$SPAD/src/algebra intfact.spad

Michael Monagan

December 14, 2004

Abstract

Contents

1 package PRIMES IntegerPrimesPackage

⟨*package PRIMES IntegerPrimesPackage*⟩≡

```
)abbrev package PRIMES IntegerPrimesPa
++ Author: Michael Monagan
++ Date Created: August 1987
++ Date Last Updated: 31 May 1993
++ Updated by: James Davenport
++ Updated Because: of problems with strong pseudo-primes
++   and for some efficiency reasons.
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: integer, prime
++ Examples:
++ References: Davenport's paper in ISSAC 1992
++             AXIOM Technical Report ATR/6
++ Description:
++   The \spadtype{IntegerPrimesPackage} implements a modification of
++   Rabin's probabilistic
++   primality test and the utility functions \spadfun{nextPrime},
++   \spadfun{prevPrime} and \spadfun{primes}.
IntegerPrimesPackage(I:IntegerNumberSystem): with
   prime?: I -> Boolean
     ++ \spad{prime?(n)} returns true if n is prime and false if not.
     ++ The algorithm used is Rabin's probabilistic primality test
     ++ (reference: Knuth Volume 2 Semi Numerical Algorithms).
     ++ If \spad{prime? n} returns false, n is proven composite.
     ++ If \spad{prime? n} returns true, prime? may be in error
     ++ however, the probability of error is very low.
     ++ and is zero below 25*10**9 (due to a result of Pomerance et al),
     ++ below 10**12 and 10**13 due to results of Pinch,
     ++ and below 341550071728321 due to a result of Jaeschke.
     ++ Specifically, this implementation does at least 10 pseudo prime
     ++ tests and so the probability of error is \spad{< 4**(-10)}.
     ++ The running time of this method is cubic in the length
     ++ of the input n, that is \spad{O( (log n)**3 )}, for n<10**20.
     ++ beyond that, the algorithm is quartic, \spad{O( (log n)**4 )}.
     ++ Two improvements due to Davenport have been incorporated
     ++ which catches some trivial strong pseudo-primes, such as
     ++ [Jaeschke, 1991] 1377161253229053 * 413148375987157, which
     ++ the original algorithm regards as prime
   nextPrime: I -> I
     ++ \spad{nextPrime(n)} returns the smallest prime strictly larger than n
   prevPrime: I -> I
     ++ \spad{prevPrime(n)} returns the largest prime strictly smaller than n
```

```
  primes: (I,I) -> List I
    ++ \spad{primes(a,b)} returns a list of all primes p with
    ++ \spad{a <= p <= b}
== add
  smallPrimes: List I := [2::I,3::I,5::I,7::I,11::I,13::I,17::I,19::I,_
                    23::I,29::I,31::I,37::I,41::I,43::I,47::I,_
                    53::I,59::I,61::I,67::I,71::I,73::I,79::I,_
                    83::I,89::I,97::I,101::I,103::I,107::I,109::I,_
                    113::I,127::I,131::I,137::I,139::I,149::I,151::I,_
                    157::I,163::I,167::I,173::I,179::I,181::I,191::I,_
                    193::I,197::I,199::I,211::I,223::I,227::I,229::I,_
                    233::I,239::I,241::I,251::I,257::I,263::I,269::I,_
                    271::I,277::I,281::I,283::I,293::I,307::I,311::I,_
                    313::I]

  productSmallPrimes    := */smallPrimes
  nextSmallPrime        := 317::I
  nextSmallPrimeSquared := nextSmallPrime**2
  two                   := 2::I
  tenPowerTwenty:=(10::I)**20
  PomeranceList:= [25326001::I, 161304001::I, 960946321::I, 1157839381::I,
                  -- 3215031751::I, -- has a factor of 151
                  3697278427::I, 5764643587::I, 6770862367::I,
                   14386156093::I, 15579919981::I, 18459366157::I,
                    19887974881::I, 21276028621::I ]::(List I)
  PomeranceLimit:=27716349961::I  -- replaces (25*10**9) due to Pinch
  PinchList:= [3215031751::I, 118670087467::I, 128282461501::I, 354864744877::I,
             546348519181::I, 602248359169::I, 669094855201::I ]
  PinchLimit:= (10**12)::I
  PinchList2:= [2152302898747::I, 3474749660383::I]
  PinchLimit2:= (10**13)::I
  JaeschkeLimit:=341550071728321::I
  rootsMinus1:Set I := empty()
  -- used to check whether we detect too many roots of -1
  count2Order:Vector NonNegativeInteger := new(1,0)
  -- used to check whether we observe an element of maximal two-order

  primes(m, n) ==
     -- computes primes from m to n inclusive using prime?
     l:List(I) :=
       m <= two => [two]
       empty()
     n < two or n < m => empty()
     if even? m then m := m + 1
