```
/*
 * Copyright (C) 2009, 2010 Thomas Steinke
 * (schonhagestrassen@thomassteinke.org) 2010-03-09
 * This is a program to test the Schonhage-Strassen Algorithm using
 * complex interval arithmetic.
 * It uses the C-XSC library for validated arithmetic.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

// to compile:
// g++ multiply.cpp -I/usr/local/include -I/usr/local/cxsc/include
//     -L/usr/local/cxsc/lib -lcxsc -lm -o SSmultiply
/*
Include standard libraries
*/
#include <iostream>
#include <cassert>
#include <fstream>
#include <ctime>

/*
Include C-XSC libraries
*/
#include "real.hpp"
#include "interval.hpp"
#include "complex.hpp"
#include "cinterval.hpp"
#include "rmath.hpp"
#include "imath.hpp"


/*
Clock resolution
*/
#ifndef CLOCKS_PER_SEC
#ifdef CLK_PER_SEC
#define CLOCKS_PER_SEC CLK_PER_SEC
#else
#error CLOCKS_PER_SEC Not Defined
#endif
#endif

using namespace std;
using namespace cxsc;

/*****************************************************************************
Random number generation code for generating random test cases
*****************************************************************************/

/*
Delegate random number generation to the operating system.
This will not work on all systems; if it doesn't, implement something else here.
*/
ifstream urand("/dev/urandom");

/*
Generate a random bit
*/
```

```
bool RandomBit() {
  return ((urand.get() % 2) == 0);
}

/*****************************************************************************
We define a class to represent natural numbers that we can then multiply. We
define basic operations. Later we implement the Schonhage-Strassen fast
multiplication algorithm.
*****************************************************************************/
typedef unsigned long long unint; //The basic hardware integer we use


/*
This class represents a natural number.
Since there are several multiplication algorithms that need to access the data
contained in Natural, they are left public and much of the management is done
outside of the class. Bad karma, but meh!
*/
struct Natural {
  unint * x; //digits
  int n; //number of digits
  int k; //number of bits per digit
  //value = x[0] + 2^k*x[1] + 2^(2*k)*x[2] + ... + 2^((n-1)*k)*x[n-1]

  /*
  Get the lth bit of n.
  */
  unint GetBit(int l) const {
    if (l < 0 || l >= n * k) return ((unint) 0);
    else return (x[l / k] >> (l % k)) & ((unint) 1);
  }

  /*
  Compare -- used for checking correctness of results
  */
  bool operator==(const Natural & other) const {
    assert(k == other.k);
    if (n != other.n) return false;
    if (n == 0) return true;
    for (int i = 0; i < n; i++)
      if (x[i] != other.x[i]) return false;
    return true;
  }

  bool operator!=(const Natural & other) const {
    return !(*this == other);
  }
};

/*
Create a random n-digit number in base 2^k
*/
Natural RandomNumber(int n, int k) {
  Natural a;
  a.n = n;
  a.k = k;
  a.x = new unint[n];
  for (int i = 0; i < n; i++) {
    a.x[i] = 0;
    for (int j = 0; j < k; j++) {
      a.x[i] = ((a.x[i] << 1) | (RandomBit() ? (unint) 1 : (unint) 0));
    }
  }
  a.x[n - 1] = (a.x[n - 1] | (((unint) 1) << (k - 1)));
  return a;
}

/*
Output hexadecimal
*/
ostream & operator<<(ostream & s, const Natural & n) {
```

```
  if (n.n == 0) {
    s << '0';
  } else {
    const char * hexdigits = "0123456789abcdef";
    assert(n.n > 0 && n.x[n.n - 1] != 0);
    int nbits = 0;
    while (n.x[n.n - 1] >= (((unint) 1) << nbits)) nbits++;
    nbits += (n.n - 1) * n.k;
    int ndigits = nbits / 4 + (nbits % 4 == 0 ? 0 : 1);
    for (int i = ndigits - 1; i >= 0; i--) {
      int k = n.GetBit(4*i) + 2 * n.GetBit(4 * i + 1) + 4 * n.GetBit(
                4 * i + 2) + 8 * n.GetBit(4 * i + 3);
      s << hexdigits[k];
    }
  }
  return s;
}

/*
Multiply two numbers using the O(n^2) elementary method
*/
Natural Multiply(const Natural & a, const Natural & b) {
  assert(a.k == b.k);
