Implementation of the Quadratic Sieve

Mark Gordon
Cryptography 475
November 30th, 2008

Introduction

Finding the factorization of a number, n, is a computationally difficult problem that is at the heart of the security of the RSA encryption algorithm. The naïve solution, trial division, takes $O(\sqrt{n})$ time and is ineffective for even moderately large n. The quadratic sieve offers an improvement on this algorithm and is currently the second asymptotically fastest algorithm for factoring integers. In this paper I present my implementation of the quadratic sieve and its results.

The Quadratic Sieve

The main idea behind the quadratic sieve is that if we have a and b such that $a^2 \equiv b^2 \pmod{n}$ then we have $a^2 - b^2 = n \pmod{n}$ which can be re-written as $(a-b)(a+b) = n \pmod{n}$. Therefore we can take either $a-b$ or $a+b$ and test if it contains a non trivial factor of n using the Euclidean algorithm. One idea to find perfect squares is to find pairs $(x, Q(x))$, where $Q(x)$ is $x^2 - n$ (different polynomials are possible), and test if $Q(x)$ is a perfect square. Note that $x^2 = Q(x) \pmod{n}$ so if $Q(x)$ happens to be a perfect square we have a congruence of squares which will allow us to factor n.

Finding Squares

Unfortunately there are very few $x$ such that $Q(x)$ will be a perfect square. An improvement on this concept would be to try and combine pairs $(x, Q(x))$ and $(y, Q(y))$ to yield $(xy, Q(x)Q(y))$. Note that the congruence $(xy)^2 = Q(x)Q(y) \pmod{n}$ still holds so if it happens that $Q(x)Q(y)$ is a perfect square we once again have a congruence of squares. If we view $Q(x)$ as an exponent vector, $Q(x) = p_1^{v_1}p_2^{v_2} \cdots p_B^{v_B}$, then $Q(x)$ would be a perfect square if and only if every $v_i$ was even.

Linear System

Restating the problem at hand; we want to take the product of some subset of $Q(x)$ such that each exponent in the exponent vector is even. This is equivalent to saying we want to add some subset of vectors such that they sum to 0 modulo 2. This is the same
as saying we want to find a linear dependence between the vectors modulo 2 which can be solved easily with Gaussian Elimination.

**Factor Base**

Because there are a lot of primes that could divide Q(x) requiring us to collect a lot of pairs (x, Q(x)) (and because factoring is hard) it will help if we choose some factor base, F, such that we only collect pairs (x, Q(x)) such that Q(x) factors completely over F. With a restricted size on F we will only need to gather |F| +1 pairs to ensure that we will have a linear dependence and thus be able to generate congruent squares solving the factoring problem.

**Preparations**

In order to do anything with large integers a library for doing basic integer operations is required since native integer types on any processor have a fixed (small) size. To accomplish this I wrote a bigint class that stores the number it represents in the base 2^{30} and handles all of the basic arithmetic operations (+, -, *, /, %) and comparisons one would expect.

In addition to basic arithmetic the bigint library contains some other functions that were needed to implement the quadratic sieve. This includes:

- A square root function that computes the floor of the square root of a big integer.
- A gcd function for computing the greatest common divisor of two big integers using the Euclidean Algorithm.
- A modular exponentiation algorithm for raising b^e modulo m using repeated squaring.
- A modular inverse algorithm for computing x^1 modulo m using the Extended Euclidean Algorithm.
- A method for detecting if a number is probably prime using the Miller-Rabin probabilistic primality test.
- A method for generating random numbers of a fixed number of bits.
- A method for generating a random prime of a fixed number of bits. (This was useful for testing)
A method for computing the Legendre Symbol. This symbol tells you about the existence of square roots of a number modulo a prime. This was implemented as described in [4].

A method for computing the square root of a number modulo a prime. This uses the Shanks-Tonelli algorithm and is implemented as described in [3].

The Algorithm

All of the previous preparation has been for the purpose of implementing the quadratic sieve which is located within the bigint::factor(bool) method in my implementation (Appendix B). In fact the factor method isn’t just the quadratic sieve as it tries trial division to remove small primes to start with, checks if the number is probably prime (and therefore has no factors), runs Pollard’s Rho algorithm for a fixed number of iterations to try and get lucky and find some factors, and then finally moves on to the quadratic sieve.

Computing the factor base

The quadratic sieve starts out by calculating a factor base. The size of the factor base is calculated as $B = 2^{\frac{2\sqrt{\ln(n)\ln(\ln(n))}}{4}}$ approximately as suggested in [1]. The elements of $F$ are chosen as the first $B$ primes that have a quadratic residue modulo $n$. The Sieve of Eratosthenes is used for quickly calculating primes and Legendre’s symbol is checked to be 1.

After calculating the factor base the square root of $n$ modulo $p$ is calculated for each $p$ in $F$ using the Shanks-Tonelli algorithm. This square root is useful because it allows us to find up to two $x$’s such that $Q(x)$ is divisible by $p$. Then this can be used, together with the fact that $Q(x) = Q(x + p) \pmod{p}$, to find two arithmetic progressions that represent all $x$’s such that $p$ divides $Q(x)$.

Sieving

Now we’re ready to sieve. Traditionally sieving is done by selecting some sieving interval and dividing out the largest power of each prime $p$ from each element in the arithmetic progressions associated with that prime within the sieving interval. If at the end of sieving an element is 1 then the associated $Q(x)$ is factorable over the factor base.
Instead of doing that I chose to start at \( x=1+\text{floor}(\sqrt{n}) \) and just keep going forward iteratively. To tell what primes should be divided out of \( Q(x) \) I instead keep a heap that tracks what primes will be appearing next. The main advantages to this approach are that I don't need tons of extra memory, I don't have to try and guess the size of the sieving interval, and I can get pairs \((x, Q(x))\) early and test for linear dependencies as I go. However these advantages come at a slight cost to runtime due to the heap operations.

**Gaussian Elimination**

Unlike the approach described in [1], I search for linear dependencies as I find new pairs. This usually reduces the number of pairs required but the cut is not substantial. The search for linear dependencies is just done using Gaussian elimination optimized for sparse matrices. The elimination is done carefully enough so that the actual subset of pairs can be reconstructed that formed a linear dependence.

**Constructing a Solution**

After finding the subset of pairs that creates the linear dependence we can calculate \( a \) and \( b \) such that \( a^2 \equiv b^2 \pmod{n} \). \( a \) will be calculated as the product of each of \( x \) modulo \( n \) in the selected pairs. Then \( b \) can be computed by calculating the exponent vector of the product of each \( Q(x) \) and halving each exponent. Finally we can take \( \gcd(a - b, n) \) and \( \gcd(a + b, n) \) to try and find a non trivial factor of \( n \).