     ll:List(I) := [k::I for k in
            convert(m)@Integer..convert(n)@Integer by 2 | prime?(k::I)]
```

```
      reverse_! concat_!(ll, l)

   rabinProvesComposite : (I,I,I,I,NonNegativeInteger) -> Boolean
   rabinProvesCompositeSmall : (I,I,I,I,NonNegativeInteger) -> Boolean


   rabinProvesCompositeSmall(p,n,nm1,q,k) ==
         -- probability n prime is > 3/4 for each iteration
         -- for most n this probability is much greater than 3/4
         t := powmod(p, q, n)
         -- neither of these cases tells us anything
--         if not (one? t or t = nm1) then
         if not ((t = 1) or t = nm1) then
            for j in 1..k-1 repeat
               oldt := t
               t := mulmod(t, t, n)
--               one? t => return true
               (t = 1) => return true
               -- we have squared someting not -1 and got 1
               t = nm1 =>
                   leave
            not (t = nm1) => return true
         false

   rabinProvesComposite(p,n,nm1,q,k) ==
         -- probability n prime is > 3/4 for each iteration
         -- for most n this probability is much greater than 3/4
         t := powmod(p, q, n)
         -- neither of these cases tells us anything
         if t=nm1 then count2Order(1):=count2Order(1)+1
--         if not (one? t or t = nm1) then
         if not ((t = 1) or t = nm1) then
            for j in 1..k-1 repeat
               oldt := t
               t := mulmod(t, t, n)
--               one? t => return true
               (t = 1) => return true
               -- we have squared someting not -1 and got 1
               t = nm1 =>
                   rootsMinus1:=union(rootsMinus1,oldt)
                   count2Order(j+1):=count2Order(j+1)+1
                   leave
            not (t = nm1) => return true
         # rootsMinus1 > 2 => true  -- Z/nZ can't be a field
         false
```

```
   prime? n ==
      n < two => false
      n < nextSmallPrime => member?(n, smallPrimes)
--      not one? gcd(n, productSmallPrimes) => false
      not (gcd(n, productSmallPrimes) = 1) => false
      n < nextSmallPrimeSquared => true

      nm1 := n-1
      q := (nm1) quo two
      for k in 1.. while not odd? q repeat q := q quo two
      -- q = (n-1) quo 2**k for largest possible k

      n < JaeschkeLimit =>
          rabinProvesCompositeSmall(2::I,n,nm1,q,k) => return false
          rabinProvesCompositeSmall(3::I,n,nm1,q,k) => return false

          n < PomeranceLimit =>
              rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
              member?(n,PomeranceList) => return false
              true

          rabinProvesCompositeSmall(7::I,n,nm1,q,k) => return false
          n < PinchLimit =>
              rabinProvesCompositeSmall(10::I,n,nm1,q,k) => return false
              member?(n,PinchList) => return false
              true

          rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
          rabinProvesCompositeSmall(11::I,n,nm1,q,k) => return false
          n < PinchLimit2 =>
              member?(n,PinchList2) => return false
              true

          rabinProvesCompositeSmall(13::I,n,nm1,q,k) => return false
          rabinProvesCompositeSmall(17::I,n,nm1,q,k) => return false
          true

      rootsMinus1:= empty()
      count2Order := new(k,0) -- vector of k zeroes

      mn := minIndex smallPrimes
      for i in mn+1..mn+10 repeat
          rabinProvesComposite(smallPrimes i,n,nm1,q,k) => return false
      import IntegerRoots(I)
      q > 1 and perfectSquare?(3*n+1) => false
      ((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare?(8*n+1) => false
```

```
   -- Both previous tests from Damgard & Landrock
   currPrime:=smallPrimes(mn+10)
   probablySafe:=tenPowerTwenty
   while count2Order(k) = 0 or n > probablySafe repeat
       currPrime := nextPrime currPrime
       probablySafe:=probablySafe*(100::I)
       rabinProvesComposite(currPrime,n,nm1,q,k) => return false
   true

nextPrime n ==
   -- computes the first prime after n
   n < two => two
   if odd? n then n := n + two else n := n + 1
   while not prime? n repeat n := n + two
   n

prevPrime n ==
   -- computes the first prime before n
   n < 3::I => error "no primes less than 2"
   n = 3::I => two
   if odd? n then n := n - two else n := n - 1
   while not prime? n repeat n := n - two
   n
```

