CSCI-762 Assignment 3

10/10

Tom Arnold <tca4384@rit.edu>

For this assignment we implemented algorithms to generate RSA regular and safe primes and compared the time performance for different bit sizes. We also implemented Pollard RHO and Brent's improved version to factor RSA moduli, and likewise compared time performance for different bit sizes.

RSA Safe Primes

The output from the RSA experiment is shown in the table below. The header contains the parameters of the run:

- 7 is the seed used for the PRNG. This was kept constant to simplify testing so that the same random numbers would be returned every time.
- 50 is the number of primes generated for each iteration. A larger number (than just 2) was used to increase the runtime so that a more accurate comparison could be drawn.
- 3,5,7 were the bases used for the Miller Rabin primality test for each prime.

Each row of the table includes information from one iteration. For each iteration the bit size of the primes was increased by 10. Both regular and RSA safe primes (where (p-1)/2 is also prime) were generated, and the times for each are recorded. Additionally to help with verification the first pair of primes for each bit size and algorithm are included in the table.

```
RSA-safe primes (seed=7 primes=50 bases=3,5,7)

b    regular    safe   regular                                            safe
bits milliseconds      first pair of primes
----/----------------/--------------------------------------------------------
10     0.000    15.625 859,761                                            587,1019
20     0.000    78.125 551689,882961                                      555287,536267
30    15.625   625.000 783552227,645126617                                1064193407,539541083
40    15.625   609.375 566096066743,864132157963                          570101195099,951237254459
50    15.625  1578.125 570617253596899,1091837116895101                   776756355444299,1042211200631519
60    46.875  3218.750 1095087101131266251,590979936164854933             1097885443855451003,1079771080704308423
70   296.875  10781.250  961073357312136300173,590715269883722362513 1017561728336942515163,591294071774551644263
```

Next is the code for the main driver program for the output shown above. This code uses a PRNG to generate random numbers in a range appropriate for the current bit size (e.g. 10 bit numbers are between 512 and 1023 inclusive). The code then filters those numbers through the Miller Rabin primality check, twice for the safe primes, and then finally prints out the times and first pairs of the results.

```haskell
{- |
Description : CSCI-762 Assignment 3 - RSA Safe primes main application.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables #-}
module Main where

import Control.Monad
import Data.List
import Text.Printf
import System.Random

import Util
import Number
import MillerRabin

main :: IO ()
main = do
  let -- Seed to use for PRNG.
      seed = 7

      -- Bases to use for MR primality test.
      bases = [3,5,7]

      -- Number of primes to search for at each iteration.
      numPrimes = 50

      -- Test if N is a regular prime.
      isPrime = \n -> mrPrimalityTests n bases == LikelyPrime

      -- Test if N is a safe prime ((n-1)/2 is prime as well).
      isSafePrime = \n -> isPrime n && isPrime ((n - 1) `div` 2)

  -- Initialze random number generator.
  let rng = mkStdGen seed

  -- Print out table header.
  putStrLn $ printf "RSA-safe primes (seed=%d primes=%d bases=%s)\n"
    seed numPrimes (toCSV bases)
  putStrLn "b    regular    safe   regular                                     safe"
  putStrLn "bits milliseconds      first pair of primes"
  putStrLn "----|----------------|-------------------------------------------------------"

  -- For each iteration print number of bits we generate
  -- primes for and how long it took in milliseconds.
  forM_ [10,20..70] $ \bits -> do

    -- Domain is the lower/upper bounds that give us
    -- B bit numbers. For example 8 bit numbers are
    -- between 128 and 255. This is used to get the RNG
    -- to give us numbers in this range.
    let domain = fixBitsDomain (bits :: Integer)

    -- Check regular primes.
    (regularTime, regularPrimes) <- do
      time $ randomRs domain rng
        |> filter isPrime
        |> take numPrimes

    -- Check safe primes.
    (safeTime, safePrimes) <- do
      time $ randomRs domain rng
        |> filter isSafePrime
        |> take numPrimes

    -- Print results for this iteration.
    putStrLn $ printf "%d %8.3f  %8.3f  %-43s %-43s"
      bits regularTime safeTime
      (regularPrimes |> take 2 |> toCSV)
      (safePrimes |> take 2 |> toCSV)
```

