```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)
             */
```

```
 67            result = randomNumber(n);
 68            int lastDigit = (int) (base * GENERATOR.nextDouble());
 69            result.multiplyBy10(lastDigit);
 70            n.multiplyBy10(d);
 71            if (result.compareTo(n) > 0) {
 72                /*
 73                 * In this case, we need to try again because generated number
 74                 * is greater than n; the recursive call's argument is not
 75                 * "smaller" than the incoming value of n, but this recursive
 76                 * call has no more than a 90% chance of being made (and for
 77                 * large n, far less than that), so the probability of
 78                 * termination is 1
 79                 */
 80                result = randomNumber(n);
 81            }
 82        }
 83        return result;
 84    }
 85
 86    /**
 87     * Finds the greatest common divisor of n and m.
 88     *
 89     * @param n
 90     *            one number
 91     * @param m
 92     *            the other number
 93     * @updates n
 94     * @clears m
 95     * @ensures n = [greatest common divisor of #n and #m]
 96     */
 97    public static void reduceToGCD(NaturalNumber n, NaturalNumber m) {
 98        /*
 99         * Use Euclid's algorithm; in pseudocode: if m = 0 then GCD(n, m) = n
100         * else GCD(n, m) = GCD(m, n mod m)
101         */
102        NaturalNumber mod = new NaturalNumber2();
103
104        if (m.compareTo(ZERO) != 0) {
105            mod.copyFrom(n.divide(m));
106            reduceToGCD(m, mod);
107            n.copyFrom(m);
108            m.copyFrom(mod);
109        }
110
111    }
112
113    /**
114     * Reports whether n is even.
115     *
116     * @param n
117     *            the number to be checked
118     * @return true iff n is even
119     * @ensures isEven = (n mod 2 = 0)
120     */
121    public static boolean isEven(NaturalNumber n) {
122
123        boolean even = false;
124        int nInt = n.toInt();
125
```

```
126         // TODO - fill in body
127
128         if (nInt % 2 == 0) {
129             even = true;
130         }
131
132         /*
133          * This line added just to make the program compilable. Should be
134          * replaced with appropriate return statement.
135          */
136         return even;
137     }
138
139     /**
140      * Updates n to its p-th power modulo m.
141      *
142      * @param n
143      *            number to be raised to a power
144      * @param p
145      *            the power
146      * @param m
147      *            the modulus
148      * @updates n
149      * @requires m > 1
150      * @ensures n = #n ^ (p) mod m
151      */
152     public static void powerMod(NaturalNumber n, NaturalNumber p,
153             NaturalNumber m) {
154         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
155
156         /*
157          * Use the fast-powering algorithm as previously discussed in class,
158          * with the additional feature that every multiplication is followed
159          * immediately by "reducing the result modulo m"
160          */
161
162         // TODO - fill in body
163         int intN = n.toInt();
164         int intP = p.toInt();
165         int intM = m.toInt();
166         int nTot = n.toInt();
167
168         for (int i = 1; i < intP; i++) {
169             nTot *= intN;
170             nTot = nTot % intM;
171         }
172
173         if (intP == 0) {
174             nTot = 1;
175         }
176
177         NaturalNumber nTotNN = new NaturalNumber2(nTot);
178         n.copyFrom(nTotNN);
179
180     }
181
182     /**
183      * Reports whether w is a "witness" that n is composite, in the sense that
184      * either it is a square root of 1 (mod n), or it fails to satisfy the
```

```
185      * criterion for primality from Fermat's theorem.
186      *
187      * @param w
188      *            witness candidate
189      * @param n
190      *            number being checked
191      * @return true iff w is a "witness" that n is composite
192      * @requires n > 2 and 1 < w < n - 1
193      * @ensures <pre>
194      * isWitnessToCompositeness =
195      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
196      * </pre>
197      */
198     public static boolean isWitnessToCompositeness(NaturalNumber w,
199             NaturalNumber n) {
200         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
201         assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 < w";
202         n.decrement();
203         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
204         n.increment();
205
206         boolean isWitness = false;
207         NaturalNumber remainder;
208         NaturalNumber a = w.newInstance();
209         NaturalNumber p = n.newInstance();
210
211         p.decrement();
212         a.power(p.toInt());
213         p.increment();
214         remainder = a.divide(p);
215
216         if (remainder.compareTo(ONE) == 1) {
217             isWitness = true;
218         }
219
220         return isWitness;
221     }
222
223     /**
224      * Reports whether n is a prime; may be wrong with "low" probability.