**Double Large Primes**

In addition to tracking pairs \((x, Q(x))\) that factor completely over the factor base I also track when \( Q(x) \) almost factors over the factor base except one large prime \( L \). In this case I check if I've seen any other pair \((y, Q(y))\) such that \( Q(y) \) factors over the factor base except for \( L \). If this is the case I combine the pairs and form a pair \((x*y, Q(x)*Q(y))\). Now \( Q(x)*Q(y) \) factor entirely over the factor base except for an \( L^2 \) term which I can remember is there when I try to calculate the square root of the products of the subset of \( Q(x) \). Since there may be many of these large-prime \( Q(x) \) only \( R \) pairs with the smallest associated \( L \) are kept in memory where \( R \) was selected to be 10,000,000.

**Performance**

To test the effectiveness and correctness of my implementation of the quadratic sieve I generated two probable primes \( p \) and \( q \) and fed \( p*q \) to my factoring algorithm. Below is a table of the performance of my quadratic sieve on different sizes of \( p*q \). Note that the quadratic sieve does not change
<table>
<thead>
<tr>
<th>Size of p*q (bits)</th>
<th>Time to factor</th>
<th>Factor base size</th>
<th>(x, Q(x)) pairs needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>2 hours 21 minutes</td>
<td>6454</td>
<td>6288</td>
</tr>
<tr>
<td>144</td>
<td>7 minutes 13 seconds</td>
<td>3650</td>
<td>3910</td>
</tr>
<tr>
<td>128</td>
<td>5 minutes 16 seconds</td>
<td>2313</td>
<td>2114</td>
</tr>
<tr>
<td>112</td>
<td>39 seconds</td>
<td>1280</td>
<td>1096</td>
</tr>
<tr>
<td>96</td>
<td>33 seconds</td>
<td>368</td>
<td>350.7</td>
</tr>
</tbody>
</table>

**Known bugs**

Additionally there appear to be some bugs implementation bugs present when large semi-primes are tested. The symptoms that have been observed include a segmentation fault (may be fixed) and not finding non-trivial factors after finding a large number of linear dependencies (not fixed).

**Potential for improvement**

The sieving process is what takes by far the longest in the quadratic sieve. Finding a factor base and finding a linear dependence in the resulting matrix take negligible time compared to this. With that in mind the obvious place for improvement is the sieving process.

The main way that my sieving algorithm differs from the sieving algorithm described in [1] is that I do not fix a sieve interval and instead, starting at \( x=1+\text{floor}(\sqrt{n}) \), just keep testing if \( Q(x) \) factors over \( F \). There are two main advantages to this approach. The first
is that suitable \((x, Q(x))\) pairs are found pretty uniformly throughout the sieving process making it possible to finish early if a linear dependency is detected early. The second is that it doesn’t require an extreme amount of memory to store the values for \(Q(x)\) for the entire sieving interval.

The disadvantage appears to be a negative impact on runtime. Doing the sieve my way requires a heap to track which primes should be tested as factors of \(Q(x)\). With some profiling tools it was revealed that the heap operations play a major part in the runtime of my implementation. A second disadvantage is that this approach doesn’t lend itself well to pruning techniques to drop some elements of the sieve interval early.

The sieving section of my code is pretty much as basic as it gets and could use improvement the most. Some improvements that could be made include sieving several polynomials at the same time, selecting polynomials to increase the chance that \(Q(x)\) will factor completely over the factor base, and using logarithms to estimate if a number will completely factor without doing large division. Each of these improvements is mentioned in [2].

**Parallelization**

Due to the nature of the sieving process, the quadratic sieve algorithm lends itself nicely to parallelization. With \(m\) computers working on the sieving process it can be effectively cut down in time by exactly \(m\). This is done by just having each of the computers having a different starting point and just moving to the next available starting point when they would start overlapping something another computer has worked on. Whenever a computer finds a pair \((x, Q(x))\) it just sends it over the network to some master server that processes incoming data and looks for linear dependencies.

This is a practical solution since it requires very little communication. In total only the polynomials for each computer and each pair \((x, Q(x))\) (of which there are not many) need to be sent over the network.

**Compiling and Running**

In order to compile the project you will need the bigint.h and factor.cpp files in the same folder.

Compilation is known to work on gcc 3.4 but should work on other versions as well. Additionally the program should compile and run in most environments. The following command will compile the code.

```
g++ -O3 factor.cpp -o factor.exe
```
The program can then be run by calling

factor.exe [OPTIONS] [NUMBER]

OPTIONS may include the flags -v, -t, -r, or -R. -v will run the program in verbose mode. -t will print timing information upon completion of the factorization. ‘-r bits’ can be used to specify factor.exe to generate a random bits bit number to factor. ‘-R bits’ can be used to specify factor.exe to generate a random bits bit semi-prime to factor.

**Sample Runs**

```
factor.exe 215162348567985064941538253710978589183
12304153318466298323
17486969074504742821

factor.exe 14880418504917757351101603048440338318867081
3671136386461879550263
4053354857583761703487

factor.exe -r 97
2
3
88743659
268526099520762159803

factor.exe -R 97
153622394604761
528471621105563
```

**Source Code**

The source code to the quadratic sieve implementation mentioned in this article can be found at http://umich.edu/~msgsss/factor/qs_src.zip. Additionally a copy of this report can be found at http://umich.edu/~msgsss/factor/qs_rep.pdf.
References

Appendix A - factor.cpp

/**
 * A simple driver program for factoring. Doesn't really have error checking.
 **/

#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
#include <queue>
#include <math.h>

#include "bigint.h"

using namespace std;

void print_usage() {
    cout << "./factor [OPTIONS] [NUMBER]" " << endl;
    cout << endl;
    cout << "Outputs a sorted list of the prime factors of NUMBER" " << endl;
    cout << endl;
    cout << ",[OPTIONS] may include:" " << endl;
    cout << " -r b - factor a random number with b bits. [NUMBER] will be ignored if present" " << endl;
    cout << " -R b - factor a random semiprime with b bits. [NUMBER] will be ignored" " << endl;
    cout << " -v - give verbose output" " << endl;
    cout << " -t - output timing information" " << endl;
}

int main(int argc, const char * argv[]) {
    srand(time(NULL));
    bigint n;

    int haven = 0;
    bool verbose = false;
    bool timing = false;
    for(int i = 1; i < argc; i++) {
        if(argv[i][0] == '-') {
            for(int j = 1; argv[i][j]; j++) {
                if(argv[i][j] == 'v') {
                    verbose = true;
                } else if(argv[i][j] == 't') {
                    timing = true;
                } else if(argv[i][j] == 'r') {
                    if(haven) {
                        cerr << "Two or more r/R flags cannot be present" " << endl;
                        return -1;
                    }
                    haven = 1;
                } else if(argv[i][j] == 'R') {
                    if(haven) {
                        cerr << "Two or more r/R flags cannot be present" " << endl;
                        return -1;
                    }
                    haven = 2;
                }
            }
        }
    }
}
if(haven > 0) {
    if(i + 1 == argc) {
        cerr << "Expected number after r/R flag" << endl;
        return -1;
    }
    int bits = atoi(argv[++i]);
    if(haven == 1) {
        n = bigint::random(bits);
    } else if(haven == 2) {
        n = bigint::random_prime(bits / 2) *
            bigint::random_prime((bits + 1) / 2);
    }
    haven = -1;
}

if(!haven) {
    n = bigint(argv[argc - 1]);
} else if(verbose) {
    cout << "Factoring " << n << endl;
}

clock_t start_time = clock();
vector<bigint> f = n.factor(verbose);
if(timing) {
    cout << "Factoring took " << 1.0 * (clock() - start_time) / CLOCKS_PER_SEC
        " seconds" << endl;
}

for(int i = 0; i < f.size(); i++) {
    cout << f[i] << endl;
}
return 0;
Appendix B - bigint.h