The code for the Miller Rabin test is shown below. This is split into several functions for composability:

- mrPrimalityTests - This simply takes a list of bases and calls the mrPrimalityTest function, i.e. it is the outer loop of the algorithm
- mrPrimalityTest - This performs the Miller Rabin primality test for a given base and return either composite if the number is composite, or likely prime if the number was not determined to be composite.
- findUR - This is used in the Miller Rabin test and finds numbers U and R such that `P-1 = 2^U * R`.

```haskell
{- |
Description : CSCI-762 Assignment 3 - Miller Rabin primality test.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables,
             DeriveGeneric, DeriveAnyClass #-}
module MillerRabin where

import Control.DeepSeq
import Control.Monad
import Data.Bits
import Data.Char
import Data.Function
import Data.Monoid
import GHC.Generics (Generic)

import Number
import Util

-- | Primality test result from Miller-Rabin which is
-- always either Composite or LikelyPrime.
data PrimalityResult = Composite | LikelyPrime
  deriving (Show, Eq, Generic, NFData)

-- | Miller-Rabin primality test using a given set of bases to test.
--
-- Examples:
--
-- >>> mrPrimalityTests 122591 [610,611]
-- Composite
mrPrimalityTests :: Integer -- ^ Candidate 'p' to test.
                 -> [Integer] -- ^ Chosen bases.
                 -> PrimalityResult -- ^ The result of the tests.

-- Base case, exhausted bases to test so this might be prime!
mrPrimalityTests p [] = LikelyPrime

-- Recursive case, test with base and then either terminate
-- if composite or recurse to next base otherwise.
mrPrimalityTests p (a:as) =
  let result = mrPrimalityTest p a
  in
    -- Base case, if result is composite then number is
    -- definitely composite so return.
    if result == Composite
    then result

    -- Recursive case, test with next base.
    else mrPrimalityTests p as

-- | Simplified Miller-Rabin primality test.
-- This does not contain the outer-loop to test a random
-- number of bases and instead a base is explicitly provided.
--
-- (This is mostly a straightforward translation from the pseudocode
-- in "Understanding Cryptography" except converted to use recursion,
-- and with the outer loop removed.)
--
-- Examples:
--
-- >>> mrPrimalityTest 1553 7
-- LikelyPrime
--
-- >>> mrPrimalityTest 122591 610
-- LikelyPrime
--
-- >>> mrPrimalityTest 122591 611
-- Composite
mrPrimalityTest :: Integer -- ^ Candidate 'p' to test.
                -> Integer -- ^ Chosen base 'a'.
                -> PrimalityResult -- ^ The result of the test.
mrPrimalityTest p a =
  if a < 1 || a > p - 1
  then error "Base must be >=1 and <= p - 1"
  else
    let !z = modExp a r p
    in
      if z /= 1 && z /= p - 1
      then
        let !z' = squareZ z 1
        in
          if z' == 1 || z' /= p - 1
          then Composite
          else LikelyPrime
      else
        LikelyPrime
  where
    (u, r) = findUR p

    -- | Inner loop of algorithm, this squares Z (modular
    -- so it wraps around) U-1 times or until Z becomes
    -- congruent with 1 or -1.
    squareZ z i =
      if i <= u - 1
      then
        let !z' = modExp z 2 p
        in
          if z' == 1 || z' == p - 1
          then z'
          else squareZ z' (i + 1)
      else
        z

-- | Find U and R such that P-1 = 2^U×R.
--
-- Examples:
--
-- >>> findUR 89
-- (3,11)
findUR :: Integer -> (Integer, Integer)
findUR p = loop 0 $ p - 1
  where
    loop u r = if odd r
               then (u, r)
               else loop (u + 1) $ r `div` 2
```

