

Homework 4

Brian Alliet

October 7, 2004

1 Questions

Question 1

The largest error probability of Miller-Rabin for integers between 100,000 and 101,000 is about **.0145356**. 100171 fails the Miller-Rabin on for 1,456 of the integers between 2 and 100,169. The exact error probability is 1456/100168.

The largest error probability of Miller-Rabin for integers between 1,000,000 and 1,010,000 is about **.0712114**. 1003213 fails the Miller-Rabin on 71,440 of the integers between 2 and 1,003,211.

The code used to compute these (and a description of it) is below.

Question 2

256,961 can be factored into 877 and 293. This answer was obtained with the `quadraticSieveFactor` function shown below, It was run as follows:

```
quadraticSieveFactor [2,3,5,7,11,13,17,19,23,29,31] 256961
Just (877,293)
```

1.1 Miller-Rabin Error Probability Testing Code

To reduce the amount of computation required to find the largest error probability for the Miller-Rabin test we do the computation in separate rounds, reducing the number of integers we need to test after each round. Each round has `tries` and `cutoff` parameters. The the Miller-Rabin test is run `tries` times on each integer and only integers that fail at least `cutoff` times are sent on to the next round.

```

runRound :: Int -> [Int] -> IO([(Int,Int)])
runRound tries = mapM $ \x -> do
    errors <- liftM (length.filter id) $ mapM (\_ -> do
        r <- randomRIO(2,fromIntegral x - 2)
        return $! millerRabinPrimality (fromIntegral x) r
    ) [1..tries]
    return $! (x,errors)

```

runRound runs a single round. The tries and the set of integers to try are given as arguments. The Miller-Rabin test is then run `tries` times (using random integers) on each integer. We record the number of times each integer fails the test (and is incorrectly classified as prime).

```

runRounds :: Int -> Int -> [(Int,Int)] -> IO([Int])
runRounds mn mx = foldM (\prev (tries,cutoff) -> do
    putStrLn $ "Running " ++ show tries
                ++ "/" ++ show cutoff ++ " round..."
    res <- filter ((>= cutoff).snd) `liftM` runRound tries prev
    let sres = sortBy (\(_,a) (_,b) -> compare b a) res
    putStrLn "Results:"
    mapM_ (\(n,errs) ->
        putStrLn $ show n ++ ": Miller-Rabin errors: "
                    ++ show errs ++ " ("
                    ++ show (fromIntegral errs/fromIntegral tries :: Double)
                    ++ ")"
    ) sres
    putStrLn ""
    return $ map fst sres
    )
    $ dropWhile (<mn)
    $ filter odd
    $ [fst x|x <- fastPrimeSieve mx, not $ snd x]

```

runRounds runs all the rounds specified on the command line and displays useful status information to stdout after each round. First the round is run on the output from the previous round (or all the odd composite numbers within the range if this is the first round). Next the results are filtered based on the `cutoff` for this round. They are then sorted in descending order by the number of failures. The current results are printed to stderr and then the filtered results are returned for use in the next round.

```

main :: IO()
main = do

```

main is the main function.

```

    args <- getArgs >>= mapM readIntegral
    when (length args < 4) $ do
        me <- getProgName
        hPutStrLn stderr $ "Usage: " ++ me ++ " min max tests cutoff ..."
        exitFailure
    let (mn:mx:rounds) = args

```

First it reads all the command line arguments and ensures they are all integers. The first two are used as the minimum and maximum and the remaining arguments are treated as the number of tries and the cutoff for each round.

```
final ← runRounds mn mx $ map listToTuple $ chunkify 2 rounds
```

Next `runRounds` is used to run all the rounds specified on the command line.

```
when (null final) (do  
  putStrLn "All results filtered out!"  
  exitFailure)
```

If there are no results from the final round then we can't continue.

```
let  
  x = fromIntegral $ head final  
  errors = length  
    $ filter id  
    $ map (millerRabinPrimality x) [2..x-2]
```

Next we take the first integer returned from the last round (the one that failed the most), run it on every possible random input to the Miller-Rabin function, and record the number of times it failed.

```
putStrLn $ "We had the most errors on: " ++ (show x)  
putStrLn $ "There is a " ++ show (errors) ++ "/" ++ show (x-3)  
  ++ " chance that Miller Rabin will fail on " ++ (show x)
```

Finally, we print the results.

