

\$SPAD/src/input ecfact.as

The Axiom Team

July 31, 2014

Abstract

Contents

1 License

3

1 License

```
-- Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd.
-- All rights reserved.
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are
-- met:
--
--   - Redistributions of source code must retain the above copyright
--     notice, this list of conditions and the following disclaimer.
--
--   - Redistributions in binary form must reproduce the above copyright
--     notice, this list of conditions and the following disclaimer in
--     the documentation and/or other materials provided with the
--     distribution.
--
--   - Neither the name of The Numerical ALgorithms Group Ltd. nor the
--     names of its contributors may be used to endorse or promote products
--     derived from this software without specific prior written permission.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
-- IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
-- TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
-- PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
-- OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
-- EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
-- PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
-- PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
-- LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
-- NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
-- SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

— * —

```
#include "axiom.as"
#pile
```

```
--% Elliptic curve method for integer factorization
-- This file implements Lenstra's algorithm for integer factorization.
-- A divisor of N is found by computing a large multiple of a rational
-- point on a randomly generated elliptic curve in  $P^2 \mathbb{Z}/N\mathbb{Z}$ .
-- The Hessian model is used for the curve (1) to simplify the selection
-- of the initial point on the random curve and (2) to minimize the
-- cost of adding points.
-- Ref: IBM RC 11262, DV Chudnovsky & GV Chudnovsky
-- SMW Sept 86.
```

```

--% EllipticCurveRationalPoints
--)abbrev domain ECPTS EllipticCurveRationalPoints

EllipticCurveRationalPoints(x0:Integer, y0:Integer, z0:Integer, n:Integer): ECcat == ECdef where
  Point ==> Record(x: Integer, y: Integer, z: Integer)

  ECcat ==> AbelianGroup with
    double: % -> %
    p0:      %
    HessianCoordinates: % -> Point

  ECdef ==> add
    Rep == Point
    import from Rep
    import from List Integer

    Ex == OutputForm

    default u, v: %

    apply(u:%,x:'x'):Integer == rep(u).x
    apply(u:%,y:'y'):Integer == rep(u).y
    apply(u:%,z:'z'):Integer == rep(u).z
    import from 'x'
    import from 'y'
    import from 'z'

    coerce(u:%): Ex == [u.x, u.y, u.z]$List(Integer) :: Ex
    p0:% == per [x0 rem n, y0 rem n, z0 rem n]
    HessianCoordinates(u:%):Point == rep u

    0:% ==
      per [1, (-1) rem n, 0]
    -(u:%):% ==
      per [u.y, u.x, u.z]
    (u:%) = (v:%):Boolean ==
      XuZv := u.x * v.z
      XvZu := v.x * u.z
      YuZv := u.y * v.z
      YvZu := v.y * u.z
      (XuZv-XvZu) rem n = 0 and (YuZv-YvZu) rem n = 0
    (u:%) + (v:%): % ==
      XuZv := u.x * v.z
      XvZu := v.x * u.z
      YuZv := u.y * v.z
      YvZu := v.y * u.z
      (XuZv-XvZu) rem n = 0 and (YuZv-YvZu) rem n = 0 => double u
      XuYv := u.x * v.y
      XvYu := v.x * u.y

```

```

Xw := XuZv*XuYv - XvZu*XvYu
Yw := YuZv*XvYu - YvZu*XuYv
Zw := XvZu*YvZu - XuZv*YuZv
per [Yw rem n, Xw rem n, Zw rem n]
double(u: %): % ==
import from PositiveInteger
X3 := u.x**(3@PositiveInteger)
Y3 := u.y**(3@PositiveInteger)
Z3 := u.z**(3@PositiveInteger)
Xw := u.x*(Y3 - Z3)
Yw := u.y*(Z3 - X3)
Zw := u.z*(X3 - Y3)
per [Yw rem n, Xw rem n, Zw rem n]
(n:Integer)*(u: %): % ==
n < 0 => (-n)*(-u)
v := 0
import from UniversalSegment Integer
for i in 0..length n - 1 repeat
    if bit?(n,i) then v := u + v
    u := double u
v

--% EllipticCurveFactorization
--)abbrev package ECFAC EllipticCurveFactorization

EllipticCurveFactorization: with
    LenstraEllipticMethod: (Integer) -> Integer
    LenstraEllipticMethod: (Integer, Float) -> Integer
    LenstraEllipticMethod: (Integer, Integer, Integer) -> Integer
    LenstraEllipticMethod: (Integer, Integer) -> Integer

    lcmLimit: Integer -> Integer
    lcmLimit: Float-> Integer

    solveBound: Float -> Float
    bfloor: Float -> Integer
    primesTo: Integer -> List Integer
    lcmTo: Integer -> Integer
== add
import from List Integer
Ex == OutputForm
import from Ex
import from String
import from Float