Pollard-RHO Factoring

The output from the Pollard RHO experiment is shown below. Like the RSA experiment the header includes information about the parameters of the experiment. Each row of the table includes an iteration where 100 RSA moduli are generated from RSA regular primes, and then Pollard RHO and Brent's improved version are used to factor each of the moduli. The table also includes timing information and an example of the results for verification. The results are in the format `(589,[19,31])` where the first number is the moduli and the values in the list are the two factors.

```
Pollard-RHO factoring (seed=7 moduli=100 bases=3,5,7)

b    k     pollard    brent  pollard                                             brent
bits       milliseconds      first factoring
---/---/-------------------/---------------------------------------------------------------
10    2      0.00      0.00 (589,[19,31])                                           (589,[19,31])
20    5      0.00      0.00 (653699,[859,761])                                      (653699,[859,761])
30   12      0.00      0.00 (546790259,[31907,17137])                               (546790259,[31907,17137])
40   29    234.38      0.00 (647065323257,[921563,702139])                          (647065323257,[921563,702139])
50   69    703.12    546.88 (994171832578871,[29761499,33404629])                   (994171832578871,
                                                                                        [29761499,33404629])

60  165   6937.50   3156.25 (652943655782632931,[721116553,905462027])              (652943655782632931,
                                                                                        [721116553,905462027])

70  394  51765.62  26468.75 (600294127374388058669,[27198919793,22070513533])       (600294127374388058669,
                                                                                        [27198919793,22070513533])

80  939 296171.88 156296.88 (820548576742555225988651,[1088185220611,754052307641]) (820548576742555225988651,

[1088185220611,754052307641])
```

The code generating the previous output is shown next. This has a similar structure to the RSA experiment except this time we're only interested in regular RSA primes, and we group them into pairs and multiply them together to get RSA moduli of a fixed bit size.

For the experiment we then factor these moduli using Pollard RHO and Brent's algorithm. The Pollard RHO pass is straightforward. Brent's algorithm requires a K parameter to be set to determine the number of non-GCD iterations to perform; determining this was tricky because it depends on the moduli being factored. Choosing a value too low or high can make the performance worse than the regular algorithm. After some experimentation it seems that the fifth root of the moduli gives us a good K value; see comments in the code for more details.

```
{- |
Description : CSCI-762 Assignment 3 - Pollard-RHO factoring main application.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables #-}
module Main where

import Control.Monad
import Data.Function
import Data.List
import Data.Maybe
import Text.Printf
import System.Random

import Util
import Number
import MillerRabin
import PollardRho

main :: IO ()
main = do
  let -- Seed to use for PRNG.
      seed = 7
```

```
    -- Function to use to increment pointer.
    f = \x -> x^2 + 1

    -- Bases to use for MR primality test.
    bases = [3,5,7]

    -- Number of RSA moduli to search for at each iteration.
    numModuli = 100

    -- Test if N is a regular prime.
    isPrime = \n -> mrPrimalityTests n bases == LikelyPrime

-- Initialze random number generator.
let rng = mkStdGen seed

-- Print out table header.
putStrLn $ printf "Pollard-RHO factoring (seed=%d moduli=%d bases=%s)\n"
  seed numModuli (toCSV bases)
putStrLn "b    k     pollard    brent  pollard                                            brent"
putStrLn "bits       milliseconds      first factoring"
putStrLn "---|---|-------------------|---------------------------------------------------------------"

forM_ [10,20..80] $ \bits -> do

  -- Domain is the lower/upper bounds that give us
  -- B bit numbers. For example 8 bit numbers are
  -- between 128 and 255. This is used to get the RNG
  -- to give us numbers in this range.
  -- Get B/2 bit numbers since we want B bit modulus.
  let domain = fixBitsDomain (bits `div` 2)

  -- Generate RSA moduli by finding regular primes
  -- grouping them into pairs and multiplying them.
  !(moduleTime, moduli) <- do
    time $ randomRs domain rng
        |> filter isPrime
        |> fix (\f xs -> case xs of
                   x:y:rest -> x * y : f rest
                   _ -> [])
        |> filter (\n -> numBits n == bits)
        |> take numModuli

  -- Check PR factors.
  (prTime, prFactors) <- do
    time $ moduli
      ||>> factors (prFactor f 1)

  -- Choose K value for this bit level.
  --
  -- This calculation is based on experimentation starting
  -- from some ideas in Brent's "An Improved Monte Carlo Factorization"
  -- paper, mainly that the K value should be significantly less than
  -- the fourth root of N. We use the fifth root, and to avoid
  -- sampling each moduli we simply choose a standard moduli for this
  -- bit level.
  --
  -- The goal is to minimize K and the number of GCD iterations. This
  -- can be tricky because a large K can return a small number of iterations
  -- than a smaller K, but the smaller K will likely be faster (but it is
  -- a balance).
  --
  -- This formula seems to work at least for this data set (note that the
  -- RNG seed is fixed).
  let k = (2^(bits - 1)) |> sqrt |> sqrt |> sqrt |> truncate :: Integer
```

```
    -- Check Brents factors.
    (brentTime, brentFactors) <- do
      time $ moduli
        ||>> \n -> factors (brentFactor k f 1) n

    putStrLn $ printf "%d %4d %9.2f %9.2f %-55s %-55s"
      bits k prTime brentTime
      (zip moduli (prFactors |> take 1 |>> fromJust) |> toCSV)
      (zip moduli (brentFactors |> take 1 |>> fromJust) |> toCSV)
```

