TCSS543A – Autumn 2019
Programming Project (20+10+5 points)

Objective: implement basic functionality of elliptic curve cryptography in Java, and apply Pollard's rho algorithm to compute discrete logarithms on small elliptic curves.

Specification:

Consider an elliptic curve whose points (x, y) satisfy an equation of form $x^2 + y^2 = 1 + dx^2y^2$ for some given d, with all arithmetic operations performed on integers modulo a given prime number p (this is called an Edwards curve, which is the kind of curve adopted in RFC 7748). Let n denote the given number of points on that curve.

Given any two points $a_1 = (x_1, y_1)$ and $a_2 = (x_2, y_2)$ on that curve, their "product" is a third point on that curve, defined to be $a_3 = a_1 \cdot a_2 := (x_3, y_3) = \left(\frac{x_1y_2+y_1x_2}{1+dx_1x_2y_1y_2}, \frac{y_1y_2-x_1x_2}{1-dx_1x_2y_1y_2}\right)$. The "inverse" of a point $a = (x, y)$ is the point $a^{-1} := (-x, y)$, and the unit (neutral) element of point multiplication is the point $u := (0, 1)$.

It is crucial that all additions, subtractions, products, and inversions be performed modulo p (to get the remainder of a BigInteger variable `v` modulo a `BigInteger` variable `p`, invoke the method `v.mod(p)`, and to get the inverse of `v` modulo `p`, invoke the method `v.modInverse(p)` whenever needed). Notice that there is no actual division involved in the above formulas: the notation u/v means the product (modulo p) of u by the inverse of v (modulo p), which with the Java BigInteger class corresponds to `u.multiply(v.modInverse(p)).mod(p)`, *never* to `u.divide(v)`.

By extension, one can define an "exponentiation" operation on curve points: given a curve point a and an integer $0 \le m < p$, one can define a^m to be the result of multiplying a by itself m times using the above formula (with the provision that $a^0 := u$). The following algorithm computes a^m from a and m efficiently (*far* more efficiently, in fact, than by m repeated multiplications), assuming that curve points are implemented by a class `ECPoint`:

```
ECPoint Exp(ECPoint a, BigInteger m)
      ECPoint b = u; // NB: u is the point (0, 1)
      for (int i = m.bitLength() – 1; i >= 0; i--) {
            b = b.multiply(b); // i.e. b = b^2
            if (m.testBit(i)) {
                  b = b.multiply(a);
            }
      }
      return b;
}
```

Given two curve points a and b such that $b = a^m$, Pollard's rho algorithm recovers the value m, which is aptly called the discrete logarithm (base a) of b. It maintains a sextuple of variables $(\alpha_k, \beta_k, z_k, \alpha_{2k}, \beta_{2k}, z_{2k})$ for $k = 0, 1, 2, \ldots$ until it finds a sextuple where $z_k = z_{2k}$. The initialization sets $\alpha_0 = \beta_0 = 0$, $z_0 = (0, 1)$ (this point is the neutral element of point multiplication), and updates z_k to z_{k+1} (and also α_k to α_{k+1} and β_k to β_{k+1}) according to the following pseudocode rules, where $xcoord(z_k)$ stands for the x-coordinate of $z_k = (x, y)$:

```
      switch (xcoord(z_k) mod 3) {
      case 0:
            z_{k+1} ← b · z_k
            α_{k+1} ← α_k + 1
            β_{k+1} ← β_k
            break;
      case 1:
            z_{k+1} ← z_k · z_k
            α_{k+1} ← 2α_k
            β_{k+1} ← 2β_k
            break;
      case 2:
            z_{k+1} ← a · z_k
            α_{k+1} ← α_k
            β_{k+1} ← β_k + 1
            break;
      }
```

To update z_{2k}, α_{2k}, β_{2k}, the above procedure is applied twice with these variables in the role of the above ones, thus updating first z_{2k} to z_{2k+1} according to the value of $xcoord(z_{2k}) \bmod 3$, then updating z_{2k+1} to $z_{2k+2} = z_{2(k+1)}$ according to the value of $xcoord(z_{2k+1}) \bmod 3$ (and similarly for α_{2k} and β_{2k}).

When $z_k = z_{2k}$ for some k (which defines the number of computational steps necessary to obtain the discrete logarithm), the algorithm recovers $m = \frac{\beta_{2k}-\beta_k}{\alpha_k-\alpha_{2k}} \pmod n$. Notice that the arithmetic operations in this algorithm (namely, the subtractions, the inversion and the multiplication of the numerator by the inverse of the denominator) are all performed modulo the given number n of points on the curve. Also notice that it might happen that $\alpha_k - \alpha_{2k} = 0 \pmod n$, with probability about $1/n$, meaning that the rho method failed for the simple initialization of variables α_0, β_0, and z_0 above. Such occurrences should be reported by throwing an exception.

Implementation (5 points each item):

1. Implement a method `BigInteger[] mul(BigInteger[] a1, BigInteger[] a2, BigInteger d, BigInteger p)` that computes point `BigInteger[] a3` as specified by the above formula from the given points a_1 and a_2 (all points represented as `BigInteger` arrays containing exactly 2 elements each, namely, its x-coordinate and its y-coordinate) and given values of d and p.

2. Implement a method `BigInteger[] exp(BigInteger[] a, BigInteger m, BigInteger d, BigInteger p)` that computes the curve point $b = a^m$ given a point a represented as above, a BigInteger exponent m, and values of d and p.

3. Implement a method `BigInteger[] rho(BigInteger[] a, BigInteger[] b, BigInteger d, BigInteger p, BigInteger n)` that, given points a and b such that $b = a^m$ and the values of d, p, and n, recovers the exponent ("discrete logarithm") m modulo n and counts the number k of steps necessary for that computation. The result must be contained in a `BigInteger` array containing exactly 2 elements, m and k, in this order.

4. Implement a method `long check(BigInteger[] a, BigInteger d, BigInteger p, BigInteger n)` that, given a point a and the values d, p, and n, generates a random `BigInteger` exponent m modulo n, computes $b = a^m$ using method `exp()`, recovers the discrete logarithm m' from a and b using method `rho()`, checks whether $m = m'$ (and throws a RuntimeException if they don't match) and returns the number of steps k that method rho needed to compute m. Also implement a driver program that calls method `check(...)` a given number N of times, computes the average number $\langle k \rangle = \frac{1}{N}\sum_{i=1}^{N} k_i$ of

steps needed to compute N random discrete logarithms, and prints the result. Run your program and summarize your results using the following parameters: $p = 2^{16} - 17$, $d = 154$, $n = 16339$, $a = (12, 61833)$, $N = 1000$.

5. (**bonus**) Implement and report on all of the following parameter sets as well ($N = 1000$ for each of them):
   a. $p = 2^{18} - 5$, $d = 294$, $n = 65717$, and $a = (5, 261901)$.
   b. $p = 2^{20} - 5$, $d = 47$, $n = 262643$, and $a = (3, 111745)$.
   c. $p = 2^{22} - 17$, $d = 314$, $n = 1049497$, and $a = (4, 85081)$.

Keep in mind that you must present your project in the last lecture (this will be worth the remaining **10 points** of this assignment), in the same fashion you would present your research at a conference (or during the defense), or to your customers (or the company's CEO) during a business meeting.