/**
 * Description: Class to be used for large integer arithmetic. The class
 * contains operators for doing simple arithmetic (+, -, *, /, %),
 * comparisons (<, >, <=, >=, ==, !=), bitwise operations
 * (&, |, ^, <<, >>), prefix/postfix increments/decrements
 * (++,-), stream operators for reading in and writing to a
 * stream, as well as many other operations that can be performed
 * on integers.
 * Author: Mark Gordon
 * Date: November 30th, 2008
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser 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 Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 **/

#include <iostream>
#include <algorithm>
#include <vector>
#include <set>
#include <queue>
#include <math.h>

using namespace std;

// The number of bits to use per digit in the integer representation.
static const int BASE_BITS = 30;

// The base of the representation of the integer.
static const int BASE = 1 << BASE_BITS;

class bigint {
public:
    bigint() : s(false) {} // Initializes integer to 0.
    bigint(const bigint &); // Copies the parameter.
    bigint(const string &); // Initializes to the value given in the string.
    bigint(int); // Initializes to the passed int.
    bigint & operator=(const bigint &);

    // a.compare(b) returns -1 if a < b, 0 if a == b, and 1 if a > b.
    int compare(const bigint &;)

    // Basic comparision operators.
    bool operator==(const bigint & x) const { return compare(x) == 0; }
    bool operator!=(const bigint & x) const { return compare(x) != 0; }
    bool operator<(const bigint & x) const { return compare(x) < 0; }
    bool operator<=(const bigint & x) const { return compare(x) <= 0; }
    bool operator>(const bigint & x) const { return compare(x) > 0; }
    bool operator>=(const bigint & x) const { return compare(x) >= 0; }
}
bool operator>=(const bigint & x) const { return compare(x) >= 0; }

// Takes the absolute value of the integer.
bigint abs() { bigint r = *this; r.s = false; return r; }
// Negates the integer.
bigint & negate() { if(!d.empty()) s = !s; return *this; }
// Returns *this negated.
bigint operator-() const { bigint cpy = *this; return cpy.negate(); }

// Basic scalar arithmetic. Note that these operators work faster than the
// bigint equivalent and should be prefered where applicable.
bigint & operator+=(int);
bigint & operator-=(int);
bigint & operator*=(int);
bigint & operator/=(int);
bigint & operator%=(int x) { *this = bigint(*this % x); }

bigint operator+(int x) const { bigint r = *this; r += x; return r; }
bigint operator-(int x) const { bigint r = *this; r -= x; return r; }
bigint operator*(int x) const { bigint r = *this; r *= x; return r; }
bigint operator/,(int x) const { bigint r = *this; r /= x; return r; }

// Basic integer arithmetic.
bigint & operator+=(const bigint &);
bigint & operator-=(const bigint &);
bigint & operator*=(const bigint & x) { *this = *this * x; return *this; }
bigint & operator/=(const bigint & x) { *this = *this / x; return *this; }
bigint & operator%=(const bigint & x) { *this = *this % x; return *this; }

bigint operator+(const bigint & x) const { bigint r = *this; r += x; return r; }
bigint operator-(const bigint & x) const { bigint r = *this; r -= x; return r; }
bigint operator*(const bigint &) const;
bigint operator/(const bigint &) const;
bigint operator%(const bigint &) const;

// Basic binary arithmetic. All binary operators ignore the sign bit.
bigint & operator|=(const bigint &);
bigint & operator&=(const bigint &);
bigint & operator^=(const bigint &);
bigint operator|(const bigint & x) const { bigint r = *this; r |= x; return r; }
bigint operator&(const bigint & x) const { bigint r = *this; r &= x; return r; }
bigint operator^(const bigint & x) const { bigint r = *this; r ^= x; return r; }

bigint operator<<(int) const;
bigint operator>>(int) const;

// Postfix/Prefix incrementors and decrementors.
bigint & operator++() { return *this += 1; }
bigint & operator++(int) { bigint r = *this; *this += 1; return r; }
bigint & operator--() { return *this -= 1; }
bigint operator--(int) { bigint r = *this; *this -= 1; return r; }

// Allows for swapping two integers in constant time.
void swap(bigint & x) { std::swap(s, x.s); d.swap(x.d); }
// Converts the integer into a base radix string representation.
string to_string(int radix = 10) const;
// Returns the length of the integer in bits.
int bits() const { return d.empty() ? 0 : BASE_BITS * d.size() -
  __builtin_clz(d.back()) + 32 - BASE_BITS; }

// Returns true if the xth bit is set.
bool get_bit(int x) const { if(x >= d.size() * BASE_BITS) return 0; return d[x / BASE_BITS] & 1 << (x % BASE_BITS); }

// Sets the xth bit to v.
void set_bit(int x, bool v) { if(x >= d.size() * BASE_BITS) d.resize(x / BASE_BITS + 1); if(v) d[x / BASE_BITS] |= 1 << (x % BASE_BITS); else { d[x / BASE_BITS] &= ~(1 << (x % BASE_BITS)); purge(); } }

// Converts the integer to an int.
int to_int() const { int ret = 0; for(int i = (int)d.size() - 1; i >= 0; i--) ret = ret * BASE + d[i]; return ret; }

// Converts the integer to a long long.
long long to_long_long() const { long long ret = 0; for(int i = (int)d.size() - 1; i >= 0; i--) ret = ret * BASE + d[i]; return ret; }

// Computes a random number with the passed number of bits.
static bigint random(int);

// Computes a random probable prime number with the passed number of bits.
static bigint random_prime(int);

// Computes the floor of the square root of *this.
bigint sqrt() const;

// Computes the greatest common divisor of *this and x.
bigint gcd(const bigint & x) const;

// Computes *this^e modulo mod.
bigint mod_exp(const bigint & e, const bigint & mod) const;

// Computes x such that *this * x = 1 modulo mod.
bigint mod_inv(const bigint & mod) const;

// Returns true if *this is almost certainly prime.
bool probably_prime() const;

// Returns 1 if there exists an x such that x^2=*this modulo p.
// Returns 0 if x = 0 modulo p.
// Returns -1 otherwise.
int legendre(const bigint & p) const;