2 package IROOT IntegerRoots

⟨*package IROOT IntegerRoots* ⟩≡

```
)abbrev package IROOT IntegerRoots
++ Author: Michael Monagan
++ Date Created: November 1987
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: integer roots
++ Examples:
++ References:
++ Description: The \spadtype{IntegerRoots} package computes square roots and
++   nth roots of integers efficiently.
IntegerRoots(I:IntegerNumberSystem): Exports == Implementation where
  NNI ==> NonNegativeInteger

  Exports ==> with
    perfectNthPower?: (I, NNI) -> Boolean
      ++ \spad{perfectNthPower?(n,r)} returns true if n is an \spad{r}th
      ++ power and false otherwise
    perfectNthRoot: (I,NNI) -> Union(I,"failed")
      ++ \spad{perfectNthRoot(n,r)} returns the \spad{r}th root of n if n
      ++ is an \spad{r}th power and returns "failed" otherwise
    perfectNthRoot: I -> Record(base:I, exponent:NNI)
      ++ \spad{perfectNthRoot(n)} returns \spad{[x,r]}, where \spad{n = x\^r}
      ++ and r is the largest integer such that n is a perfect \spad{r}th power
    approxNthRoot: (I,NNI) -> I
      ++ \spad{approxRoot(n,r)} returns an approximation x
      ++ to \spad{n**(1/r)} such that \spad{-1 < x - n**(1/r) < 1}
    perfectSquare?: I -> Boolean
      ++ \spad{perfectSquare?(n)} returns true if n is a perfect square
      ++ and false otherwise
    perfectSqrt: I -> Union(I,"failed")
      ++ \spad{perfectSqrt(n)} returns the square root of n if n is a
      ++ perfect square and returns "failed" otherwise
    approxSqrt: I -> I
      ++ \spad{approxSqrt(n)} returns an approximation x
      ++ to \spad{sqrt(n)} such that \spad{-1 < x - sqrt(n) < 1}.
      ++ Compute an approximation s to \spad{sqrt(n)} such that
      ++           \spad{-1 < s - sqrt(n) < 1}
      ++ A variable precision Newton iteration is used.
```

```
    ++ The running time is \spad{O( log(n)**2 )}.

  Implementation ==> add
    import IntegerPrimesPackage(I)

    resMod144: List I := [0::I,1::I,4::I,9::I,16::I,25::I,36::I,49::I,_
                     52::I,64::I,73::I,81::I,97::I,100::I,112::I,121::I]
    two := 2::I


    perfectSquare? a       == (perfectSqrt a) case I
    perfectNthPower?(b, n) == perfectNthRoot(b, n) case I

    perfectNthRoot n ==  -- complexity (log log n)**2 (log n)**2
      m:NNI
--      one? n or zero? n or n = -1 => [n, 1]
      (n = 1) or zero? n or n = -1 => [n, 1]
      e:NNI := 1
      p:NNI := 2
      while p::I <= length(n) + 1 repeat
         for m in 0.. while (r := perfectNthRoot(n, p)) case I repeat
            n := r::I
         e := e * p ** m
         p := convert(nextPrime(p::I))@Integer :: NNI
      [n, e]

    approxNthRoot(a, n) ==   -- complexity (log log n) (log n)**2
      zero? n => error "invalid arguments"
--      one? n => a
      (n = 1) => a
      n=2 => approxSqrt a
      negative? a =>
        odd? n => - approxNthRoot(-a, n)
        0
      zero? a => 0
--      one? a => 1
      (a = 1) => 1
      -- quick check for case of large n
      ((3*n) quo 2)::I >= (l := length a) => two
      -- the initial approximation must be >= the root
      y := max(two, shift(1, (n::I+l-1) quo (n::I)))
      z:I := 1
      n1:= (n-1)::NNI
      while z > 0 repeat
        x := y
        xn:= x**n1
```

```
    y := (n1*x*xn+a) quo (n*xn)
    z := x-y
  x

perfectNthRoot(b, n) ==
  (r := approxNthRoot(b, n)) ** n = b => r
  "failed"

perfectSqrt a ==
  a < 0 or not member?(a rem (144::I), resMod144) => "failed"
  (s := approxSqrt a) * s = a => s
  "failed"

approxSqrt a ==
  a < 1 => 0
  if (n := length a) > (100::I) then
     -- variable precision newton iteration
     n := n quo (4::I)
     s := approxSqrt shift(a, -2 * n)
     s := shift(s, n)
     return ((1 + s + a quo s) quo two)
  -- initial approximation for the root is within a factor of 2
  (new, old) := (shift(1, n quo two), 1)
  while new ^= old repeat
     (new, old) := ((1 + new + a quo new) quo two, new)
  new
```