Next is the actual implementations of Pollard RHO and Brent's algorithm, as well as a helper function to recursively find the factors of a number which is useful because the aforementioned algorithms only return one factor.

HS

```
{- |
Description : CSCI-762 Assignment 3 - Pollard-Rho factoring algorithm.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables,
             DeriveGeneric, DeriveAnyClass #-}
module PollardRho where

import Data.Bits
import Data.Char
import Data.Function
import Data.Monoid
import Debug.Trace
import Text.Printf

import Number
import Util

-- | Pollard-Rho factoring algorithm.
--
-- Computes significant factor of a given number.
--
-- From https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm
--
-- Examples:
--
-- >>> prFactor (\x -> x^2 + 1) 1 7171
-- Just (101,10)
--
-- >>> prFactor (\x -> x^2 + 1) 1 15770708441
-- Just (135979,212)
prFactor :: (Integer -> Integer) -- ^ Step function for X.
         -> Integer -- ^ Number to factor.
         -> Integer -- ^ Starting X.
         -> Maybe (Integer,Integer) -- ^ Factor and # iterations, or nothing if failed to find cycle.
prFactor f x n =
  let !(p, i) = loop 1 x x 1
  in
    if p == n
    then Nothing
    else Just (p, i)
  where
    -- Loop recursive case (p = 1).
    loop 1 x y i =
      let -- Apply function to step X forward once.
          !x' = f x `mod` n

          -- Apply function twice to step Y forward twice.
          !y' = f (f y `mod` n) `mod` n
          -- Get GCD of the difference between new X/Y and
          -- the modulus.
          !p = gcd (x' - y') n
        in
          -- Go to next iteration.
          loop p x' y' (i + 1)

      -- Loop base case, return p and iteration.
      loop p _ _ i = (p, i)

-- | Pollard-Rho factoring algorithm with Brent's accumulator.
--
-- Computes significant factor of a given number, using Brent's accumulator
-- to perform fewer GCD iterations.
--
-- From https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm
--
-- Examples:
--
-- >>> brentFactor 10 (\x -> x^2 + 1) 1 7171
-- Just (101,2)
--
-- >>> brentFactor 10 (\x -> x^2 + 1) 1 15770708441
-- Just (135979,23)
brentFactor :: Integer -- ^ Accumulator K parameter.
            -> (Integer -> Integer) -- ^ Step function for X.
            -> Integer -- ^ Number to factor.
            -> Integer -- ^ Starting X.
            -> Maybe (Integer,Integer) -- ^ Factor and # iterations, or nothing if failed to find cycle.
brentFactor k f x n =
  let !(p, i) = loop k 1 x x 1
  in
    if p == n
    then Nothing
    else Just (p, i)
  where
    -- Loop recursive case (p = 1).
    loop k 1 x y i =
      let -- Call accumulator to step X K-times,
          -- Y 2K-times, and return the product of
          -- their differences as Z.
          !(x', y', z) = accum k x y 1

          -- Get GCD between the product Z and the
          -- modulus.
          !p = gcd z n
      in