// Returns x such that x^2=*this.  The behavior of this function is not
// defined if legendre(p) is not 1.  This uses the Shanks-Tonelli algorithm.
bigint mod_square_root(const bigint & p) const;

// Returns a sorted list of the prime factors of *this.  This uses a
// combination of trial division, Pollard's Rho algorithm and the quadratic
// sieve.
vector<bigint> factor(bool verbose = false) const;

private:
// Sign bit.  s = true means the integer is negative.
bool s;

// A list of the digits of the integer.  Less significant digits have lower
// indices.
vector<int> d;

// Helper function to remove undesired trailing 0s.
void purge();
};

// Stream operator for writing a big integer to a stream.
ostream & operator <<(ostream &, const bigint &);

// Stream operator for reading a big integer in from a stream.
istream & operator>>(istream &, bigint &)

bigint::bigint(const bigint & x) {
    s = x.s;
    d = x.d;
}

bigint::bigint(const string & x) {
    s = false;
    for(int i = x[0] == '-' ? 0 : 1; i < x.size(); i++) {
        *this *= 10;
        *this += x[i] - '0';
    }
    s = x[0] == '-' ? 1 : 0;
}

bigint::bigint(int x) {
    s = 0;
    *this += x;
}

bigint & bigint::operator=(const bigint & x) {
    s = x.s;
    d = x.d;
}

int bigint::compare(const bigint & x) const {
    if(s != x.s) {
        return s ? -1 : 1;
    }
    if(d.size() < x.d.size()) {
        return s ? 1 : -1;
    } else if(x.d.size() < d.size()) {
        return s ? -1 : 1;
    }
    for(int i = (int)d.size() - 1; i >= 0; i--) {
        if(d[i] < x.d[i]) {
            return s ? 1 : -1;
        } else if(x.d[i] < d[i]) {
            return s ? -1 : 1;
        }
    }
    return 0;
}

void bigint::purge() {
    int sz;
    for(sz = d.size(); sz && d[sz - 1] == 0; sz--);
    d.resize(sz);
}

bigint & bigint::operator+=(int x) {
    if(x < 0 != s) {
        *this -= -x;
    } else {
        if(x < 0) x = -x;
        if(d.size() < 3) d.resize(2);
        d[0] += x & BASE - 1;
        d[1] += (d[0] >> BASE_BITS) + (x >> BASE_BITS);
        d[0] &= BASE - 1;
        d[1] &= BASE - 1;
        int c = d[1] >> BASE_BITS;
        d[1] &= BASE - 1;
        for(int i = 2; c && i < d.size(); i++) {
            d[i] += x >> BASE * (BASE - 1);
            d[i] &= BASE - 1;
        }
    }
}
d[i] += c;
c = d[i] >> BASE_BITS;
d[i] &= BASE - 1;
}
if(c) {
d.push_back(c);
}
purge();
return *this;
}

bigint & bigint::operator-=(int x) {
if(x < 0 != s) {
  *this += -x;
} else {
  if(x < 0) x = -x;
  if(d.size() < 2) d.resize(2);
  d[0] -= x & BASE - 1;
  d[1] -= x >> BASE_BITS;
  if(d[0] < 0) {
    d[0] += BASE;
    d[1]--;
  }
  for(int i = 1; i + 1 < d.size() && d[i] < 0; i++) {
    d[i] += BASE;
    d[i + 1]--;
  }
  if(d.back() < 0) {
    bool pull = false;
    for(int i = 0; i + 1 < d.size(); i++) {
      if(pull) {
        d[i] = BASE - d[i] - 1;
      } else if(d[i]) {
        d[i] = BASE - d[i];
        pull = true;
      }
    }
    d.back() = -d.back() - 1;
    s = !s;
  }
  purge();
}
return *this;
}

bigint & bigint::operator*=(int x) {
if(x == 0) {
  s = false;
  d.resize(0);
  return *this;
}
int c = 0;
for(int i = 0; i < d.size(); i++) {
  long long v = 1LL * x * d[i] + c;
  d[i] = v & BASE - 1;
  c = v >> BASE_BITS;
}
if(c) d.push_back(c);
return *this;
}

bigint & bigint::operator/=(int x) {
long long c = 0;
for(int i = (int)d.size() - 1; i >= 0; i--) {
    long long nc = (d[i] + c) % x;
    d[i] = (d[i] + c) / x;
    c = nc * BASE;
}  
purge();
return *this;
}

int bigint::operator%(int x) const {
    long long m = 1;
    long long res = 0;
    for(int i = 0; i < d.size(); i++) {
        res = (res + d[i] * m) % x;
        m = (m * BASE) % x;
    }
    return res;
}

bigint & bigint::operator+=(const bigint & x) {
    if(x.s != s) {
        const_cast<bigint &>(x).negate();
        *this -= x;
        const_cast<bigint &>(x).negate();
    } else {
        if(d.size() < x.d.size()) {
            d.resize(x.d.size());
        }
        int c = 0;
        for(int i = 0; i < d.size(); i++) {
            d[i] += c + (i < x.d.size() ? x.d[i] : 0);
            c = d[i] >> BASE_BITS;
            d[i] &= BASE - 1;
        }
        if(c) {
            d.push_back(c);
        }
        return *this;
    }
}

bigint & bigint::operator-=(const bigint & x) {
    if(x.s != s) {
        const_cast<bigint &>(x).negate();
        *this -= x;
        const_cast<bigint &>(x).negate();
    } else {
        if(d.size() < x.d.size()) {
            d.resize(x.d.size());
        }
        for(int i = 0; i < d.size(); i++) {
            d[i] -= i < x.d.size() ? x.d[i] : 0;
            if(i + 1 < d.size() && d[i] < 0) {
                d[i] += BASE;
                d[i + 1]--;
            }
        }
        if(d.back() < 0) {
            bool pull = false;
            for(int i = 0; i + 1 < d.size(); i++) {
                if(pull) {
                    d[i] = BASE - d[i] - 1;
                }
            }
        }
    }
}

bigint & bigint::operator*(const bigint & x) {
    if(x.s != s) {
        const_cast<bigint &>(x).negate();
        *this *= x;
        const_cast<bigint &>(x).negate();
    } else {
        int c = 0;
        for(int i = 0; i < d.size(); i++) {
            d[i] *= x;
            if(d[i] >= BASE) {
                d[i] -= BASE;
            }
        }
    }
    return *this;
}

