```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Gage Farmer worked on this with Tucker in lab
 *
 */
public final class CryptoUtilities {

    /**
     * Private constructor so this utility class cannot be instantiated.
     */
    private CryptoUtilities() {
    }

    /**
     * Useful constant, not a magic number: 3.
     */
    private static final int THREE = 3;
    private static final NaturalNumber ZERO = new NaturalNumber2();
    private static final NaturalNumber ONE = new NaturalNumber2(1);
    private static final NaturalNumber TWO = new NaturalNumber2(2);

    /**
     * Pseudo-random number generator.
     */
    private static final Random GENERATOR = new Random1L();

    /**
     * Returns a random number uniformly distributed in the interval [0, n].
     *
     * @param n
     *            top end of interval
     * @return random number in interval
     * @requires n > 0
     * @ensures <pre>
     * randomNumber = [a random number uniformly distributed in [0, n]]
     * </pre>
     */
    public static NaturalNumber randomNumber(NaturalNumber n) {
        assert !n.isZero() : "Violation of: n > 0";
        final int base = 10;
        NaturalNumber result;
        int d = n.divideBy10();
        if (n.isZero()) {
            /*
             * Incoming n has only one digit and it is d, so generate a random
             * number uniformly distributed in [0, d]
             */
            int x = (int) ((d + 1) * GENERATOR.nextDouble());
            result = new NaturalNumber2(x);
            n.multiplyBy10(d);
        } else {
            /*
             * Incoming n has more than one digit, so generate a random number
             * (NaturalNumber) uniformly distributed in [0, n], and another
             * (int) uniformly distributed in [0, 9] (i.e., a random digit)
             */
            result = randomNumber(n);
            int lastDigit = (int) (base * GENERATOR.nextDouble());
            result.multiplyBy10(lastDigit);
            n.multiplyBy10(d);
            if (result.compareTo(n) > 0) {
                /*
                 * In this case, we need to try again because generated number
                 * is greater than n; the recursive call's argument is not
                 * "smaller" than the incoming value of n, but this recursive
                 * call has no more than a 90% chance of being made (and for
                 * large n, far less than that), so the probability of
                 * termination is 1
                 */
                result = randomNumber(n);
            }
        }
        return result;
    }

    /**
     * Finds the greatest common divisor of n and m.
     *
     * @param n
     *            one number
     * @param m
     *            the other number
     * @updates n
     * @clears m
     * @ensures n = [greatest common divisor of #n and #m]
     */
    public static void reduceToGCD(NaturalNumber n, NaturalNumber m) {
        /*
         * Use Euclid's algorithm; in pseudocode: if m = 0 then GCD(n, m) = n
         * else GCD(n, m) = GCD(m, n mod m)
         */
        NaturalNumber mod = new NaturalNumber2();

        if (m.compareTo(ZERO) != 0) {
            mod.copyFrom(n.divide(m));
            reduceToGCD(m, mod);
            n.copyFrom(m);
            m.copyFrom(mod);
        }

    }

    /**
     * Reports whether n is even.
     *
     * @param n
     *            the number to be checked
     * @return true iff n is even
     * @ensures isEven = (n mod 2 = 0)
     */
    public static boolean isEven(NaturalNumber n) {

        boolean even = false;
        int nInt = n.toInt();

        // TODO - fill in body

        if (nInt % 2 == 0) {
            even = true;
        }

        /*
         * This line added just to make the program compilable. Should be
         * replaced with appropriate return statement.
         */
        return even;
    }

    /**
     * Updates n to its p-th power modulo m.
     *
     * @param n
     *            number to be raised to a power
     * @param p
     *            the power
     * @param m
     *            the modulus
     * @updates n
     * @requires m > 1
     * @ensures n = #n ^ (p) mod m
     */
    public static void powerMod(NaturalNumber n, NaturalNumber p,
            NaturalNumber m) {
        assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";

        /*
         * Use the fast-powering algorithm as previously discussed in class,
         * with the additional feature that every multiplication is followed
         * immediately by "reducing the result modulo m"
         */

        // TODO - fill in body
        int intN = n.toInt();
        int intP = p.toInt();
        int intM = m.toInt();
        int nTot = n.toInt();

        for (int i = 1; i < intP; i++) {
            nTot *= intN;
            nTot = nTot % intM;
        }

        if (intP == 0) {
            nTot = 1;
        }

        NaturalNumber nTotNN = new NaturalNumber2(nTot);
        n.copyFrom(nTotNN);

    }

    /**
     * Reports whether w is a "witness" that n is composite, in the sense that
     * either it is a square root of 1 (mod n), or it fails to satisfy the
     * criterion for primality from Fermat's theorem.