3 package INTFACT IntegerFactorizationPackage

⟨*package INTFACT IntegerFactorizationPackage* ⟩≡

```
)abbrev package INTFACT IntegerFactorizationP
++ This Package contains basic methods for integer factorization.
++ The factor operation employs trial division up to 10,000.  It
++ then tests to see if n is a perfect power before using Pollards
++ rho method.  Because Pollards method may fail, the result
++ of factor may contain composite factors.  We should also employ
++ Lenstra's eliptic curve method.

IntegerFactorizationPackage(I): Exports == Implementation where
  I: IntegerNumberSystem

  B      ==> Boolean
  FF     ==> Factored I
  NNI    ==> NonNegativeInteger
  LMI    ==> ListMultiDictionary I
  FFE    ==> Record(flg:Union("nil","sqfr","irred","prime"),
                                                   fctr:I, xpnt:Integer)

  Exports ==>  with
    factor : I -> FF
      ++ factor(n) returns the full factorization of integer n
    squareFree   : I -> FF
      ++ squareFree(n) returns the square free factorization of integer n
    BasicMethod : I -> FF
      ++ BasicMethod(n) returns the factorization
      ++ of integer n by trial division
    PollardSmallFactor: I -> Union(I,"failed")
       ++ PollardSmallFactor(n) returns a factor
       ++ of n or "failed" if no one is found

  Implementation ==> add
    import IntegerRoots(I)

    BasicSieve: (I, I) -> FF

    squareFree(n:I):FF ==
       u:I
       if n<0 then (m := -n; u := -1)
              else (m := n; u := 1)
```

```
      (m > 1) and ((v := perfectSqrt m) case I) =>
         for rec in (l := factorList(sv := squareFree(v::I))) repeat
           rec.xpnt := 2 * rec.xpnt
         makeFR(u * unit sv, l)
   -- avoid using basic sieve when the lim is too big
      lim := 1 + approxNthRoot(m,3)
      lim > (100000::I) => makeFR(u, factorList factor m)
      x := BasicSieve(m, lim)
      y :=
--         one?(m:= unit x) => factorList x
        ((m:= unit x) = 1) => factorList x
        (v := perfectSqrt m) case I =>
           concat_!(factorList x, ["sqfr",v,2]$FFE)
        concat_!(factorList x, ["sqfr",m,1]$FFE)
      makeFR(u, y)

   -- Pfun(y: I,n: I): I == (y**2 + 5) rem n
   PollardSmallFactor(n:I):Union(I,"failed") ==
      -- Use the Brent variation
      x0 := random()$I
      m := 100::I
      y := x0 rem n
      r:I := 1
      q:I := 1
      G:I := 1
      until G > 1 repeat
         x := y
         for i in 1..convert(r)@Integer repeat
            y := (y*y+5::I) rem n
            q := (q*abs(x-y)) rem n
            k:I := 0
         until (k>=r) or (G>1) repeat
            ys := y
            for i in 1..convert(min(m,r-k))@Integer repeat
               y := (y*y+5::I) rem n