bigint & bigint::operator/=(const bigint & x) {
    if(x.s != s) {
        const_cast<bigint &>(x).negate();
        *this /= x;
        const_cast<bigint &>(x).negate();
    } else {
        int c = 0;
        for(int i = 0; i < d.size(); i++) {
            d[i] /= x;
            if(d[i] < 0) {
                d[i] += BASE;
            }
        }
    }
    return *this;
}
else if(d[i]) {
    d[i] = BASE - d[i];
    pull = true;
}
}
d.back() = -d.back() - 1;
s = !s;
}
purge();
}
return *this;
}
bigint bigint::operator*(const bigint & x) const {
    bigint ret;
    if(d.empty() || x.d.empty()) {
        return ret;
    }
    ret.s = s != x.s;
    ret.d = vector<int>(d.size() + x.d.size(), 0);
    for(int i = 0; i + 1 < d.size() + x.d.size(); i++) {
        for(int j = max(0, i - (int)x.d.size() + 1); j <= i && j < d.size(); j++) {
            long long v = 1LL * d[j] * x.d[i - j];
            ret.d[i] += v & BASE - 1;
            ret.d[i + 1] += (v >> BASE_BITS) + (ret.d[i] >> BASE_BITS);
            ret.d[i] &= BASE - 1;
            for(int k = i + 1; ret.d[k] > BASE; k++) {
                if(k + 1 == ret.d.size()) {
                    ret.d.push_back(0);
                }
                ret.d[k + 1] += ret.d[k] >> BASE_BITS;
                ret.d[k] &= BASE - 1;
            }
        }
    }
    ret.purge();
    return ret;
}
bigint bigint::operator/(const bigint & x) const {
    bigint ret = 0;
    bigint cpy = *this;
    cpy.s = x.s;
    while(true) {
        int lo = -1;
        int hi = cpy.bits();
        while(lo < hi) {
            int mid = (lo + hi + 1) / 2;
            if((x << mid) <= cpy) {
                lo = mid;
            } else {
                hi = mid - 1;
            }
        }
        if(lo == -1) {
            break;
        }
        cpy -= x << lo;
        ret += bigint(1) << lo;
    }
    if(!ret.d.empty()) {
        ret.s = s != x.s;
    }
}
bigint bigint::operator%(const bigint & x) const {
    bigint cpy = *this;
    cpy.s = x.s;
    while(true) {
        int lo = -1;
        int hi = cpy.bits();
        while(lo < hi) {
            int mid = (lo + hi + 1) / 2;
            if((x << mid) <= cpy) {
                lo = mid;
            } else {
                hi = mid - 1;
            }
        }
        if(lo == -1) {
            return cpy;
        }
        cpy -= x << lo;
    }
    return bigint();
}

bigint & bigint::operator|=(const bigint & x) {
    if(d.size() < x.d.size()) {
        d.resize(x.d.size());
    }
    for(int i = 0; i < x.d.size(); i++) {
        d[i] |= x.d[i];
    }
    return *this;
}

bigint & bigint::operator&=(const bigint & x) {
    if(x.d.size() < d.size()) {
        d.resize(x.d.size());
    }
    for(int i = 0; i < x.d.size(); i++) {
        d[i] &= x.d[i];
    }
    purge();
    return *this;
}

bigint & bigint::operator^=(const bigint & x) {
    if(d.size() < x.d.size()) {
        d.resize(x.d.size());
    }
    for(int i = 0; i < x.d.size(); i++) {
        d[i] ^= x.d[i];
    }
    purge();
    return *this;
}

bigint bigint::operator<<(int x) const {
    bigint ret;
    ret.s = s;
    ret.d = vector<int>(d.size() + (x - 1) / BASE_BITS + 1, 0);
    int y = x % BASE_BITS;
for(int i = 0; i < d.size(); i++) {
    if(y) {
        int p1 = (d[i] & (1 << (BASE_BITS - y)) - 1) << y;
        int p2 = d[i] >> (BASE_BITS - y);
        ret.d[i + x / BASE_BITS] |= p1;
        ret.d[i + x / BASE_BITS + 1] |= p2;
    } else {
        ret.d[i + x / BASE_BITS] = d[i];
    }
}
ret.purge();
return ret;
}

bigint bigint::operator>>(int x) const {
    bigint ret;
    ret.s = s;
    ret.d = vector<int>(d.size() + (x - 1) / BASE_BITS + 1, 0);
    int y = x % BASE_BITS;
    for(int i = 0; i < d.size(); i++) {
        if(y) {
            int p1 = (d[i] & (1 << y) - 1) << (BASE_BITS - y);
            int p2 = d[i] >> y;
            if(x / BASE_BITS <= i) ret.d[i - x / BASE_BITS] |= p2;
            if(x / BASE_BITS < i) ret.d[i - x / BASE_BITS - 1] |= p1;
        } else if(x / BASE_BITS <= i) {
            ret.d[i - x / BASE_BITS] = d[i];
        }
    }
    ret.purge();
    return ret;
}

string bigint::to_string(int radix) const {
    if(d.empty()) {
        return "0";
    }
    string ret;
    bigint cpy = *this;
    while(cpy.d.size()) {
        int val = cpy % radix;
        cpy /= radix;
        if(val >= 10) {
            ret += 'A' + (val - 10);
        } else {
            ret += '0' + val;
        }
    }
    if(s) {
        ret += '-';
    }
    reverse(ret.begin(), ret.end());
    return ret;
}

bigint bigint::random(int bits) {
    bigint ret;
    for(int i = 0; i < bits; i++) {
        ret.set_bit(i, 1.0 * rand() / RAND_MAX < 0.5);
    }
    return ret;
}
bigint bigint::random_prime(int bits) {
    bigint ret;
    do {
        ret = random(bits);
        ret.set_bit(bits - 1, true);
        ret.set_bit(0, true);
    } while(!ret.probably_prime());
    return ret;
}

bigint bigint::sqrt() const {
    bigint ret;
    for(int i = 1 + bits() / 2; i >= 0; i--) {
        ret.set_bit(i, true);
        if(ret * ret > *this) {
            ret.set_bit(i, false);
        }
    }
    return ret;
}

bigint bigint::gcd(const bigint & x) const {
    bigint a = *this;
    bigint b = x;
    while(!a.d.empty()) {
        b %= a;
        a.swap(b);
    }
    return b;
}

bigint bigint::mod_exp(const bigint & e, const bigint & mod) const {
    bigint ret = 1;
    for(int i = e.bits(); i >= 0; i--) {
        ret *= ret;
        ret %= mod;
        if(e.get_bit(i)) {
            ret *= *this;
            ret %= mod;
        }
    }
    return ret;
}

bigint bigint::mod_inv(const bigint & mod) const {
    bigint a = *this;
    bigint b = mod;
    bigint A = 1, B = 0;
    while(a != 0) {
        bigint m = b / a;
        B += mod - m * A % mod;
        B %= mod;
        b %= a;
        a.swap(b); A.swap(B);
    }
    return B;
}

bool bigint::probably_prime() const {
    if(*this < 2) {
        return false;
    } else if(*this < 100) {
        for(int i = 2; i * i <= 100 && *this > i; i++) {
            if(ret.probably_prime()) {
if(*this % i == 0) {
    return false;
}
return true;
}

int s;
bigint d = *this - 1;
for(s = 0; !d.get_bit(s); s++);
d = d >> s;

int n = bits();
for(int k = 0; k < 20; k++) {
    int a = rand() & 0x7FFFFFFF;
    if(*this <= a) a %= to_int();
    if(a < 2) a = 2;
    bigint x = bigint(a).mod_exp(d, *this);
    if(x == 1 || x == *this - 1) {
        continue;
    }
    for(int i = 0; i < s; i++) {
        x *= x;
        x %= *this;
        if(x == *this - 1) {
            return true;
        }
    }
    return false;
}
return true;