2 Math Functions

This sections contains a small number theory library. Most of the interesting stuff is contained in here. However, there is also a lot of stuff irrelevant to this assigning. `millerRabinPrimality` and `quadraticPrimeSieve` are the most important functions.

2.1 Number Theory Functions

```
mulMod :: Integral a => a -> a -> a -> a
mulMod a b c = (b * c) `mod` a

squareMod :: Integral a => a -> a -> a
squareMod a b = (b * b) `rem` a

addMod :: Integral a => a -> a -> a -> a
addMod a b c = (b + c) `mod` a

negateMod :: Integral a => a -> a -> a
negateMod a b = negate b `mod` a

eqMod :: Integral a => a -> a -> a -> Bool
eqMod a b c = b `mod` a == c `mod` a

productMod :: Integral a => a -> [a] -> a
productMod a = foldl (mulMod a) 1
```

`mulMod`, `addMod`, and `negateMod` simply implement addition, multiplication, and negation modulo a .

```
pow' :: (Num a, Integral b) => (a -> a -> a) -> (a -> a) -> a -> b -> a
pow' _ _ _ 0 = 1
pow' mul sq x' n' = f x' n' 1
  where
    f x n y
      | n == 1 = x `mul` y
      | r == 0 = f x2 q y
      | otherwise = f x2 q (x `mul` y)
    where
      (q,r) = quotRem n 2
      x2 = sq x
```

`pow'` implements the standard exponentiation by squaring algorithm.

```
pow :: (Num a, Integral b) => a -> b -> a
pow = pow' (*) (\x->x*x)

powMod :: Integral a => a -> a -> a -> a
powMod m = pow' (mulMod m) (squareMod m)
```

`pow` uses `pow'` to implement normal exponentiation. `powMod` implements exponentiation (mod m).

```

extEuclid :: Integral a => a -> a -> (a,a,a)
extEuclid a 0 = (1,0,a)
extEuclid a b = (y,x-y*q,thegcd)
  where (x,y,thegcd) = extEuclid b r
        (q,r) = quotRem a b

```

This is the Extended Euclid's Algorithm implemented as a recursive function. The correctness of this algorithm can be proven using induction.

FIXME Do the proof

```

multInvMod :: Integral a => a -> a -> a
multInvMod b n =
  if thegcd /= 1
  then error $ (show n) ++ " is not relatively prime"
  else x `mod` b -- ensure it is within the base
  where (x,_,thegcd) = extEuclid (n `mod` b) b

```

The Extended Euclid's Algorithm is used to find the multiplicative inverse of a number (mod b). `extEuclid` returns the gcd and x and y in $ax + by = gcd$. If the gcd is 1 (indicating that the given number is relatively prime and has a multiplicative inverse) then x is treated as the multiplicative inverse. x must be the multiplicative inverse because when we substitute in for b and gcd in the above equation we get $ax + b(base) = 1$. $b(base)$ equals $0 \pmod{b}$ so $ax = 1$.

```

crt :: Integral a => [(a,a)] -> a
crt congruencies = x `mod` mprod
  where
    (a,m) = unzip congruencies
    mprod = product m
    z = map (div mprod) m
    y = zipWith multInvMod m z
    x = sum $ zipWith3 (\b c d -> b*c*d) a y z

```

`crt` implements the Chinese Remainder Theorem

```

lazyPrimeSieve :: [Integer] -> [Integer]
lazyPrimeSieve [] = []
lazyPrimeSieve (x:xs) = x : (lazyPrimeSieve $ filter (\y -> y `rem` x /= 0) xs)

lazyPrimes :: [Integer]
lazyPrimes = 2 : lazyPrimeSieve [3,5..]