     *
     * @param w
     *            witness candidate
     * @param n
     *            number being checked
     * @return true iff w is a "witness" that n is composite
     * @requires n > 2 and 1 < w < n - 1
     * @ensures <pre>
     * isWitnessToCompositeness =
     *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
     * </pre>
     */
    public static boolean isWitnessToCompositeness(NaturalNumber w,
            NaturalNumber n) {
        assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
        assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 < w";
        n.decrement();
        assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
        n.increment();

        boolean isWitness = false;
        NaturalNumber remainder;
        NaturalNumber a = w.newInstance();
        NaturalNumber p = n.newInstance();

        p.decrement();
        a.power(p.toInt());
        p.increment();
        remainder = a.divide(p);

        if (remainder.compareTo(ONE) == 1) {
            isWitness = true;
        }

        return isWitness;
    }

    /**
     * Reports whether n is a prime; may be wrong with "low" probability.
     *
     * @param n
     *            number to be checked
     * @return true means n is very likely prime; false means n is definitely
     *         composite
     * @requires n > 1
     * @ensures <pre>
     * isPrime1 = [n is a prime number, with small probability of error
     *         if it is reported to be prime, and no chance of error if it is
     *         reported to be composite]
     * </pre>
     */
    public static boolean isPrime1(NaturalNumber n) {
        assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
        boolean isPrime;
        if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
            /*
             * 2 and 3 are primes
             */
```

```java
            isPrime = true;
        } else if (isEven(n)) {
            /*
             * evens are composite
             */
            isPrime = false;
        } else {
            /*
             * odd n >= 5: simply check whether 2 is a witness that n is
             * composite (which works surprisingly well :-)
             */
            isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
        }
        return isPrime;
    }

    /**
     * Reports whether n is a prime; may be wrong with "low" probability.
     *
     * @param n
     *            number to be checked
     * @return true means n is very likely prime; false means n is definitely
     *         composite
     * @requires n > 1
     * @ensures <pre>
     * isPrime2 = [n is a prime number, with small probability of error
     *         if it is reported to be prime, and no chance of error if it is
     *         reported to be composite]
     * </pre>
     */
    public static boolean isPrime2(NaturalNumber n) {
        assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";

        boolean isPrime = true;
        /*
         * p = n a = ans
         *
         * ans to the (n-1) divided by n
         *
         */

        NaturalNumber randTop = n.newInstance();
        NaturalNumber nDec = n.newInstance();
        randTop.copyFrom(n);
        nDec.copyFrom(n);
        nDec.decrement();
        NaturalNumber ans;

        for (int i = 0; i < 25; i++) {
            ans = randomNumber(randTop);
            ans.increment();
            ans.power(nDec.toInt());
            ans.divide(n);

            if (ans.compareTo(ONE) > 0) {
                isPrime = false;
            }

        }

        /*
         * as stupid as i look for not figuring out the bug in my 'is prime'
         * function, at least i didn't go on stackoverflow to find the answer
         *
         * that's worth something, right?
         *
         *
         *
         *
         *
         *
         *
         * ....right?
         */

        return isPrime;
    }

    /**
     * Generates a likely prime number at least as large as some given number.
     *
     * @param n
     *            minimum value of likely prime
     * @updates n
     * @requires n > 1
     * @ensures n >= #n and [n is very likely a prime number]
     */
    public static void generateNextLikelyPrime(NaturalNumber n) {
        assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";

        if (!isEven(n)) {
            n.increment();
            generateNextLikelyPrime(n);
        } else {
            if (!isPrime2(n)) {
                n.increment();
                n.increment();
                generateNextLikelyPrime(n);
            }
        }

    }

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments
     */
    public static void main(String[] args) {
        SimpleReader in = new SimpleReader1L();
        SimpleWriter out = new SimpleWriter1L();

        /*
         * Sanity check of randomNumber method -- just so everyone can see how
         * it might be "tested"
         */
        final int testValue = 17;
        final int testSamples = 100000;
        NaturalNumber test = new NaturalNumber2(testValue);
        int[] count = new int[testValue + 1];
        for (int i = 0; i < count.length; i++) {
            count[i] = 0;
        }
        for (int i = 0; i < testSamples; i++) {
            NaturalNumber rn = randomNumber(test);
            assert rn.compareTo(test) <= 0 : "Help!";
            count[rn.toInt()]++;
        }
        for (int i = 0; i < count.length; i++) {
            out.println("count[" + i + "] = " + count[i]);
        }
        out.println("  expected value = "
                + (double) testSamples / (double) (testValue + 1));

        /*
         * Check user-supplied numbers for primality, and if a number is not
         * prime, find the next likely prime after it
         */
        while (true) {
            out.print("n = ");
            NaturalNumber n = new NaturalNumber2(in.nextLine());
            if (n.compareTo(new NaturalNumber2(2)) < 0) {
                out.println("Bye!");
                break;
            } else {
                if (isPrime1(n)) {
                    out.println(n + " is probably a prime number"
                            + " according to isPrime1.");
                } else {
                    out.println(n + " is a composite number"
                            + " according to isPrime1.");
                }
                if (isPrime2(n)) {
                    out.println(n + " is probably a prime number"
                            + " according to isPrime2.");
                } else {
                    out.println(n + " is a composite number"
                            + " according to isPrime2.");
                    generateNextLikelyPrime(n);
                    out.println("  next likely prime is " + n);
                }
            }
        }

        /*
         * Close input and output streams
         */
        in.close();
        out.close();
    }

}
```