  if (a.n == 0 || b.n == 0) {//zero
    Natural cc;
    cc.x = NULL;
    cc.n = 0;
    cc.k = a.k;
    return cc;
  }
  assert((int) sizeof(unint) * 4 >=
         a.k); // if this isn't true we are entering overflow territory
  unint base = (((unint) 1) << a.k);
  Natural c;
  c.n = a.n + b.n;
  c.k = a.k;
  c.x = new unint[c.n];
  unint carry = 0;
  for (int i = 0; i < c.n; i++) {
    unint newcarry = carry / base;
    carry = carry % base;
    for (int j = (i - b.n + 1 > 0 ? i - b.n + 1 : 0); j <= i && j < a.n; j++) {
      carry += a.x[j] * b.x[i - j];
      newcarry += carry / base;
      carry = carry % base;
    }
    c.x[i] = carry;
    carry = newcarry;
  }
  assert(carry == 0);
  while (c.n > 0 && c.x[c.n - 1] == 0) c.n--;
  return c;
}

/*****************************************************************************
The Complex number handling routines. These need to provide complex arithmetic
and they need to round complex numbers to unints. Most of this is done by the
C-XSC library.
*****************************************************************************/

/*
Complex class based on a real type such as float or double
*/
template <class R> class Complex {
  R x, y; //Real and imaginary parts.
public:
  //Just the basic operations
  Complex<R>(R xx = 0, R yy = 0) : x(xx), y(yy) {}
  Complex<R> operator+(const Complex<R> & other) const {
    return Complex<R>(x + other.x, y + other.y);
  }
```

```
  Complex<R> operator-(const Complex<R> & other) const {
    return Complex<R>(x - other.x, y - other.y);
  }
  Complex<R> operator-() const {
    return Complex<R>(-x, -y);
  }
  Complex<R> operator*(const Complex<R> & other) const {
    return Complex<R>(x * other.x - y * other.y, x * other.y + y * other.x);
  }
  Complex<R> operator/(const Complex<R> & other) const {
    R t = (other.x * other.x + other.y * other.y);
    return *this * Complex<R>(other.x / t, - other.y / t);
  }
  R re() const {
    return x;
  }
  R im() const {
    return y;
  }
};

/*
Round Complex<R> x to the nearest non-negative integer
*/
template <class R> unint Roundunint(const Complex<R> & x) {
  //unint i = floor(Re(x));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint min = 0;
  unint max = 1;
  while (((R) (double) max) <= x.re()) max *= 2;
  while (min + 1 < max) {
    unint mid = (min + max) / 2;
    if (((R) (double) mid) <= x.re()) {
      min = mid;
    } else {
      max = mid;
    }
  }
  unint i = (abs(x.re() - (R) (double) min) <= abs(x.re() - (R) (
              double) max) ? min : max);
  return i;
}

/*
Do the same with a C-XSC complex point
*/
unint Roundunint(const complex & x) {
  //unint i = floor(Re(x));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint min = 0;
  unint max = 1;
  while (((real) (double) max) <= Re(x)) max *= 2;
  while (min + 1 < max) {
    unint mid = (min + max) / 2;
    if (((real) (double) mid) <= Re(x)) {
      min = mid;
    } else {
      max = mid;
    }
  }
  unint i = (abs(Re(x) - (real) (double) min) <= abs(Re(x) - (real) (
              double) max) ? min : max);
  return i;
}

/*
Find the unique integer in the cinterval x
If none exists throw an exception
*/
```

```
unint Roundunint(const cinterval & x) {
  assert(0 <= Im(x));
  //unint i = floor(Sup(Re(x)));
  //This is sinfully inefficient, but there is no automatic function for
  //converting a real to an int should take O(log(x)) time
  unint i = Roundunint(complex(Sup(Re(x))));
  if (((real) (double) i) <= Re(x)) {
    //do nothing
  } else if (((real) (double) i + 1) <= Re(x)) {
    i++;
  } else if (((real) (double) i - 1) <= Re(x)) {
    i--;
  } else {
    assert(false);
  }
  if (((real) (double) i + 1) <= Re(x) || (i > 0
      && ((real) (double) i - 1) <= Re(x)))  {
    throw diam(Re(x));
  }
  return i;
}

/*****************************************************************************
Now we have the Schonhage-Strassen Algorithm proper (with some preceeding
subroutines).