```

`lazyPrimes` is a list of all the prime numbers. It is generated using the Sieve of Eratosthenes (so it isn't very fast).

```

#ifdef __GLASGOW_HASKELL__
fastPrimeSieve :: Int → [(Int,Bool)]
fastPrimeSieve limit = assocs $ runST run
  where
    sqrLimit = intSqrt limit
    run :: ST s (UArray Int Bool)
    run = do
      a ← newArray (0,limit) True :: ST s (STUArray s Int Bool)
      mapM_ (\x → writeArray a x False) [0,1]
      mapM_ (\x → do
        p ← readArray a x
        when(p) $ mapM_ (\y → writeArray a y False) [x*2,x*3..limit]
        ) [2..sqrLimit+1]
      unsafeFreeze a >>= return
#endif

```

fastPrimeSieve is a fast Sieve of Eratosthenes. It generates lists of primes in large blocks rather than one at a time. It works well even for large values lazyPrimes is painfully slow for larger values). This uses an array containing a slot for every number from 0 to limit. For every prime it finds it marks off multiples of that prime in the array (so it eliminates a whole bunch of numbers in one shot).

```

factor :: Integral a ⇒ a → [(a,a)]
factor = factorWithBase (map fromIntegral lazyPrimes)

```

```

factorWithBase :: Integral a ⇒ [a] → a → [(a,a)]
factorWithBase base x'
  | x' ≤ 0 = error "factorWithBase: can only factor numbers > 0"
  | otherwise = f x' 0 base
  where
    f 1 0 _ = []
    f x _ [] = [(x,1)]
    f x e ps@(p:ps')
      | r ≠ 0 = (p,e) : f x 0 ps'
      | otherwise = f q (e+1) ps
      where (q,r) = quotRem x p

```

factor factors a number and returns a list of prime factors and exponents. factorWithBase uses a factor base other than the list of all primes. (However, everything in the factor base should still be prime). If x cannot be factored within the factor base the last "factor" will be whatever is left over to the power of 1.

```

factor 450 = [(2,1),(3,2),(5,2)]

```

```

factorWithinBase :: Integral a ⇒ [a] → a → Maybe [(a,a)]
factorWithinBase fb n
  | null fs = Just fs
  | (fst $ last fs) `elem` fb = Just fs
  | otherwise = Nothing
  where fs = factorWithBase fb n

```

factorWithinBase is just like factorWithBase except Nothing is returned if x could not be factored with only the factor base.

```
quadraticSieveFactor :: Integral a => [a] -> a -> Maybe (a,a)
quadraticSieveFactor fb n = listToMaybe
  $ map(\(x,y) -> let g = gcd (x-y) n in (g,n`div`g))
  $ filter (\(x,y) -> x /= y && not (eqMod n x (-y)))
  $ map (\(x,fs) ->
    (x,productMod n $ zipWith (^) fb [e`div`2|e<-fs])
    )
  $ filter ((all even).snd)
  $ map (\row ->
    let (ns,fs) = unzip row
        in (productMod n ns, map sum $ transpose fs)
    )
  $ concat $ map (flip listPerms rows) [2..fblen`div` 2]
```

quadraticSieveFactor factors a large integer using a quadratic sieve. A factor base and the integer are given as arguments and if factorization is possible two factors are returned. Factorization is performed using the following steps:

