Software Testing Assignment 6

Cindy Berghuizen, Omar Pakker, Chiel Peter, Maria Gouseti

October 12, 2013

Exercise 2

```
-- EXERCISE 2
time :: IO t -> IO t
time a = do
    start <- getCPUTime
    v <- a
    end   <- getCPUTime
    let diff = (fromIntegral (end - start)) / (10^12)
    printf "Computation time: %0.3f sec\n" (diff :: Double)
    return v

timeEx x y m = do
    putStrLn "Starting..."
    putStrLn (show x)
    putStrLn (show y)
    putStrLn (show m)
    time $ exM1 x y m `seq` return ()
    time $ expM x y m `seq` return ()
    putStrLn "Done."

multT :: Integer -> IO ()
multT 0 = return ()
multT n = do
    x <- randomRIO (1000,1000000)
    y <- randomRIO (1000,1000000)
    m <- randomRIO (1000,1000000)
    a <- timeEx x y m
    b <- multT(n-1)
    return ()
```

By timing the two functions at calculating $rem(x, y)m$ the following results show up:

```
*Lab6> multT 5
Starting...
313212
682595
665260
Computation time: 0.000 sec
Computation time: 0.655 sec
Done.
Starting...
784179
115155
731957
Computation time: 0.000 sec
Computation time: 0.094 sec
Done.
Starting...
308391
274042
943299
Computation time: 0.000 sec
Computation time: 0.265 sec
Done.
Starting...
950314
628098
429619
Computation time: 0.000 sec
Computation time: 0.671 sec
Done.
Starting...
347081
990840
900524
Computation time: 0.000 sec
Computation time: 1.014 sec
Done.
*Lab6>
```

Where the first computation time is of exM1 and the second one of expM. exM1 is thus significanlty faster.

Exercise 4

```
-- EXERCISE 4
-- shows all the composite numbers primeF checks as prime
testF :: Int -> IO [Integer]
testF k =
    filterM (primeF k) (take 50 composites)
-- Run multiple tests
testFMore :: Integer -> Int -> IO [[Integer]]
testFMore 0 _ = return []
testFMore n k = do
    c <- testF k
    d <- testFMore (n-1) k
    return $ filter (not . null) (c:d)
```

k = 1, k =2 and k = 3 give 4 as a prime number. When the value of k gets higher there are less fool primes found, this is because more different random numbers are chosen for a which lowers the probability a composite number is considered a prime.

Exercise 5

```
-- EXERCISE 5
testCar :: Int -> IO [Integer]
testCar k =
    filterM (primeF k) (take 50 carmichael)
```

```
testMore1 :: Integer -> Int -> IO [[Integer]]
testMore1 0 _ = return []
testMore1 n k = do
    c <- testCar k
    d <- testMore1 (n-1) k
    return $ filter (not . null) (c:d)
```

Carmichael numbers almost always pass the Fermat's primality check. That was also shown in the testing, most of the numbers passed our test.
The Carmichael numbers are of the form $b^n \equiv b \pmod{n}$ for all integers $1 < b < n - 1$. This is also how Fermat's little theorem define prime numbers ($a^{p-1} \equiv 1 \pmod{p}$.) . Because Fermat defines prime numbers in the same way Carmichael defines the Carmichael numbers, the carmichael numbers do satisfy the definition of a prime number used in Fermat's primality check. Although the carmichael numbers are not prime numbers but do satisfy Fermat's definition of a prime number, they pass the testing.

Exercise 6

```
-- EXERCISE 6
testMR :: Int -> IO [Integer]
testMR k =
    filterM (primeMR k) (take 50 carmichael)

test2More :: Integer -> Int -> IO [[Integer]]
test2More 0 _ = return []
test2More n k = do
    c <- testMR k
    d <- test2More (n-1) k
    return $ filter (not . null) (c:d)
```

