```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Put your name here
 *
 */
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;

    /**
     * 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);
```

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

```
122         boolean even = false;
123
124         int r = n.divideBy10();
125
126         if (r % 2 == 0) {
127             even = true;
128         }
129
130         n.multiplyBy10(r);
131
132         return even;
133     }
134
135     /**
136      * Updates n to its p-th power modulo m.
137      *
138      * @param n
139      *            number to be raised to a power
140      * @param p
141      *            the power
142      * @param m
143      *            the modulus
144      * @updates n
145      * @requires m > 1
146      * @ensures n = #n ^ (p) mod m
147      */
148     public static void powerMod(NaturalNumber n, NaturalNumber p,
149             NaturalNumber m) {
150         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
151
152         /*
153          * Use the fast-powering algorithm as previously discussed in class,
154          * with the additional feature that every multiplication is followed
155          * immediately by "reducing the result modulo m"
156          */
157
158         NaturalNumber one = new NaturalNumber2(1);
159         NaturalNumber two = new NaturalNumber2(2);
160         NaturalNumber ncopy = new NaturalNumber2(n);
161
162         if (p.isZero()) {
163             n.transferFrom(one);
164         } else if (p.compareTo(one) == 0) {
165             NaturalNumber remainder = n.divide(m);
166             n.transferFrom(remainder);
167         } else if (isEven(p)) {
168             p.divide(two);
169
170             powerMod(n, p, m);
171             n.power(2);
172
173             NaturalNumber remainder = n.divide(m);
174             n.transferFrom(remainder);
175
176             p.multiply(two);
177         } else {
178             p.divide(two);
```

```
179
180             powerMod(n, p, m);
181             n.power(2);
182
183             n.multiply(ncopy.divide(m));
184
185             NaturalNumber remainder = n.divide(m);
186             n.transferFrom(remainder);
187
188             p.multiply(two);
189             p.increment();
190         }
191     }
192
193     /**
194      * Reports whether w is a "witness" that n is composite, in the sense that
195      * either it is a square root of 1 (mod n), or it fails to satisfy the
196      * criterion for primality from Fermat's theorem.
197      *
198      * @param w
199      *            witness candidate
200      * @param n
201      *            number being checked
202      * @return true iff w is a "witness" that n is composite
203      * @requires n > 2 and 1 < w < n - 1
204      * @ensures <pre>
205      * isWitnessToCompositeness =
206      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
207      * </pre>
208      */
209     public static boolean isWitnessToCompositeness(NaturalNumber w,
210             NaturalNumber n) {
211         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
212         assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 < w";
213         n.decrement();
214         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
215         n.increment();
216
217         NaturalNumber one = new NaturalNumber2(1);
218         NaturalNumber two = new NaturalNumber2(2);
219
220         NaturalNumber nCopy = new NaturalNumber2(n);
221         nCopy.decrement();
222
223         boolean witness = false;
224
225         powerMod(w, two, n);
226         if (w.compareTo(one) == 0) {
227             witness = true;
228         }
229
230         powerMod(w, nCopy, n);
231         if (w.toInt() != 1) {
232             witness = true;
233         }
234
235         return witness;
```

```
236     }
237
238     /**
239      * Reports whether n is a prime; may be wrong with "low" probability.
240      *
241      * @param n
242      *            number to be checked
243      * @return true means n is very likely prime; false means n is definitely
244      *         composite
245      * @requires n > 1
246      * @ensures <pre>
247      * isPrime1 = [n is a prime number, with small probability of error
248      *         if it is reported to be prime, and no chance of error if it is
249      *         reported to be composite]
250      * </pre>
251      */
252     public static boolean isPrime1(NaturalNumber n) {
253         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
254
255         NaturalNumber two = new NaturalNumber2(2);
256         NaturalNumber three = new NaturalNumber2(3);
257
258         boolean prime;
259         if (n.compareTo(three) <= 0) {
260             prime = true;
261         } else if (isEven(n)) {
262             prime = false;
263         } else {
264             prime = !isWitnessToCompositeness(two, n);
265         }
266         return prime;
267     }
268
269     /**
270      * Reports whether n is a prime; may be wrong with "low" probability.