*****************************************************************************/

/*
Calculate the vector [1, w, w^2, ..., w ^ (2^n - 1)], where w = exp(2*Pi*i/2^n)
That is, we compute the 2^n-th primitive roots of unity.
R is a real class and C is a corresponding complex class.
R must be compatible with integers and have a sqrt function;
C must have a C(R, R) constructor.
*/
template<class R, class C> C * ComplexRoots(int n) {
  //(k = 0) cos(2*Pi/1) = 1
  //(k = 1) cos(2*Pi/2) = -1
  //(k = 2) cos(2*Pi/4) = 0
  //(k >= 3) cos(2*Pi/2^k) = sqrt((1 + cos(2*Pi/2^(k-1)))/2)  where the
  //square root is of a non-negative real number
  //(k = 0) sin(2*Pi/1) = 0
  //(k >= 1) sin(2*Pi/2^k) = sqrt((1 - cos(2*Pi/2^(k-1)))/2) where the
  //square root is of a non-negative real number
  //(k >= 0) exp(2*Pi*i/2^k) = cos(2*Pi/2^k) + i*sin(2*PI/2^k)
  //cosines[k] = cos(2*Pi/2^k); sines[k] = sin(2*Pi/2^k)
  R * cosines = new R[n + 3];//Extra spaces to avoid going over the end
  R * sines = new R[n + 1];
  cosines[0] = 1;
  cosines[1] = -1;
  cosines[2] = 0;
  for (int k = 3; k <= n; k++)
    cosines[k] = sqrt(((1 + cosines[k-1])/2));
  sines[0] = 0;
  for (int k = 1; k <= n; k++)
    sines[k] = sqrt(((1 - cosines[k-1])/2));
  //ans[k] = exp(2*Pi*i/2^n)^k
  C * ans = new C[1 << n];
  for (int i = 0; i < (1 << n); i++) {
    C x(1);
    for (int j = 0; j < n; j++) {
      if ((i & (1 << j)) != 0) {
        //x *= exp(2*Pi*i/2^n)^(2^j)
        x = x * C(cosines[n-j], sines[n-j]);
      }
    }
    ans[i] = x; // & exp(cinterval(0, (2 * Pi() * i) / (1 << n)));
  }
  delete[] cosines;
  delete[] sines;
  return ans;
}
```

```
/*
Reverse the binary representation of the n-bit integer x
-- used by the DFT algorithm
*/
int ReverseBinary(int n, int x) {
  int y = 0;
  for (int i = 0; i < n; i++) {
    if ((x & (1 << i)) != 0) {
      y = (y | (1 << (n - i - 1)));
    }
  }
  return y;
}

/*
Compute the discrete fourier transform of input and store in output
Both input and output have 2^n elements. Roots is a vector of the 2^n 2^n-th
primitive roots of unity if the inverse flag is set to true, then the inverse
discrete fourier transform is computed instead.
*/
template <class C> void DFT(bool inverse, int n, const C * roots, C * output,
                            const C * input) {
  int m = (1 << n);
  for (int i = 0; i < m; i++) {
    output[ReverseBinary(n, i)] = input[i];
  }
  for (int i = 1; i <= n; i++) {
    //Split into 2^(n-i) blocks of 2^i and do dft
    for (int j = 0; j < (1 << (n - i)); j++) {
      //block is j*2^i to (j+1)*2^i-1
      //w = roots[1 << (n - i)]
      for (int k = 0; k < (1 << (i - 1)); k++) {
        C u, v, w;
        int l = ((1 << (n - i)) * k) % (1 << n);
        if (inverse) {
          l = ((1 << n) - l) % (1 << n);
        }
        w = roots[l];
        u = output[j * (1 << i) + k];
        v = w * output[j * (1 << i) + (1 << (i - 1)) + k];
        output[j * (1 << i) + k] = u + v;
        output[j * (1 << i) + (1 << (i - 1)) + k] = u - v;
      }
    }
  }
  if (inverse) {
    for (int i = 0; i < (1 << n); i++) {
      output[i] = output[i] / (1 << n);
    }
  }
}

/*
This is the Schonhage-Strassen multiplication algorithm
R and C are as in the function ComplexRoots, and represent the real and
complex types to be used for the floating-point computations.