        -- Check for failure case from using the
        -- accumulator.
        if p == n && k /= 1
        then
          -- Failure case: redo this iteration
          -- with K hardcoded to 1, essentially
          -- falling back to regular PR factoring
          -- for the rest of the algorithm.
          loop 1 1 x y i
        else
          -- Go to next iteration.
          loop k p x' y' (i + 1)

    -- Loop base case, return p and iteration number.
    loop _ p _ _ i = (p, i)

    -- Brent's accumulator: Perform K-iterations
    -- without calculating the GCD (expensive), but instead
    -- getting the product of the differences so that
    -- we can do a single GCD (in loop function) on the product
    -- which saves a lot of work and "usually" works.

    -- Base case, ran out of K so return X/Y and product Z.
    accum 0 x y z = (x, y, z)

    -- Recursive case, step X/Y as normal, calculate product of their
    -- differences, and then go to next iteration.
    accum k' x y z =
      let !x' = f x `mod` n
          !y' = f (f y `mod` n) `mod` n
          !z' = z * abs (x' - y') `mod` n
      in
        accum (k' - 1) x' y' z'

-- | Given a factoring algorithm find all factors of a number.
--
-- Note: The strategy used here may not work well with Brent's
-- accumulator beyond 2 factors since you have to choose a
-- K value small enough for the smaller factors. For this
-- assignment this is not an issue since we are working with
-- RSA moduli.
--
-- Examples:
--
-- >>> factors (brentFactor 10 (\x -> x^2 + 1) 1) 7171
-- Just [71,101]
--
-- >>> factors (brentFactor 100 (\x -> x^2 + 1) 1) 7171
-- Just [71,101]
--
-- >>> factors (prFactor (\x -> x^2 + 1) 1) 7171
-- Just [71,101]
--
-- >>> factors (prFactor (\x -> x^2 + 1) 1) 7484267
-- Just [109,577,17,7]
factors f n =
  let factors = loop n []
  in
    -- If the only factor is the input then we
    -- failed. This might mean the input is prime.
    if factors == [n]
    then Nothing
    else Just factors
  where
    -- Get factor from N, then recurse to get factor
    -- from quotient.
    --
    -- E.g. 120 has factor of 5, 120/5
    -- gives us 24 which has factor of 8, 24/8 gives
    -- us 3 which is prime so we're done.
    loop n factors =
      case f n of

        -- Recursive case, add factor to our running list
        -- and factorize quotient.
        Just (x, _) ->
          loop (n `div` x) $ x : factors

        -- Base case, failed to factor N so add it to list
        -- of factors. If it's the only factor then it is
        -- either prime or our algorithm failed.
        _ -> n : factors
```

Solve exercise 6.27 page 253

> *Factor 262063, 9420457, 181937053 using Pollard RHO with f(x)=x^2+1. How many iterations are needed for each of the 3 integers?*

262063 factors into 503 and 521 in 36 iterations.

9420457 factors into 2351 and 4007 in 51 iterations.

181937053 factors into 12391 and 14683 in 166 iterations.

