Homework 5

Jared Roesch (4826574)

Spring 2015

Problem 1

According to the Hasse Theorem

The order of an elliptic curve group over GF(p) is bounded by:

$p + 1 - 2\sqrt{p} \leq order(\varepsilon) \leq p + 1 + 2\sqrt{p}$

So for GF(29):

$29 + 1 - 2\sqrt{29} \leq order(\varepsilon) \leq 29 + 1 + 2\sqrt{29}$ $18 \leq order(\varepsilon) \leq 42$

Problem 2

The elements of the group are (see Appendix for code used):

$(0,1)(0,28)(2,1)(2,28)(3,1)(3,28)(6,1)(6,28)(7,1)(7,28)(8,1)(8,28)(13,1)$
$(13,28)(14,1)(14,28)(15,1)(15,28)(17,1)(17,28)(19,1)(19,28)(21,1)(21,28)(22,1)$
$(22,28)(23,1)(23,28)(28,1)(28,28)$

Problem 3

The exact order is 31, there are 30 points and one zero element.

Problem 4

We will pick our point P to be $(14, 3)$.

Problem 5

The answer is $(7, 6)$. I computed this via `binaryMethod (Point 14 1) 15 29 (-3)` using the implementation contained in the Appendix.

Problem 6

The answer is also $(7, 6)$. I computed this via `cannonicalRecoding (Point 14 3) 15 29 (-3)` using the implementation contained in the Appendix.

Appendix

```
module Main where

import Control.Monad (forM)
import Data.Maybe
import Data.List
import Data.Char
import Debug.Trace
import Numeric
import Data.Bits

hwEquation :: Integer -> Integer -> Integer
hwEquation x n = ((x ^ 3) + (-3 * x) + 4) `mod` n

testEquation :: Integer -> Integer -> Integer
testEquation x n = ((x ^ 3) + x + 1) `mod` n

-- bug around here
isQuadraticResidue :: Integer -> Integer -> Bool
isQuadraticResidue q p = (q `mod` p ) `elem` [(x ^ 2 `mod` p) | x <- [0..(p - 1)]]

modExp :: Integer -> Integer -> Integer -> Integer
modExp b 0 m = 1
modExp b e m = t * modExp ((b * b) `mod` m) (shiftR e 1) m `mod` m
    where t = if testBit e 0 then b `mod` m else 1

solveForY :: Integer -> Integer -> Integer
solveForY y p = snd $ head $ filter (\(r, _) -> r == y) [(modExp x 2 p, x) | x <- [0..(p -

data Point = Zero
           | Point Integer Integer
```

```haskell
           deriving (Eq, Ord, Show)

-- Extended Euclidean algorithm.  Given non-negative a and b, return x, y and g
-- such that ax + by = g, where g = gcd(a,b).  Note that x or y may be negative.
gcdExt a 0 = (1, 0, a)
gcdExt a b = let (q, r) = a `quotRem` b
                 (s, t, g) = gcdExt b r
             in (t, s - q * t, g)

-- Given a and m, return Just x such that ax = 1 mod m.  If there is no such x
-- return Nothing.
modInv :: Integer -> Integer -> Maybe Integer
-- modInv 0 _ = Just 0
modInv a m = let (i, _, g) = gcdExt a m
             in if g == 1 then Just (mkPos i) else Nothing
  where mkPos x = if x < 0 then x + m else x

pointAdd :: Integer -> Integer -> Point -> Point -> Point
pointAdd n a (Point x1 y1) Zero = Point x1 y1
pointAdd n a Zero (Point x2 y2) = Point x2 y2
pointAdd n a (Point x1 y1) (Point x2 y2) =
    if x2 == x1 && y2 == -y1
        then Zero
        else
            let m = (if x1 /= x2
                        then (y2 - y1) * (fromJust $ ((x2 - x1) `mod` n) `modInv` n)
                        else (3 * x1^2 + a) * (fromJust $ (2 * y1) `modInv` n)) `mod` n
                x3 = (m^2 - x1 - x2) `mod` n
                y3 = (m * (x1 - x3) - y1) `mod` n
            in Point x3 y3

pointOrder :: Integer -> Integer -> Point -> Integer
pointOrder n a p = go p 1
    where go Zero i = i
          go p' i =
              let p'' = pointAdd n a p p'
                  negp'' = pointAdd n a p (neg n p')
              in if negp'' == Zero
                     then i + 1
                     else go p'' (i + 1)

neg :: Integer -> Point -> Point
neg n Zero = Zero
neg n (Point x y) = (Point x (y - n))

findPrimitiveElements :: Integer -> Integer -> [Point] -> Maybe [Point]
```

```haskell
findPrimitiveElements n a [] = Nothing
findPrimitiveElements n a ps =
    let order = fromIntegral $ length ps + 1
        in Just $ filter (\p -> pointOrder n a p == order) ps

enumeratePoints :: (Integer -> Integer -> Integer) -> Integer -> Integer -> [Point]
enumeratePoints ec p a =
    let pairs = (flip map) [0..(p - 1)] $ \x ->
            let y2 = ec x p
                y = solveForY y2 p
                yDual = (-y) `mod` p
            in if not $ isQuadraticResidue y2 p
                then Nothing
                else if y /= 0
                    then Just $ [Point x y, Point x yDual]
                    else Just $ [Point x y]
    in sort $ concat $ map fromJust $ filter isJust $ pairs

binaryMethod :: Point -> Integer -> Integer -> Integer -> Point
binaryMethod p k n a =
    let c = case toBits k of
                ('1':_) -> p
                _       -> Zero
        body res ei =
            let res' = pointAdd n a res res
                in if ei == '1'
                    then pointAdd n a p res'
                    else res'
    in foldl body c (drop 1 $ toBits k)

cannonicalRecoding :: Point -> Integer -> Integer -> Integer -> Point
cannonicalRecoding p @ (Point x y) k n a =
    let pInverse = Point x (-y)
        pK = binaryMethod p (k + 1) n a
    in pointAdd n a pInverse pK

toBits :: Integer -> String
toBits n = showIntAtBase 2 intToDigit n ""

runECCInfo :: (Integer -> Integer -> Integer) -> Integer -> Integer -> IO ()
runECCInfo ec p a = do
    pairs <- forM [0..(p-1)] $ \x -> do
        putStrLn $ take 10 $ repeat '-'
        putStrLn $ "X:" ++ show x
        let y2 = ec x p
        putStrLn $ "QR: " ++ (show $ isQuadraticResidue y2 p)
```

```haskell
        let y = solveForY y2 p
        putStrLn $ "Y:" ++ (show y)
        let yDual = (-y) `mod` p
        putStrLn $ "Second Y:" ++ show yDual
        return $ if not $ isQuadraticResidue y2 p
            then Nothing
            else if y /= 0
                then Just $ [(x, y), (x, yDual)]
                else Just $ [(x, y)]
    print $ sort $ concat $ map fromJust $ filter isJust $ pairs

formatPoints :: [Point] -> String
formatPoints ps = intercalate " " $ map (show . toTuple) ps
    where toTuple (Point x y) = (x, y)

-- This only dumps some information, I used ghci as an interpreter to compute
-- my results.
main :: IO ()
main = do
    runECCInfo testEquation 23 1
    runECCInfo hwEquation 29 3
```