// Uses the simple formula for calculating legendre numbers.
bigint::legendre(const bigint & p) const {
    bigint res = mod_exp((p - 1) / 2, p);
    if(res == 1) {
        return 1;
    } else if(res == p - 1) {
        return -1;
    } else {
        return 0;
    }
}

// Uses Shanks-Tonelli algoithm
bigint bigint::mod_square_root(const bigint & p) const {
    if(p == 2) {
        return get_bit(0) ? 1 : 0;
    } else if(p % 4 == 3) {
        return mod_exp((p + 1) / 4, p);
    }
    bigint Q = p - 1;
    int S = 0;
    while(Q % 2 == 0) {
        Q /= 2;
        S++;
    }
    bigint W;
for (W = 2; ; W++)
    if (W.legendre(p) == -1)
        break;

bigint R = mod_exp((Q + 1) / 2, p);
bigint V = W.mod_exp(Q, p);
bigint ninv = mod_inv(p);

while (true) {
    bigint val = R * R % p;
    val *= ninv; val %= p;

    int i;
    for (i = 0; val != 1; i++) {
        val *= val; val %= p;
    }

    if (i == 0) {
        break;
    }

    bigint RR = V;
    for (int j = 1; j < S - i - 1; j++) {
        RR *= RR; RR %= p;
    }

    R *= RR; R %= p;
}

return R;
}

// Used to expose the sqrt(double) method that is otherwise hidden from within
// the bigint class.
static double sq_root(double x) { return sqrt(x); }

// A simple helper function for computing the symmetric difference between two
// sorted lists. This is used several times in the factor method.
template<class T>
static vector<T> list_xor(vector<T> & A, const vector<T> & B) {
    int a = 0;
    int b = 0;
    vector<T> ret;
    while (a < A.size() && b < B.size()) {
        if (A[a] == B[b]) {
            a++;
            b++;
        } else if (A[a] < B[b]) {
            ret.push_back(A[a++]);
        } else {
            ret.push_back(B[b++]);
        }
    }
    while (a < A.size()) {
        ret.push_back(A[a++]);
    }
    while (b < B.size()) {
        ret.push_back(B[b++]);
    }
    return ret;
}

// A simple helper function for heapifying a tree.
template<class T>
static void heapify(vector<T> & A, int x) {
    while(true) {
        int c1 = 2 * x + 1;
        int c2 = c1 + 1;
        if(c2 < A.size()) {
            if(A[c1] < A[c2]) {
                if(A[c1] < A[x]) {
                    swap(A[c1], A[x]);
                    x = c1;
                    continue;
                }
            } else {
                if(A[c2] < A[x]) {
                    swap(A[c2], A[x]);
                    x = c2;
                    continue;
                }
            }
        } else if(c1 < A.size() && A[c1] < A[x]) {
            swap(A[c1], A[x]);
        }
        break;
    }
}

vector<bigint> bigint::factor(bool verbose) const {
    static const int TRIVIAL_DIVISION = 10000;
    static const int PRIME_SIEVE = 1000000;
    static const int POLLARD_RHO_ITERATIONS = 100;
    static const int DOUBLE_LARGE_PRIME_SET_SIZE = 1000000;

    bigint n = *this;
    if(verbose) {
        cout << "Starting trial division" << endl;
    }

    // Search for small prime factors using trial division.
    vector<bigint> ret;
    int div_bound = TRIVIAL_DIVISION;
    if(n.sqrt() < div_bound) {
        div_bound = n.sqrt().to_int();
    }
    for(int i = 2; i <= div_bound; i++) {
        while(n % i == 0) {
            n /= i;
            ret.push_back(i);
        }
    }
    if(n == 1 || n <= bigint(div_bound) * div_bound) {
        if(n != 1) {
            ret.push_back(n);
        }
        return ret;
    }

    // Check if we are probably wasting our time.
    if(n.probably_prime()) {
        ret.push_back(n);
        return ret;
    }

    if(verbose) {
        cout << "Starting Pollard Rho" << endl;
        for(int i = 0; i < POLLARD_RHO_ITERATIONS; i++) {
            bigint a = n;
            bigint b = n;
            bigint m = 2;
            bigint c = n.sqrt().to_int();
            bigint s = bigint(1);
            bigint t = bigint(1);
            while(s < c) {
                a = (a * a + a) % n;
                b = (b * b + b) % n;
                if(a == b) {
                    // a and b are equal, try to find a cycle
                    for(int j = 0; j < POLLARD_RHO_ITERATIONS; j++) {
                        a = (a * a + a) % n;
                        b = (b * b + b) % n;
                        if(a == b) {
                            ret.push_back(a);
                            return ret;
                        }
                    }
                } else {
                    s = s + 1;
                }
            }
        }
    }
}
cout << "Finished trial division, trying Pollard's Rho algorithm" << endl;
}

// Try Pollard's Rho algorithm for a little bit.
for(int iter = 0; iter < POLLARD_RHO_ITERATIONS; iter += POLLARD_RHO_ITERATIONS / 100) {
  bigint c = random(n.bits() + 4) % n;
  bigint x = 2;
  bigint y = 2;
  bigint g = 1;
  for( ; iter < POLLARD_RHO_ITERATIONS && g == 1; iter++) {
    x *= x; x += c; x %= n;
    y *= y; y += c; y %= n;
    y *= y; y += c; y %= n;
    g = (x - y).abs().gcd(n);
  }
  if(g != 1 && g != n) {
    if(verbose) {
      cout << "Pollard's Rho algorithm found non-trivial factor: " << g << endl;
    }
    // Divide and recursively factor each half and merge the lists.
    vector<bigint> fa = g.factor(verbose);
    vector<bigint> fb = (n / g).factor(verbose);
    for(int i = 0; i < fa.size(); i++) {
      ret.push_back(fa[i]);
    }
    for(int i = 0; i < fb.size(); i++) {
      ret.push_back(fb[i]);
    }
    sort(ret.begin(), ret.end());
    return ret;
  }
}

// Calculate how large the factor base should be. This formula comes from
// the paper found at http://www.math.uiuc.edu/~landquis/quadsieve.pdf.
int fsz = (int)pow(exp(sq_root(n.bits() * log(2) * log(n.bits() * log(2)))),
               sq_root(2) / 4) * 2;

// Perform the Sieve of Eratosthenes to get a list of small primes to use
// as the factor base. This only needs to be done once. If large primes are
// required the probably_prime method will be used instead.
static vector<bool> is_prime;
if(is_prime.empty()) {
  is_prime = vector<bool>(PRIME_SIEVE, true);
  for(int i = 2; i * i < PRIME_SIEVE; i++) {
    for(int j = i * i; is_prime[i] && j < PRIME_SIEVE; j += i) {
      is_prime[j] = false;
    }
  }
}