Support Code

Support code for the prior algorithms is shown below. This has been split into number theory utilities and general utilities. See the function comments and examples for more details. (The examples double as unit tests.)

```
{- |
Description : CSCI-762 Assignment 3 - General crypto number theory functions.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables,
             DeriveGeneric, DeriveAnyClass, MagicHash #-}
module Number where

import Data.Bits
import Data.Function
import Data.Maybe
import Data.Monoid
import Data.Ord
import GHC.Exts
import GHC.Generics (Generic)
import GHC.Integer.Logarithms

import Util

-- | Modular exponent, i.e. x^h mod n.
--
-- Implemented using square-and-multiply algorithm
-- from "Understanding Cryptography" page 182.
--
-- Examples:
--
-- >>> modExp 15 461 121
-- 26
--
-- >>> modExp 3 0 808
-- 1
modExp :: Integer -- ^ Base 'x' to exponentiate.
       -> Integer -- ^ Power 'h' to raise base to.
       -> Integer -- ^ Modulus 'n'.
       -> Integer -- ^ Resulting exponentiation.
modExp _ 0 _ = 1
modExp x h n = loop (t - 1) x
  where
    t = numBits h

    loop 0 r = r
```

```haskell
    loop i r =
      let r' = r^2 `mod` n
          i' = i - 1
      in
        loop i' $ if h .&. 2^i' /= 0
                  then r' * x `mod` n
                  else r'

-- | Get domain for numbers with n bits.
--
-- Examples:
--
-- >>> fixBitsDomain 8
-- (128,255)
--
-- >>> fixBitsDomain 70
-- (590295810358705651712,1180591620717411303423)
fixBitsDomain n = (2^(n-1), (2^n)-1)

-- | Get number of bits in number.
--
-- This uses GHC's low-level binding to GMP
-- for binary discrete log. In previous assignments
-- this was using floating-point log which worked
-- for small examples but returned incorrect results
-- for large examples due to lack of precision.
--
-- Examples:
--
-- >>> numBits 16
-- 5
--
-- >>> numBits 15
-- 4
--
-- >>> numBits 1180591620717411303423
-- 70
numBits :: Integer -> Int
numBits n = I# (integerLog2# n) |> (+1)


{- |
Description : CSCI-762 Assignment 3 - General utility functions.
Copyright : 2020 Tom Arnold <tca4384@rit.edu>
-}
{-# LANGUAGE StrictData, BangPatterns, ScopedTypeVariables #-}
module Util where

import Control.DeepSeq
import Control.Exception
import Control.Monad
import Control.Monad.IO.Class
import Control.Parallel.Strategies
import Data.List
import System.CPUTime
import System.Environment
import System.Exit
import System.IO.Unsafe
import System.IO

-- | Process computation left-to-right (like Unix pipe).
--
-- Examples:
--
```

```haskell
-- >>> [1,2,3] |> drop 1
-- [2,3]
(|>) :: a -> (a -> c) -> c
(|>) = flip ($)

-- | Like "|>" except applies to inside of elements. For example when
-- used with a list this will apply a function to each element.
--
-- Examples:
--
-- >>> [1,2,3] |>> (*2)
-- [2,4,6]
(|>>) :: Functor f => f a -> (a -> b) -> f b
(|>>) = flip (<$>)

-- | Like "|>>" except applied in parallel when compiled
-- in threaded mode and the appropriate RTS option are used (-Nx).
--
-- Examples:
--
-- >>> [1,2,3] ||>> (*2)
-- [2,4,6]
(||>>) xs f = (map f xs) `using` parList rdeepseq

-- | Convert list to CSV string.
--
-- Examples:
--
-- >>> toCSV [1,2,3]
-- "1,2,3"
toCSV xs = xs
  |>> show
  |> intercalate ","

-- | Run computation and return time in milliseconds and the result.
--
-- This is tricky because of lazy evaluation. We have to "force"
-- the expression to be evaluated completely because otherwise
-- the time will be 0ms.
--
-- Based on code from strict-io and timeit packages.
time :: (MonadIO m, NFData sa)
     => sa -- ^ Expression to be forced.
     -> m (Double, sa) -- ^ Time spent evaluating expression, and the expression.
time expr = do
  t1 <- liftIO getCPUTime
  let !value = force expr
  t2 <- liftIO getCPUTime
  let delta = fromIntegral (t2-t1) * 1e-9
  return (delta, value)
```