*/
template <class R, class C> Natural SchonhageStrassen(const Natural & a,
    const Natural & b) {
  assert(a.k == b.k);
  if (a.n == 0 || b.n == 0) {//zero
    Natural cc;
    cc.x = NULL;
    cc.n = 0;
    cc.k = a.k;
    return cc;
  }
  //First we need to find a power of 2 bigger than or equal to a.n + b.n
  int n = 0;
  while ((1 << n) < a.n + b.n) n++;
```

```
  unint base = (((unint) 1) << a.k);
  //First, calculate roots
  C * roots = ComplexRoots<R, C>(n);
  //Now three working arrays
  C * work1 = new C[1 << n];
  C * work2 = new C[1 << n];
  C * work3 = new C[1 << n];
  //Load a in to work1
  for (int i = 0; i < a.n; i++) work1[i] = ((R) (double) a.x[i]);
  for (int i = a.n; i < (1 << n); i++) work1[i] = 0;
  //Do dft
  DFT<C>(false, n, roots, work2, work1);
  //Load b in to work1
  for (int i = 0; i < b.n; i++) work1[i] = ((R) (double) b.x[i]);
  for (int i = b.n; i < (1 << n); i++) work1[i] = 0;
  //Do dft
  DFT<C>(false, n, roots, work3, work1);
  //Pointwise multiplication
  for (int i = 0; i < (1 << n); i++) work1[i] = work2[i] * work3[i];
  //inverse fourier transform
  DFT<C>(true, n, roots, work2, work1);
  //Now read c out of work2
  Natural c;
  c.n = (1 << n);
  c.k = a.k;
  c.x = new unint[c.n];
  unint carry = 0;
  bool issuccessful = true;
  real ball = 0; //whatever we catch and then pass on
  try {
    for (int i = 0; i < c.n; i++) {
      carry += Roundunint(work2[i]);
      c.x[i] = carry % base;
      carry = carry / base;
    }
  } catch (real exc) {
    issuccessful = false;
    ball = exc;
  }
  //clean up
  while (c.n > 0 && c.x[c.n - 1] == 0) c.n--;
  delete[] roots;
  delete[] work1;
  delete[] work2;
  delete[] work3;
  if (!issuccessful || carry != 0) {
    delete[] c.x;
    c.x = NULL;
    c.n = 0;
    throw ball;
  }
  return c;
}

/*****************************************************************************
Now there are just some analysis routines. Testing and timing the algorithm.
*****************************************************************************/

/*
First we package every algorithm into a nice Algorithm object. Then we can
refer to algorithms just by numbers
*/
struct Algorithm {
  const char * name; // The name of the algorithm
  Natural (*function)(const Natural &,
                      const Natural
                      &); // The algorithm itself -- note that this may
  // throw an exception (in the form of a double) if it fails
  Algorithm(const char * n, Natural (*f)(const Natural &,
                                         const Natural &)) :
                                                 name(n), function(f) {}
```

```
};

Natural floatMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<float, Complex<float> >(a, b);
}
Natural doubleMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<double, Complex<double> >(a, b);
}
Natural pointMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<real, complex >(a, b);
}
Natural setMultiply(const Natural & a, const Natural & b) {
  return SchonhageStrassen<interval, cinterval >(a, b);
}

Algorithm algorithms[] = {
  Algorithm("elementary multiplication algorithm", &Multiply),
  Algorithm("float-based Schonhage-Strassen algorithm", &floatMultiply),
  Algorithm("double-based Schonhage-Strassen algorithm", &doubleMultiply),
  Algorithm("basic Schonhage-Strassen algorithm", &pointMultiply),
  Algorithm("Schonhage-Strassen algorithm with containment sets", &setMultiply),
};

/*
This function will run the algorithms specified by mask on two random n-digit
base-2^k numbers. If (mask & 1 != 0), then the remaining outputs are compared
to this output to check for correctness.