1. Find possible rows - See below
2. Find all permutations of the rows - listPerms is used to find all permutations (starting with the smallest ones) of the rows.
3. Multiply each set of rows - productMod and transpose are used to multiply all the rows in each group together. This results in an integer and a list of exponents where the integer squared is equivalent to the exponents when applied to the factor base.
4. Find dependencies mod 2 - Next we filter out all rows who don't have linear dependencies mod 2
5. Take the square root of the exponents - Now we take the square root of the exponents (by dividing each one by two) and apply them back to the factor base to get a single integer.
6. Filter out unhelpful results - If $x \equiv \pm y \pmod n$ then it won't do us much good in factoring n so these results are discarded.
7. Find factors - Finally the remaining results are run through gcd to find a non-trivial of n and the two factors are returned.

```

where
  fblen = length fb
  maxRows = fblen + fblen `div` 2
  searchRange = [1..fromIntegral $ fblen*4]

  rows = take maxRows
        $ mapMaybe(\x → do
            let sq = squareMod n x
                when (sq == 0) Nothing
                fs ← factorWithinBase fb sq
                return (x, map snd fs)
            )
        $ [intSqrt (i*n) + j | i ← searchRange, j ← searchRange]

```

To find every possible row for the matrix above we search through numbers in the form $\sqrt{ni} + j$ looking for numbers whose square mod n is able to be factored using the factor base. A few limits are set (in terms of the size of the factor base) to limit the number of results to some sane size.

```

legendre :: Integral a => a -> a -> a
legendre a p
  | p < 3 = error "p isn't an odd prime"
  | x == p-1 = -1
  | otherwise = x
  where x = powMod p a ((p-1) `div` 2)

```

legendre returns the value of the legendre symbol $(\frac{n}{p})$ as described in *A Course in Computational Algebraic Number Theory* by Henri Cohen.

```

sqrtModPrime :: Integral a => a -> a -> Maybe a
sqrtModPrime p y'
  | p == 2 = Just y
  | y == 0 = Just 0
  | otherwise = case p `mod` 4 of
    1 -> case p `mod` 8 of
      1 -> shanksTonelli p y
      5 ->
          if d == 1 then Just $ powMod p y (p `div` 8 + 1)
          else if d == p - 1 then Just $ mulMod p
              (mulMod p y 2) (powMod p (mulMod p y 4) (p `div` 8))
          else Nothing
          where d = powMod p y (p `div` 4)
    _ -> error "sqrtModPrime: p isn't prime"
  3 ->
      if mulMod p x x == y then Just x else Nothing
      where x = powMod p y ((p+1) `div` 4)
    _ -> error "sqrtModPrime: p isn't prime"
  where y = y' `mod` p

```

sqrtModPrime find the square root of $y \pmod{p}$ where p is a prime number using the methods described in

FIXME: Talk about this some more... proofs?

```
shanksTonelli' :: Integer → Integer → Maybe Integer
shanksTonelli' p a =
  iter
    z          — y ← z
    e          — r ← e
    (mulMod p a (powMod p x' 2)) — b ← ax2
    (mulMod p a x')          — x ← ax
  where
    (z,_) = findZ $ mkStdGen 42
    (e,q) = find2km (p-1)
    x' = powMod p a (q`div`2)

iter :: Integer → Integer → Integer → Integer → Maybe Integer
iter _ _ 1 x = Just x — done
iter y r b x =
  case find (λm → powMod p b (2m) == 1) [0..r-1] of
    Nothing → Nothing — isn't a square
    Just m → iter
      y'
      m          — r ← m
      (mulMod p b y') — b ← by
      (mulMod p x t) — x ← xt
      where
        t = powMod p y (2(r-m-1)) — t ← y(2(r-m-1))
        y' = powMod p t 2          — y ← t2 -}
findZ g = if legendre n p == -1 then (powMod p n q,g') else findZ g'
  where (n,g') = randomR (2,p-1) g
```