               q := q*abs(x-y) rem n
            G := gcd(q,n)
            k := k+m
         r := 2*r
      if G=n then
         until G>1 repeat
            ys := (ys*ys+5::I) rem n
            G := gcd(abs(x-ys),n)
      G=n => "failed"
      G
```

```
BasicSieve(r, lim) ==
   l:List(I) :=
      [1::I,2::I,2::I,4::I,2::I,4::I,2::I,4::I,6::I,2::I,6::I]
   concat_!(l, rest(l, 3))
   d := 2::I
   n := r
   ls := empty()$List(FFE)
   for s in l repeat
      d > lim => return makeFR(n, ls)
      if n<d*d then
         if n>1 then ls := concat_!(ls, ["prime",n,1]$FFE)
         return makeFR(1, ls)
      for m in 0.. while zero?(n rem d) repeat n := n quo d
      if m>0 then ls := concat_!(ls, ["prime",d,convert m]$FFE)
      d := d+s

BasicMethod n ==
   u:I
   if n<0 then (m := -n; u := -1)
          else (m := n; u := 1)
   x := BasicSieve(m, 1 + approxSqrt m)
   makeFR(u, factorList x)

factor m ==
   u:I
   zero? m => 0
   if negative? m then (n := -m; u := -1)
                  else (n := m; u := 1)
   b := BasicSieve(n, 10000::I)
   flb := factorList b
--       one?(n := unit b) => makeFR(u, flb)
   ((n := unit b) = 1) => makeFR(u, flb)
   a:LMI := dictionary() -- numbers yet to be factored
   b:LMI := dictionary() -- prime factors found
   f:LMI := dictionary() -- number which could not be factored
   insert_!(n, a)
   while not empty? a repeat
      n := inspect a; c := count(n, a); remove_!(n, a)
      prime?(n)$IntegerPrimesPackage(I) => insert_!(n, b, c)
      -- test for a perfect power
      (s := perfectNthRoot n).exponent > 1 =>
        insert_!(s.base, a, c * s.exponent)
      -- test for a difference of square
      x:=approxSqrt n;
      if (x**2<n) then x:=x+1
      (y:=perfectSqrt (x**2-n)) case I =>
```

```
         insert_!(x+y,a,c)
         insert_!(x-y,a,c)
   (d := PollardSmallFactor n) case I =>
      for m in 0.. while zero?(n rem d) repeat n := n quo d
      insert_!(d, a, m * c)
      if n > 1 then insert_!(n, a, c)
   -- an elliptic curve factorization attempt should be made here
   insert_!(n, f, c)
-- insert prime factors found
while not empty? b repeat
   n := inspect b; c := count(n, b); remove_!(n, b)
   flb := concat_!(flb, ["prime",n,convert c]$FFE)
-- insert non-prime factors found
while not empty? f repeat
   n := inspect f; c := count(n, f); remove_!(n, f)
   flb := concat_!(flb, ["nil",n,convert c]$FFE)
makeFR(u, flb)
```

4 License

⟨*license* ⟩≡

```
--Copyright (c) 1991-2002, The Numerical ALgorithms Group
--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.
```

⟨*** ⟩≡ ⟨*lic*

⟨*package PRIMES IntegerPrimesPackage* ⟩
⟨*package IROOT IntegerRoots* ⟩
⟨*package INTFACT IntegerFactorizationPackage* ⟩

References

[1] nothing