 9034 623 11
prop_6=rsmeTest (5^7) (2^7) 13
prop_7=rsmeTest ((2^7)*(11^3)) (3^3) 74
prop_8=rsmeTest ((11^7)*(17^4)) (23^2) 153

```

```

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 264 165 (lcm 264 165)
prop_103=test_lcm 2451 4253 (lcm 2451 4253)
prop_104=test_lcm 75364 254634 (lcm 75364 254634)
prop_105=test_lcm 9645124 5462343 (lcm 9645124 5462343)

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 [342547,44253,1423] (lcm 342547 (lcm 44253 1423))
prop_154=test_lcmForList [41243499,775353,5464657,875624] (lcm 41243499 (lcm 775353 (lcm 5464657 875624)))
prop_155=test_lcmForList [((2^3)*(5^7)*(7^2)),((2^9)*(5^3)*(7^3)),((5^7)*(7^10)*(9^2)),((7^2)

return []
@

```

The next 2 functions `test_pollard` and `test_pollard_with_iterations` are almost identical to `pollard` and `pollardWithIterations` functions : the ONLY difference is that `test_pollard` and `test_pollard_with_iterations` print the result in the else a result in last branch and don't return a result as `pollard` and `pollardWithIterations` functions do.

I did this because in haskell is very hard to work with functions that return an IO value(so,with `pollard` and `pollardWithIterations`).The way I did it is a work-around: I print the result in the same function(`test_pollard` and `test_pollard_with_iterations`) I introduce a IO value (the randomly generated value `r`).

The function `test_pollard` returns True if it finished(that is,it found a non-trivial divisor). In order to finish,it will have to enter the "else" branch \implies `a` will be greater than 0,so (`a>0`) will be evaluated to True.

The function `test_pollard_with_iterations` returns True if after the number of iterations specified,`a` will be 0(i.e. it didn't find any non-trivial divisor). So,using this function I will test that a Mersenne prime cannot be decomposed in a given number of iterations.

```

<<pollardTests>>=
test_pollard n b=do
  r <- randomRIO(2,n-2)
  let a=pollardFunction n b r
  if(a==0)
    then (test_pollard n b)

```

```

        else (print (a>0))

test_pollard_with_iterations n b iterations=do
  r <- randomRIO(2,n-2)
  let a=pollardFunction n b r
  if((a==0) && (iterations>0))
    then (test_pollard_with_iterations n b (iterations-1))
    else (print (a==0))

main=do
  $quickCheckAll

  print "(test_pollard 15 5) evaluates as:"
  (test_pollard 15 5)
  print "(test_pollard (7*13) 8) evaluates as:"
  (test_pollard (7*13) 8)
  print "(test_pollard ((2^5)*(3^3)) 3) evaluates as:"
  (test_pollard ((3^5)*(5^3)) 3)
  print "(test_pollard ((2^6)*(3^7)*(5^5)) 5) evaluates as:"
  (test_pollard ((3^6)*(7^7)*(5^5)) 5)
  print "(test_pollard ((3^7)*(5^5)*(7^3)) 7) evaluates as:"
  (test_pollard ((3^7)*(5^5)*(7^3)) 7)
  print "(test_pollard ((11^3)*(13^5)) 13 ) evaluates as:"
  (test_pollard ((11^3)*(13^5)) 13 )
  print "(test_pollard ((2^4)*11*(17^5)) 17 ) evaluates as:"
  (test_pollard ((7^4)*11*(17^5)) 17 )

  -- --here I use numbers of the form 2^n-1 which are not Mersenne numbers
  -- --we will test that these nubmers CAN be decomposed

  print "(test_pollard (2^15-1) 13 ) evaluates as:"
  (test_pollard (2^15-1) 13 )
  print "(test_pollard (2^18-1) 17 ) evaluates as:"
  (test_pollard (2^18-1) 17 )
  print "(test_pollard (2^22-1) 17 ) evaluates as:"
  (test_pollard (2^22-1) 17 )
  print "(test_pollard (2^25-1) 17 ) evaluates as:"
  (test_pollard (2^25-1) 17 )
  print "(test_pollard (2^50-1) 17 ) evaluates as:"
  (test_pollard (2^50-1) 17 )
  print "(test_pollard (2^60-1) 17 ) evaluates as:"
  (test_pollard (2^60-1) 17 )
  print "(test_pollard (2^70-1) 17 ) evaluates as:"
  (test_pollard (2^70-1) 17 )
  print "(test_pollard (2^100-1) 17 ) evaluates as:"

```

```

(test_pollard (2100-1) 17 )
print "(test_pollard (21231-1) 17 ) evaluates as:"
(test_pollard (21231-1) 17 )
print "(test_pollard (24250-1) 17 ) evaluates as:"
(test_pollard (24250-1) 17 )
print "(test_pollard (285329-1) 17 ) evaluates as:"
(test_pollard (285329-1) 17 )
print "(test_pollard (2133562-1) 17 ) evaluates as:"
(test_pollard (2133562-1) 17 )
print "(test_pollard (2859301-1) 17 ) evaluates as:"
(test_pollard (2859301-1) 17 )
print "(test_pollard (21000025-1) 17 ) evaluates as:"
(test_pollard (21000025-1) 17 )

```

--here I use numbers of the form 2^n-1 which are not Mersenne numbers

--here we will test that these nubmers CAN NOT be decomposed in a number of given iterations

```

print "(test_pollard_with_iterations (27-1) 10 10) evaluates as:"
(test_pollard_with_iterations (27-1) 10 10)
print "(test_pollard_with_iterations (213-1) 10 10) evaluates as:"
(test_pollard_with_iterations (213-1) 10 10)
print "(test_pollard_with_iterations (217-1) 10 10) evaluates as:"
(test_pollard_with_iterations (217-1) 10 10)
print "(test_pollard_with_iterations (231-1) 10 10) evaluates as:"
(test_pollard_with_iterations (231-1) 10 10)
print "(test_pollard_with_iterations (261-1) 10 10) evaluates as:"
(test_pollard_with_iterations (261-1) 10 10)
print "(test_pollard_with_iterations (2127-1) 10 10) evaluates as:"
(test_pollard_with_iterations (2127-1) 10 20)
print "(test_pollard_with_iterations (2521-1) 10 10) evaluates as:"
(test_pollard_with_iterations (2521-1) 10 20)
print "(test_pollard_with_iterations (22203-1) 10 10) evaluates as:"
(test_pollard_with_iterations (22203-1) 10 20)
print "(test_pollard_with_iterations (221701-1) 10 10) evaluates as:"
(test_pollard_with_iterations (221701-1) 10 30)
print "(test_pollard_with_iterations (2110503-1) 10 10) evaluates as:"
(test_pollard_with_iterations (2110503-1) 10 40)
print "(test_pollard_with_iterations (2216091-1) 10 10) evaluates as:"
(test_pollard_with_iterations (2216091-1) 10 40)

```

@

```

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

```

```
import Test.QuickCheck.All
import System.Random
import Data.Bits
```

@