**PEX 1: Implementation Hints**

**Finding a square root of a large *n*:**

The Java code below illustrates how to take the square root of a BigInteger and includes an example of its usage. This code snip-it was found online at <http://faruk.akgul.org/blog/javas-missing-algorithm-biginteger-sqrt/> .

BigInteger sqrt(BigInteger n) {

BigInteger a = BigInteger.ONE;

BigInteger b = **new** BigInteger(n.shiftRight(5).add(**new** BigInteger("8")).toString());

**while**(b.compareTo(a) >= 0) {

BigInteger mid = **new** BigInteger(a.add(b).shiftRight(1).toString());

**if**(mid.multiply(mid).compareTo(n) > 0) b = mid.subtract(BigInteger.ONE);

**else** a = mid.add(BigInteger.ONE);

}

**return** a.subtract(BigInteger.ONE);

}

Usage:

BigInteger n = **new** BigInteger("34598734524213124593869456843967349872984172392043275489357439720358723");

BigInteger temp = **new** BigInteger(sqrt(n).toString());

System.out.println(temp.toString());

**A walk-through of Dixon’s algorithm.**

1. Get the number of factors in your factor base, *F*, from the user. You may assume a maximum upper bound of 25.
2. Begin by determining the *F* primes that make up your factor base (you can pre-define this list or read from a file).
3. Generate *F* “good” equations by picking random *x* values and testing if *x*2 (mod *n*) factors over your factor base. You can do this by iterating through the primes in your factor base, if the prime divides *n*, divide out all its powers and keep track of the exponent. Stop when you get 1 or you exhaust your factor base. This will determine the corresponding row of your coefficient matrix.
4. Once you have *F* good equations, you should also have an *F*x*F* coefficient matrix. Let’s call that *C*.
5. Compute *C* (mod 2). Call this new matrix *C’*.
6. Perform mod 2 Gaussian Elimination on *C’* and an *F*x*F* identity matrix *I* in parallel. Any zero rows in the transformed version of *C’* will indicate which equations can be combined by examining the corresponding rows of the transformed version of *I*. (Looking at the example will help clarify). A Java and Python Gaussian Elimination routine has been provided for you.
7. For each zero row in the transformed version of *C’*, combine the indicated equations in the corresponding row within the transformed *I*. Find the corresponding (*x,y*) pair. If *x ≠ ± y* (mod *n*), compute gcd(|*x-y*|, *n*) to give one factor of *n*. If the gcd is non-trival you’ve succeeded! Report your time.
8. If none of the zero rows in the reduced version of *C’* produce an (*x,y*) pair that yields a factor (or maybe there were no zero rows to begin with), go back to step c) and try again, generating *F* new equations. If you try this **three times without finding a successful factor**, report “Could not find factor.”

**EXAMPLE (explained by corresponding steps):**

Let’s assume the number we’re trying to factor is *n* = 55973, which we’re told is the product of 2 primes.

1. The user defines *F* = 8, which means we have a factor base of the first 8 primes.
2. So your factor base is comprised of the following primes {2, 3, 5, 7, 11, 13, 17, 19}.
3. Suppose after picking enough random *x*’s, we eventually obtain the following 8 “good” equations:

*x* *x*2 (mod *n)* 2 3 5 7 11 13 17 19

54654 4598 1 0 0 0 2 0 0 1

27237 44000 5 0 3 0 1 0 0 0

21552 24750 1 2 3 0 1 0 0 0

45957 16640 8 0 1 0 0 1 0 0

38316 39 0 1 0 0 0 1 0 0

682 17340 2 1 1 0 0 0 2 0

14393 2376 3 3 0 0 1 0 0 0

26942 13500 2 3 3 0 0 0 0 0

In other words, the program picked a random *x* = 54654, squared it mod *n* to give 4598, and factored that into 21\*112\*191 which used only factors in our factor base, so it was kept. The rest of the rows were constructed in a similar manner.

1. The 8x8 square matrix above is the coefficient matrix *C*.
2. *C’* is *C* (mod 2), which just replaces the even numbers with 0 and the odd numbers with 1. So *C’* is:

1 0 0 0 0 0 0 1

1 0 1 0 1 0 0 0

1 0 1 0 1 0 0 0

0 0 1 0 0 1 0 0

0 1 0 0 0 1 0 0

0 1 1 0 0 0 0 0

1 1 0 0 1 0 0 0

0 1 1 0 0 0 0 0

1. Passing the above matrix and an 8x8 identity matrix, *I*, into a routine for mod 2 Gaussian elimination will produce transformed versions of *I* and *C’* that look like this:

transformed *I* = transformed *C’* =

1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1

0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0

1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1

1 0 1 1 0 0 0 0 0 0 0 0 1 1 0 1

1 0 1 0 1 1 0 0 0 0 0 0 1 1 0 1

0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 🡨 zero row

0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 🡨 zero row

0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 🡨 zero row

1. The 6th, 7th and 8th rows of the transformed *C’* are zero (it’s a property of the linear algebra that if there are any zero rows, they’ll be on the bottom, so you can start at the bottom and work up). There are three possibilities of equations to combine and therefore three possible (x,y) pairs.

The bottom row tells us that the 6th and 8th equations can be combined, because the 8th row of the transformed *I* matrix has 1’s in the 6th and 8th columns. Combining equations (6) and (8) gives

*x* = 682\*26942 (mod *n*) = 15300, y = 22\*32\*52\*17 (mod *n*) = 15300. So x ≡ y (mod *n*) in this case, which means that (x,y) pair didn’t pan out. So we have to keep going.

The 7th row tells us that the 3rd, 6th and 7th equations can be combined. Combining equations (3), (6) and (7) gives *x* = 21552\*682\*14393 (mod *n*) = 1282, y = 23\*33\*52\*11\*17 (mod *n*) = 2286. These are neither identical nor opposites (mod *n*), so we’re good to go. Therefore, gcd( |1282-2286 |, *n*) = 251 must be a factor. The other factor is then 55973/251 = 223. Success!

1. If none of the rows in *C’* were all zeros, go back to step c) to find new “good” equations, and try again (up to 3 run-throughs).