271      *
272      * @param n
273      *            number to be checked
274      * @return true means n is very likely prime; false means n is definitely
275      *         composite
276      * @requires n > 1
277      * @ensures <pre>
278      * isPrime2 = [n is a prime number, with small probability of error
279      *         if it is reported to be prime, and no chance of error if it is
280      *         reported to be composite]
281      * </pre>
282      */
283     public static boolean isPrime2(NaturalNumber n) {
284         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
285
286         /*
287          * Use the ability to generate random numbers (provided by the
288          * randomNumber method above) to generate several witness candidates --
289          * say, 10 to 50 candidates -- guessing that n is prime only if none of
290          * these candidates is a witness to n being composite (based on fact #3
291          * as described in the project description); use the code for isPrime1
292          * as a guide for how to do this, and pay attention to the requires
```

```
293          * clause of isWitnessToCompositeness
294          */
295
296         boolean prime = true;
297
298         for (int i = 1; prime == true && i < 20; i++) {
299             NaturalNumber copy = new NaturalNumber2(n);
300             copy.decrement();
301
302             NaturalNumber value = n.newInstance();
303             value.increment();
304
305             NaturalNumber witness = randomNumber(n);
306
307             while (!(witness.compareTo(value) > 0
308                     && witness.compareTo(copy) < 0)) {
309                 witness = randomNumber(n);
310             }
311
312             prime = !isWitnessToCompositeness(witness, n);
313
314         }
315
316         return prime;
317     }
318
319     /**
320      * Generates a likely prime number at least as large as some given number.
321      *
322      * @param n
323      *            minimum value of likely prime
324      * @updates n
325      * @requires n > 1
326      * @ensures n >= #n and [n is very likely a prime number]
327      */
328     public static void generateNextLikelyPrime(NaturalNumber n) {
329         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
330
331         /*
332          * Use isPrime2 to check numbers, starting at n and increasing through
333          * the odd numbers only (why?), until n is likely prime
334          */
335         while (!isPrime2(n)) {
336             n.increment();
337
338             if (isEven(n)) {
339                 n.increment();
340             }
341         }
342
343     }
344
345     /**
346      * Main method.
347      *
348      * @param args
349      *            the command line arguments
```

```
350      */
351     public static void main String[] args) {
352         SimpleReader in = new SimpleReader1L();
353         SimpleWriter out = new SimpleWriter1L();
354
355         /*
356          * Sanity check of randomNumber method -- just so everyone can see how
357          * it might be "tested"
358          */
359         final int testValue = 17;
360         final int testSamples = 100000;
361         NaturalNumber test = new NaturalNumber2(testValue);
362         int[] count = new int[testValue + 1];
363         for (int i = 0; i < count.length; i++) {
364             count[i] = 0;
365         }
366         for (int i = 0; i < testSamples; i++) {
367             NaturalNumber rn = randomNumber(test);
368             assert rn.compareTo(test) <= 0 : "Help!";
369             count[rn.toInt()]++;
370         }
371         for (int i = 0; i < count.length; i++) {
372             out.println("count[" + i + "] = " + count[i]);
373         }
374         out.println("  expected value = "
375                 + (double) testSamples / (double) (testValue + 1));
376
377         /*
378          * Check user-supplied numbers for primality, and if a number is not
379          * prime, find the next likely prime after it
380          */
381         while (true) {
382             out.print("n = ");
383             NaturalNumber n = new NaturalNumber2(in.nextLine());
384             if (n.compareTo(new NaturalNumber2(2)) < 0) {
385                 out.println("Bye!");
386                 break;
387             } else {
388                 if (isPrime1(n)) {
389                     out.println(n + " is probably a prime number"
390                             + " according to isPrime1.");
391                 } else {
392                     out.println(n + " is a composite number"
393                             + " according to isPrime1.");
394                 }
395                 if (isPrime2(n)) {
396                     out.println(n + " is probably a prime number"
397                             + " according to isPrime2.");
398                 } else {
399                     out.println(n + " is a composite number"
400                             + " according to isPrime2.");
401                     generateNextLikelyPrime(n);
402                     out.println("  next likely prime is " + n);
403                 }
404             }
405         }
406
```

```
407         /*
408          * Close input and output streams
409          */
410         in.close();
411         out.close();
412     }
413
414 }
```