Although some Carmichael numbers still pass the Miller-Rabin primality, these are significantly less than with Fermat's primality check. If we higher k, meaning that we check with more random a's we even find that Miller-Rabin doesn't consider any Carmichael numbers as prime numbers.

```
-- functions to show the difference in result of Fermat's primality and Miller-Rabin's primality check
lengthCar :: Integer -> Int ->  IO String
lengthCar n k= do
    f <- testMore1 n k
    return $ show (length f)

lengthMR :: Integer -> Int ->  IO String
lengthMR n k = do
    f <- test2More n k
    return $ show (length f)
```

```
*Lab6> lengthCar 500 10
"500"

*Lab6>lengthMR 500 10
"0"
```

Exercise 7

```
-- EXERCISE 7
--take a large prime, use miller rabin to check if 2^p -1 ook prime is (dan is het een mersenne getal)
multipleMersenne :: Integer -> Int -> IO [(Bool, Integer, Integer)]
multipleMersenne 0 _ = return []
multipleMersenne n k = do
    m <- mersenne k
    c <- multipleMersenne (n-1) k
    return $ (m : c)

mersenne :: Int -> IO (Bool, Integer, Integer)
mersenne k = do
    p <- randomPrime
    m <- primeMR k ((2^p) - 1)
```

```
    return $ (m,p,((2^p) - 1)) --

randomPrime = do
    b <- (randomRIO (1,100))
    return (primes !! b)
```

The numbers that give True for the first argument in the tuple are indeed known Mersenne numbers as can be found on `http://en.wikipedia.org/wiki/Mersenne_prime`.

Exercise 8

The function $find2PrimesBitL$ takes as input the bitlength of the prime numbers and the number of the iterations the functions is allowed to perform until it finds two different primes. If the function succeeds returns a tupple with the two different primes else returns $(0, 0)$. Firstly, this function chooses a random number from 1 to $2^{bitlength} - 1$ to use it as offset inside the requested range $\{2^{bitlength}, ..., 2^{bitlength+1} - 1\}$. The function $findPrimeIn$ is responsible for finding a prime in the given range by testing numbers with primeMR and $k = 40$. The parameter k was chosen after searching the internet and experimenting with the functions from the previous exercises. If the prime is not found 0 is returned.

```
--EXERCISE 8
--First argument is the number of bits and the second is the number of tries it is going to has until it does finds tw
--Selects two random numbers and uses findPrimesIn to find the next primes.
find2PrimesBitL :: Int -> Int -> IO (Integer, Integer)
find2PrimesBitL bits 0 = return (0,0)
find2PrimesBitL bits tries = let range = 2^bits in do
  o1 <- randomRIO (1,range-1)
  p1 <- findPrimesIn (range + o1) (2*range-1)
  o2 <- randomRIO (1,range-1)
  p2 <- findPrimesIn (range + o2) (2*range-1)
  if ((p1 == p2) || (p1 == 0) || (p2 == 0)) then find2PrimesBitL bits (tries -1)
  else  return (p1,p2)

-- k = 40 error rate negligible according to suggestion in stackoverflow //TODO change that with experimental values f
-- Finds the next prime in the range {start,...,finish}, if it does not find a prime it returns zero.
findPrimesIn :: Integer -> Integer -> IO Integer
findPrimesIn start finish
  | (start >= finish) =  return 0
  |otherwise = do
     isPrimeV <- primeMR 40 start
     if isPrimeV then return start
```

The function rsa which gets as input a message, after generating 2 prime numbers with bitlength 512 passes them to rsa_public which returns the public key and $rsa\_private$ which returns the private key. Then, the public key is given to rsa_encode along with the message which returns the encoded message. In order to decode the message the private key along with the cipher text are provided to rsa_decode and the plain text message is returned. The function rsa returns a tupple with the decoded message and its encoded version. If the 2 different primes needed were not generated, the function throws an error.

```
-- Gets as an input a number and returns a tupple with the cipher text and the plain text
rsa:: Integer -> IO (Integer, Integer)
rsa message = do
  (p,q) <- find2PrimesBitL 512 10
  if (p/=0 && p /= 0) then
   let (e,n) = rsa_public p q
       (d,_) = rsa_private p q
       cipher = rsa_encode (e,n) message
       plain = rsa_decode (d,n) cipher in
   return (plain, cipher)
  else error "Primes were not generated"
```