NNI==> NonNegativeInteger
import from OutputPackage
import from Integer, NonNegativeInteger
import from UniversalSegment Integer

```

```

blather:Boolean := true

--% Finding the multiplier
flabs (f: Float): Float == abs f
flsqrt(f: Float): Float == sqrt f
nthroot(f:Float,n:Integer):Float == exp(log f/n::Float)

bfloor(f: Float): Integer == wholePart floor f

lcmLimit(n: Integer):Integer ==
  lcmLimit nthroot(n::Float, 3)
lcmLimit(divisorBound: Float):Integer ==
  y := solveBound divisorBound
  lcmLim := bfloor exp(log divisorBound/y)
  if blather then
    output("The divisor bound is", divisorBound::Ex)
    output("The lcm Limit is", lcmLim::Ex)
  lcmLim

-- Solve the bound equation using a Newton iteration.
--
-- f = y**2 - log(B)/log(y+1)
--
-- f/f' = fdf =
--      2              2
--      y (y + 1)log(y + 1) - (y + 1)log(y + 1) logB
--      -----
--                               2
--      2y(y + 1)log(y + 1) + logB
--
fdf(y: Float, logB: Float): Float ==
  logy := log(y + 1)
  ylogy := (y + 1)*logy
  ylogy2:= y*logy*ylogy
  (y*ylogy2 - logB*ylogy)/((2@Integer)*ylogy2 + logB)
solveBound(divisorBound:Float):Float ==
  -- solve          y**2 = log(B)/log(y + 1)
  -- although it may be y**2 = log(B)/(log(y)+1)
  relerr := (10::Float)**(-5)
  logB := log divisorBound
  y0 := flsqrt log10 divisorBound
  y1 := y0 - fdf(y0, logB)
  while flabs((y1 - y0)/y0) > relerr repeat
    y0 := y1
    y1 := y0 - fdf(y0, logB)
  y1

-- maxpin(p, n, logn) is max d s.t. p**d <= n
maxpin(p:Integer,n:Integer,logn:Float): NonNegativeInteger ==

```

```

d: Integer := bfloor(logn/log(p::Float))
if d < 0 then d := 0
d::NonNegativeInteger

multiple?(i: Integer, plist: List Integer): Boolean ==
  for p in plist repeat if i rem p = 0 then return true
  false

primesTo(n:Integer):List Integer ==
  n < 2 => []
  n = 2 => [2]
  plist := [3, 2]
  i:Integer := 5
  while i <= n repeat
    if not multiple?(i, plist) then plist := cons(i, plist)
    i := i + 2
    if not multiple?(i, plist) then plist := cons(i, plist)
    i := i + 4
  plist
lcmTo(n:Integer):Integer ==
  plist := primesTo n
  m: Integer := 1
  logn := log(n::Float)
  for p in plist repeat m := m * p**maxpin(p,n,logn)
  if blather then
    output("The lcm of 1..", n::Ex)
    output("          is", m::Ex)
  m
LenstraEllipticMethod(n: Integer):Integer ==
  LenstraEllipticMethod(n, flsqrt(n::Float))
LenstraEllipticMethod(n: Integer, divisorBound: Float):Integer ==
  lcmLim0 := lcmLimit divisorBound
  multer0 := lcmTo lcmLim0
  LenstraEllipticMethod(n, lcmLim0, multer0)
InnerLenstraEllipticMethod(n:Integer, multer:Integer,
  X0:Integer, Y0:Integer, Z0:Integer):Integer ==
  import from EllipticCurveRationalPoints(X0,Y0,Z0,n)
  import from Record(x: Integer, y: Integer, z: Integer)
  p := p0
  pn := multer * p
  Zn := HessianCoordinates.pn.z
  gcd(n, Zn)

LenstraEllipticMethod(n: Integer, multer: Integer):Integer ==
  X0:Integer := random()
  Y0:Integer := random()
  Z0:Integer := random()
  InnerLenstraEllipticMethod(n, multer, X0, Y0, Z0)

LenstraEllipticMethod(n:Integer, lcmLim0:Integer, multer0:Integer):Integer ==

```

```

nfact: Integer := 1
for i:Integer in 1.. while nfact = 1 repeat
  output("Trying elliptic curve number", i::Ex)
  X0:Integer := random()
  Y0:Integer := random()
  Z0:Integer := random()
  nfact := InnerLenstraEllipticMethod(n, multer0, X0, Y0, Z0)
  if nfact = n then
    lcmLim := lcmLim0
    while nfact = n repeat
      output("Too many iterations... backing off")
      lcmLim := bfloor(lcmLim * 0.6)
      multer := lcmTo lcmLim
      nfact := InnerLenstraEllipticMethod(n, multer0, X0, Y0, Z0)
    repeat
  nfact

```

References

- [1] nothing