225      *
226      * @param n
227      *            number to be checked
228      * @return true means n is very likely prime; false means n is definitely
229      *         composite
230      * @requires n > 1
231      * @ensures <pre>
232      * isPrime1 = [n is a prime number, with small probability of error
233      *         if it is reported to be prime, and no chance of error if it is
234      *         reported to be composite]
235      * </pre>
236      */
237     public static boolean isPrime1(NaturalNumber n) {
238         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
239         boolean isPrime;
240         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
241             /*
242              * 2 and 3 are primes
243              */
```

```
244             isPrime = true;
245         } else if (isEven(n)) {
246             /*
247              * evens are composite
248              */
249             isPrime = false;
250         } else {
251             /*
252              * odd n >= 5: simply check whether 2 is a witness that n is
253              * composite (which works surprisingly well :-)
254              */
255             isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
256         }
257         return isPrime;
258     }
259
260     /**
261      * Reports whether n is a prime; may be wrong with "low" probability.
262      *
263      * @param n
264      *            number to be checked
265      * @return true means n is very likely prime; false means n is definitely
266      *         composite
267      * @requires n > 1
268      * @ensures <pre>
269      * isPrime2 = [n is a prime number, with small probability of error
270      *         if it is reported to be prime, and no chance of error if it is
271      *         reported to be composite]
272      * </pre>
273      */
274     public static boolean isPrime2(NaturalNumber n) {
275         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
276
277         boolean isPrime = true;
278         /*
279          * p = n a = ans
280          *
281          * ans to the (n-1) divided by n
282          *
283          */
284
285         NaturalNumber randTop = n.newInstance();
286         NaturalNumber nDec = n.newInstance();
287         randTop.copyFrom(n);
288         nDec.copyFrom(n);
289         nDec.decrement();
290         NaturalNumber ans;
291
292         for (int i = 0; i < 25; i++) {
293             ans = randomNumber(randTop);
294             ans.increment();
295             ans.power(nDec.toInt());
296             ans.divide(n);
297
298             if (ans.compareTo(ONE) > 0) {
299                 isPrime = false;
300             }
301
302         }
```

```
303
304         /*
305          * as stupid as i look for not figuring out the bug in my 'is prime'
306          * function, at least i didn't go on stackoverflow to find the answer
307          *
308          * that's worth something, right?
309          *
310          *
311          *
312          *
313          *
314          *
315          *
316          * ....right?
317          */
318
319         return isPrime;
320     }
321
322     /**
323      * Generates a likely prime number at least as large as some given number.
324      *
325      * @param n
326      *            minimum value of likely prime
327      * @updates n
328      * @requires n > 1
329      * @ensures n >= #n and [n is very likely a prime number]
330      */
331     public static void generateNextLikelyPrime(NaturalNumber n) {
332         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
333
334         if (!isEven(n)) {
335             n.increment();
336             generateNextLikelyPrime(n);
337         } else {
338             if (!isPrime2(n)) {
339                 n.increment();
340                 n.increment();
341                 generateNextLikelyPrime(n);
342             }
343         }
344
345     }
346
347     /**
348      * Main method.
349      *
350      * @param args
351      *            the command line arguments
352      */
353     public static void main(String[] args) {
354         SimpleReader in = new SimpleReader1L();
355         SimpleWriter out = new SimpleWriter1L();
356
357         /*
358          * Sanity check of randomNumber method -- just so everyone can see how
359          * it might be "tested"
360          */
361         final int testValue = 17;
```

```
362         final int testSamples = 100000;
363         NaturalNumber test = new NaturalNumber2(testValue);
364         int[] count = new int[testValue + 1];
365         for (int i = 0; i < count.length; i++) {
366             count[i] = 0;
367         }
368         for (int i = 0; i < testSamples; i++) {
369             NaturalNumber rn = randomNumber(test);
370             assert rn.compareTo(test) <= 0 : "Help!";
371             count[rn.toInt()]++;
372         }
373         for (int i = 0; i < count.length; i++) {
374             out.println("count[" + i + "] = " + count[i]);
375         }
376         out.println("  expected value = "
377                 + (double) testSamples / (double) (testValue + 1));
378
379         /*
380          * Check user-supplied numbers for primality, and if a number is not
381          * prime, find the next likely prime after it
382          */
383         while (true) {
384             out.print("n = ");
385             NaturalNumber n = new NaturalNumber2(in.nextLine());
386             if (n.compareTo(new NaturalNumber2(2)) < 0) {
387                 out.println("Bye!");
388                 break;
389             } else {
390                 if (isPrime1(n)) {
391                     out.println(n + " is probably a prime number"
392                             + " according to isPrime1.");
393                 } else {
394                     out.println(n + " is a composite number"
395                             + " according to isPrime1.");
396                 }
397                 if (isPrime2(n)) {
398                     out.println(n + " is probably a prime number"
399                             + " according to isPrime2.");
400                 } else {
401                     out.println(n + " is a composite number"
402                             + " according to isPrime2.");
403                     generateNextLikelyPrime(n);
404                     out.println("  next likely prime is " + n);
405                 }
406             }
407         }
408
409         /*
410          * Close input and output streams
411          */
412         in.close();
413         out.close();
414     }
415
416 }
```