(algorithm 0 is deemed to be the standard for correctness)
If data != NULL, then we output the data in CSV format (for analysis).
*/
void runalgorithm(int mask, int n, int k, int r, ostream * data) {
  int numalgorithms = sizeof(algorithms) / sizeof(
                        Algorithm); // Total number of algorithms
  for(int reps=0; reps < r; reps++) {
    //Generate test data
    Natural a = RandomNumber(n, k);
    Natural b = RandomNumber(n, k);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "n = " << n << " k = " << k << endl;
    Natural alg0;//this is the output of alg0 to compare with
    for (int i = 0; i < numalgorithms; i++) {
      if ((mask & (1 << i)) != 0) {
        int status = 0; //This identifies the outcome of the computation
        Natural axb;
        clock_t start = clock();
        try {
          axb = (*(algorithms[i].function))(a, b);
        } catch (real e) {
          cout << "The " << algorithms[i].name << " fails (" << e << ")."
               << endl;
          status = status | 1; //set the fail bit
          axb.n = 0;
          axb.k = 0;
          axb.x = NULL;
        }
        clock_t finish = clock();
        double elapsedtime = ((double) (finish - start))
                                          / ((double) CLOCKS_PER_SEC);
        cout << "The " << algorithms[i].name << " took " << elapsedtime
             << "s." << endl;
        //if we have already run algorithm 0
        if (i > 0 && (mask & 1) != 0) {
          //do check
          if (axb.x != NULL && axb != alg0) {
            //error, incorrect result
            cout << "The " << algorithms[i].name
                 << " gave an incorrect result." << endl;
            status = status | 2; //set the incorrect bit;
          }
        } else {
```

```
          status = status | 4; //set the not-checked bit
        }
        //status 0: algorithm succeeded
        //status 1: algorithm failed
        //status 2: incorrect result
        //status 3: algorithm failed
        //status 4: no error detected -- not checked
        //status 5: algorithm failed
        //status 6: invalid
        //status 7: algorithm failed
        if (data != NULL) {
          //Output data in .csv format
          // <algorithm number>,<status>,<n>,<k>,<time/s>
          *data << i << ",";
          *data << status << ",";
          *data << n << ",";
          *data << k << ",";
          *data << elapsedtime << endl;
        }
        //clean up or store for later
        if (i == 0) {
          alg0 = axb;
        } else {
          delete[] axb.x;
        }
      }
    }
    //last clean up
    if ((mask & 1) != 0) delete[] alg0.x;
    delete[] a.x;
    delete[] b.x;
    cout << endl;
  }
}

/*
The main function only calls runalgorithm. Essentially all it does is read in
4 integers and call runalgorithm with those parameters.  For example the input
"31 100 8 10" would create two 100-digit base-256 natural numbers and multiply
them with all five available algorithms 10 times.
It does provide facilities to log output. One command line parameter will
specify a file to write CSV format information to.
*/
/* you can pass 'in' into the executible 'SSmultiply' and output to 'out'
   as follows:
$ cat in | ./SSmultiply out
$ cat in
31        1024           8         100
31        2048           8         100
31        4096           8         100
31        8192           8         100
31       16384           8         100
31       32768           8         100
31       65536           8         100
*/
int main(int argc, char ** argv) {
  ostream * data = NULL;
  if (argc == 2) {
    data = (ostream *) new ofstream(argv[1], ios_base::app);
  }
  while (true) {
    int mask, n, k, r;
    cin >> mask >> n >> k >> r;
    if (cin.eof() || mask < 0
        || mask >= (int) (1 << (sizeof(algorithms) / sizeof(Algorithm)))
        || n <= 0 || k <= 1 || k > (int) sizeof(unint) * 4) break; //invalid
    runalgorithm(mask, n, k, r, data);
  }
  if (data != NULL) delete (ofstream *) data;
  return 0;
}
```