// Calculate the factor base. A factor base consists of primes p such that
// n has a quadratic residue modulo p.
vector<pair<int, bigint>> f_base;
for(int p = 2, f = 0; f < fsz; p++) {
  if(p < PRIME_SIEVE && !is_prime[p]) {
    continue;
  }
  bigint nm = n % p;
  if(nm.legendre(p) != 1) {
    continue;
  }
}
if(p >= PRIME_SIEVE && !bigint(p).probably_prime()) {
    continue;
}

f_base.push_back(make_pair(p, nm.mod_square_root(p)));
f++;

if(verbose) {
    cout << "Found " << f << " of " << fsz << " factors - " << p << endl;
}
}

// Initialize the rolling queue of known prime factors starting at rt.
// Don't include 2 and handle it as a special case.
bigint rt = n.sqrt() + 1;
if(n.get_bit(0) != rt.get_bit(0)) rt++;
vector<pair<long long, int> > q;
int bigp = f_base.back().first + 1;
vector<int> prime_count(bigp, 0);
for(int i = 1; i < f_base.size(); i++) {
    int p = f_base[i].first;
    bigint srt = f_base[i].second;
    int start_a = (srt + p - rt % p) % p;
    int start_b = (-srt + 2 * p - rt % p) % p;
    if(start_a % 2) start_a += p; start_a /= 2;
    if(start_b % 2) start_b += p; start_b /= 2;
    q.push_back(make_pair(start_a, p));
    prime_count[start_a]++;
    if(start_a != start_b) {
        prime_count[start_b]++;
        q.push_back(make_pair(start_b, p));
    }
}
for(int i = q.size() - 1; i >= 0; i--) {
    heapify(q, i);
}

if(verbose) {
    cout << "Factor base computed, beginning to sieve" << endl;
}

// Tracks pairs (x, y) such that y = x^2 - n and y factors over the factor base.
vector<pair<bigint, bigint> > field;
// mat[i].first is a list of columns that have a 1 in them for the ith row.
// mat[i].second is a list of what linear combination of original rows is
// represented in mat[i].first.
vector<pair<vector<int>, vector<int> > > mat;
// owner[i] tracks which row, if any, should be the only row to have 1 in the
// ith column.
vector<int> owner(fsz, -1);

// set for tracking double large primes.
set<pair<int, long long> > dlp;

// Keep searching for x^2 - n that factor completely over the factor base.
for(long long i = 0; ; i++, rt += 2) {
    // If there isn't a single factor here don't even bother.
    if(q[0].first != i) {
        continue;
    }

    // owner[i] tracks which row, if any, should be the only row to have 1 in the
    // ith column.
    vector<int> owner(fsz, -1);

    // set for tracking double large primes.
    set<pair<int, long long> > dlp;

    // Keep searching for x^2 - n that factor completely over the factor base.
    for(long long i = 0; ; i++, rt += 2) {
        // If there isn't a single factor here don't even bother.
        if(q[0].first != i) {
            continue;
        }

        // owner[i] tracks which row, if any, should be the only row to have 1 in the
        // ith column.
        vector<int> owner(fsz, -1);

        // set for tracking double large primes.
        set<pair<int, long long> > dlp;

        // Keep searching for x^2 - n that factor completely over the factor base.
        for(long long i = 0; ; i++, rt += 2) {
            // If there isn't a single factor here don't even bother.
            if(q[0].first != i) {
                continue;
            }

            // owner[i] tracks which row, if any, should be the only row to have 1 in the
            // ith column.
            vector<int> owner(fsz, -1);

            // set for tracking double large primes.
            set<pair<int, long long> > dlp;

            // Keep searching for x^2 - n that factor completely over the factor base.
            for(long long i = 0; ; i++, rt += 2) {
                // If there isn't a single factor here don't even bother.
                if(q[0].first != i) {
                    continue;
                }

                // owner[i] tracks which row, if any, should be the only row to have 1 in the
                // ith column.
                vector<int> owner(fsz, -1);

                // set for tracking double large primes.
                set<pair<int, long long> > dlp;

                // Keep searching for x^2 - n that factor completely over the factor base.
                for(long long i = 0; ; i++, rt += 2) {
                    // If there isn't a single factor here don't even bother.
                    if(q[0].first != i) {
                        continue;
                    }

                    // owner[i] tracks which row, if any, should be the only row to have 1 in the
                    // ith column.
                    vector<int> owner(fsz, -1);

                    // set for tracking double large primes.
                    set<pair<int, long long> > dlp;

                    // Keep searching for x^2 - n that factor completely over the factor base.
                    for(long long i = 0; ; i++, rt += 2) {
                        // If there isn't a single factor here don't even bother.
                        if(q[0].first != i) {
                            continue;
                        }

                        // owner[i] tracks which row, if any, should be the only row to have 1 in the
                        // ith column.
                        vector<int> owner(fsz, -1);

                        // set for tracking double large primes.
                        set<pair<int, long long> > dlp;

                        // Keep searching for x^2 - n that factor completely over the factor base.
                        for(long long i = 0; ; i++, rt += 2) {
                            // If there isn't a single factor here don't even bother.
                            if(q[0].first != i) {
                                continue;
                            }

                            // owner[i] tracks which row, if any, should be the only row to have 1 in the
                            // ith column.
                            vector<int> owner(fsz, -1);

                            // set for tracking double large primes.
                            set<pair<int, long long> > dlp;

                            // Keep searching for x^2 - n that factor completely over the factor base.
                            for(long long i = 0; ; i++, rt += 2) {
                                // If there isn't a single factor here don't even bother.
                                if(q[0].first != i) {
                                    continue;
                                }

                                // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                // ith column.
                                vector<int> owner(fsz, -1);

                                // set for tracking double large primes.
                                set<pair<int, long long> > dlp;

                                // Keep searching for x^2 - n that factor completely over the factor base.
                                for(long long i = 0; ; i++, rt += 2) {
                                    // If there isn't a single factor here don't even bother.
                                    if(q[0].first != i) {
                                        continue;
                                    }

                                    // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                    // ith column.
                                    vector<int> owner(fsz, -1);

                                    // set for tracking double large primes.
                                    set<pair<int, long long> > dlp;

                                    // Keep searching for x^2 - n that factor completely over the factor base.
                                    for(long long i = 0; ; i++, rt += 2) {
                                        // If there isn't a single factor here don't even bother.
                                        if(q[0].first != i) {
                                            continue;
                                        }

                                        // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                        // ith column.
                                        vector<int> owner(fsz, -1);

                                        // set for tracking double large primes.
                                        set<pair<int, long long> > dlp;

                                        // Keep searching for x^2 - n that factor completely over the factor base.
                                        for(long long i = 0; ; i++, rt += 2) {
                                            // If there isn't a single factor here don't even bother.
                                            if(q[0].first != i) {
                                                continue;
                                            }