shanksTonelli implements the Shanks-Tonelli algorithm as described in *A Course in Computational Algebraic Number Theory* by Henri Cohen. This is a direct conversion of the algorithm description in the book to Haskell. (Which is why it maintains its imperative-like style).

```
sqrtMod :: Integral a ⇒ a → a → [a]
sqrtMod m x
  | m ≤ 0 = error "Brianweb.Math.sqrtMod: m must be ≥ 1"
  | otherwise =
    case filter ((≠0).snd) $ factor m of
      [] → [0] — only happens if m == 1
      [(p,1)] →
        case sqrtModPrime p x of
          Just 0 → [0]
          Just r → [r,negateMod p r]
          Nothing → []
      [(_,_) ] → error "FIXME: This is broken"
    ps → nub
      $ map crt
      $ positionPerms
      $ map (λp → map (λr → (r,p)) $ sqrtMod p x)
      $ map (uncurry (^)) ps
```

`sqrtMod` finds the square root of $x \pmod{m}$ as described in *Cryptography with Coding Theory* by Trappe and Washington.

```
euler :: Integral a => a -> a
euler x = numerator
    $ (fromIntegral x)
    * (product $ map ((1-).(1%)) $ map fst $ factor x)
```

`euler` is Eulers ϕ -function as described in *Cryptography with Coding Theory* by Trappe and Washington.

```
millerRabinPrimality :: Integer -> Integer -> Bool
millerRabinPrimality n a
  | a <= 1 || a >= n-1 =
    error $ "millerRabinPrimality: a out of range ("
      ++ show a ++ " for " ++ show n ++ ")"
  | n < 2 = False
  | even n = False
  | b0 == 1 || b0 == n' = True
  | otherwise = iter (tail b)
  where
    n' = n-1
    (k,m) = find2km n'
    b0 = powMod n a m
    b = take (fromIntegral k) $ iterate (squareMod n) b0
    iter [] = False
    iter (x:xs)
      | x == 1 = False
      | x == n' = True
      | otherwise = iter xs
```

`millerRabinPrimality` implements the Miller-Rabin Primality Test as described in *Cryptography with Coding Theory* by Trappe and Washington. b_0 is b_0 and b represents the whole array of b s. Each b is generated by squaring the last b . `iter` runs a single iteration. If b is 1 then the number is classified as composite, if b is $n-1$ the number is classified as prime, otherwise, the next iteration is run. If we run out of b s then the number is classified as composite.

```

probablyPrime :: (RandomGen a) => Integer -> a -> (Bool,a)
probablyPrime p
  | p <= smallPrime = \r -> (smallCheck p,r)
  | otherwise = loop 10
  where
    loop :: RandomGen a => Int -> a -> (Bool,a)
    loop 0 rg = (True,rg)
    loop n rg =
      if millerRabinPrimality p r
      then loop (n-1) rg'
      else (False,rg')
      where (r,rg') = randomR (2,p-2) rg
#ifdef __GLASGOW_HASKELL__
    smallPrime = 100000
    smallCheck x = binarySearch (fromIntegral x) a
    where
      l = map fst $ filter snd
        $ fastPrimeSieve (fromIntegral smallPrime)
      a = array (0,length l-1) $ zip [0..] l
#else
    smallPrime = 2000
    smallCheck x = x `sortedElem` lazyPrimes
#endif

```

probablyPrime returns true if p it is very likely that p is prime. For small integers the Sieve of Eratosthenes is used to test primality. The Miller-Rabin test is done 10 times for larger integers.

```

getPrime :: (RandomGen a) => Int -> a -> (Integer,a)
getPrime n rg0 = if p then (r,rg2) else getPrime n rg2
  where
    (r,rg1) = randomR (2^(n-1),2^n-1) rg0
    (p,rg2) = probablyPrime r rg1

```

getPrime gets a random n -bit prime number using the Miller-Rabin test.

```

find2km :: Integral a => a -> (a,a)
find2km n = f 0 n
  where
    f k m
      | r == 1 = (k,m)
      | otherwise = f (k+1) q
      where (q,r) = quotRem m 2

```

find2kn finds two numbers (k and m) such that $2^k m = n$ and m is odd.