                                            // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                            // ith column.
                                            vector<int> owner(fsz, -1);

                                            // set for tracking double large primes.
                                            set<pair<int, long long> > dlp;

                                            // Keep searching for x^2 - n that factor completely over the factor base.
                                            for(long long i = 0; ; i++, rt += 2) {
                                                // If there isn't a single factor here don't even bother.
                                                if(q[0].first != i) {
                                                    continue;
                                                }

                                                // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                                // ith column.
                                                vector<int> owner(fsz, -1);

                                                // set for tracking double large primes.
                                                set<pair<int, long long> > dlp;

                                                // Keep searching for x^2 - n that factor completely over the factor base.
                                                for(long long i = 0; ; i++, rt += 2) {
                                                    // If there isn't a single factor here don't even bother.
                                                    if(q[0].first != i) {
                                                        continue;
                                                    }

                                                    // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                                    // ith column.
                                                    vector<int> owner(fsz, -1);

                                                    // set for tracking double large primes.
                                                    set<pair<int, long long> > dlp;

                                                    // Keep searching for x^2 - n that factor completely over the factor base.
                                                    for(long long i = 0; ; i++, rt += 2) {
                                                        // If there isn't a single factor here don't even bother.
                                                        if(q[0].first != i) {
                                                            continue;
                                                        }

                                                        // owner[i] tracks which row, if any, should be the only row to have 1 in the
                                                        // ith column.
                                                        vector<int> owner(fsz, -1);

                                                        // set for tracking double large primes.
                                                        set<pair<int, long long> > dlp;

                                                        // Keep searching for x^2 - n that factor completely over the factor base.
// Compute Q(x) and try to factor it over the factor base.
bigint v = rt * rt - n;

int div2;
for(div2 = 1; !v.get_bit(div2); div2++);
v = v >> div2;

int maxdiv = 0;
while(q[0].first == i) {
    // Divide out the largest power of p from v.
    int p = q[0].second;
    // This heuristic seems to do pretty well in cutting down
    // checks on v that aren't likely to be smooth.
    if(prime_count[i % bigp] > 7) {
        int div = 1;
        v /= p;
        while(v % p == 0) {
            v /= p;
            div++;
        }
        maxdiv = max(maxdiv, div);
    }
    // Erase the prime from the queue and put it back in the queue p positions
    // later.
    prime_count[(i + p) % bigp]++;
    q[0].first += p;
    heapify(q, 0);
    }

bool added = false;
if(v <= 0x7FFFFFFF) {
    int iv = v.to_int();
    if(iv != 1) {
        if(dlp.size() < DOUBLE_LARGE_PRIME_SET_SIZE || dlp.rbegin()->first < iv) {
            typedef(dlp.begin()) it = dlp.lower_bound(make_pair(iv, 0));
            if(it != dlp.end() && it->first == iv) {
                // We found two factors with the same large prime!
                if(verbose) {
                    cout << "Found double prime " << iv << endl;
                }
                added = true;
                bigint ort = rt - 2 * (i - it->second);
                field.push_back(make_pair(rt * ort, (rt * rt - n) * (ort * ort - n)));
                f_base.push_back(make_pair(iv, -1));
                dlp.erase(it);
            } else {
                dlp.insert(make_pair(iv, i));
                if(dlp.size() > DOUBLE_LARGE_PRIME_SET_SIZE) {
                    dlp.erase(--dlp.end());
                }
            }
        } else if(iv == 1) {
            // Lucky day, v factored completely over the factor base.
            added = true;
            field.push_back(make_pair(rt, rt * rt - n));
        }
    }
}
if(added) {
}
if(Verbose) {
    cout << i << ": " << field.size() << " of " << fsz + 1 << " with " << prime_count[i % bigp] << " primes and maxdiv " << maxdiv << endl;
}

// Create a row for this solution.
vector<int> v_primes;
bigint v = field.back().second;
for(int j = 0; j < fsz; j++) {
    int cnt = 0;
    while(v % f_base[j].first == 0) {
        v /= f_base[j].first;
        cnt++;
    }
    if(cnt % 2) {
        v_primes.push_back(j);
    }
}
mat.push_back(make_pair(v_primes, vector<int>(1, field.size() - 1)));

// Cancel columns that are already owned.
for(int j = 0; j < v_primes.size(); j++) {
    int k = owner[v_primes[j]];
    if(k != -1) {
        mat.back().first = list_xor(mat.back().first, mat[k].first);
        mat.back().second = list_xor(mat.back().second, mat[k].second);
    }
}

if(!mat.back().first.empty()) {
    // Assign a column for this row to own.
    int id = mat.back().first[0];
    owner[id] = mat.size() - 1;
    for(int j = 0; j + 1 < mat.size(); j++) {
        if(binary_search(mat[j].first.begin(), mat[j].first.end(), id)) {
            mat[j].first = list_xor(mat.back().first, mat[j].first);
            mat[j].second = list_xor(mat.back().second, mat[j].second);
        }
    }
} else {
    // We have a linear dependence! Hoorah!
    if(Verbose) {
        cout << "Linear dependence detected" << endl;
    }

    // Calculate a and b such that a^2 = b^2 mod n.
    bigint a = 1;
    bigint b = 1;
    vector<int> & v = mat.back().second;
    vector<bool> parity(fsz, false);
    for(int k = 0; k < v.size(); k++) {
        a *= field[v[k]].first; a %= n;
        bigint val = field[v[k]].second;
        for(int s = 0; s < f_base.size(); s++) {
            while(val % f_base[s].first == 0) {
                val /= f_base[s].first;
                parity[s] = !parity[s];
                if(!parity[s]) {
                    b *= f_base[s].first; b %= n;
                }
            }
        }
    }
}
if(a < b) {
    a.swap(b);
}

if(a * a % n != b * b % n) {
    cout << "Computation error: squares not congruent" << endl;
}

// We now have (a + b)(a - b) = n. Calculate gcd(a + b, n) and
gcd(a - b, n) to try to find non trivial factor. This usually works.
for(bigint f = a - b; f <= a + b; f += b << 1) {
    bigint factor = f.gcd(n);
    if(factor != 1 && factor != n) {
        if(verbose) {
            cout << "Non-trivial factor calculated: " << factor << endl;
        }

        // Divide and recursively factor each half and merge the lists.
        vector<bigint> fa = factor.factor(verbose);
        vector<bigint> fb = (n / factor).factor(verbose);
        for(int i = 0; i < fa.size(); i++) {
            ret.push_back(fa[i]);
        }
        for(int i = 0; i < fb.size(); i++) {
            ret.push_back(fb[i]);
        }
        sort(ret.begin(), ret.end());
        return ret;
    }
}

prime_count[i % bigp] = 0;
return vector<bigint>();
}

ostream & operator <<(ostream & out, const bigint & x) {
    out << x.to_string();
    return out;
}

istream & operator >> (istream & in, bigint & x) {
    string s;
    in >> s;
    x = bigint(s);
    return in;
}