```java
import components.naturalnumber.NaturalNumber;
import components.naturalnumber.NaturalNumber2;
import components.random.Random;
import components.random.Random1L;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Jonathan Pater
 *
 */
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]]
```

```
43      * </pre>
44      */
45     public static NaturalNumber randomNumber(NaturalNumber n) { //
   USED TO GENERATE W
46         assert !n.isZero() : "Violation of: n > 0";
47         final int base = 10;
48         NaturalNumber result;
49         int d = n.divideBy10();
50         if (n.isZero()) {
51             /*
52              * Incoming n has only one digit and it is d, so
   generate a random
53              * number uniformly distributed in [0, d]
54              */
55             int x = (int) ((d + 1) * GENERATOR.nextDouble());
56             result = new NaturalNumber2(x);
57             n.multiplyBy10(d);
58         } else {
59             /*
60              * Incoming n has more than one digit, so generate a
   random number
61              * (NaturalNumber) uniformly distributed in [0, n], and
   another
62              * (int) uniformly distributed in [0, 9] (i.e., a
   random digit)
63              */
64             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          // TODO - fill in body
101          NaturalNumber zero = new NaturalNumber2(0);
102          int compare = m.compareTo(zero);
103          if (compare != 0) {
104              NaturalNumber rem = n.divide(m);
105              n.transferFrom(m);
106              reduceToGCD(n, rem);
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          // TODO - fill in body
```

```
121         boolean even = false;
122         /*
123          * This line added just to make the program compilable.
    Should be
124          * replaced with appropriate return statement.
125          */
126         NaturalNumber num = new NaturalNumber2(n);
127         NaturalNumber rem = num.divide(new NaturalNumber2(2));
128         int compare = rem.compareTo(new NaturalNumber2(0));
129         if (compare == 0) {
130             even = true;
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          * Use the fast-powering algorithm as previously discussed
    in class,
153          * with the additional feature that every multiplication is
    followed
154          * immediately by "reducing the result modulo m"
155          */
156
157         // TODO - fill in body
158         NaturalNumber zero = new NaturalNumber2(0);
159         NaturalNumber one = new NaturalNumber2(1);
160         NaturalNumber pCopy = new NaturalNumber2(p);
161         int comp1 = p.compareTo(zero);
```

```
162         if (comp1 == 0) {
163             n.copyFrom(one);
164         } else {
165             NaturalNumber temp = new NaturalNumber2(n);
166             NaturalNumber zeroRem = pCopy.divide(new
    NaturalNumber2(2));
167             powerMod(n, pCopy, m);
168             NaturalNumber temp1 = new NaturalNumber2(n);
169             n.multiply(temp1);
170             int comp2 = zeroRem.compareTo(zero);
171             if (comp2 != 0) {
172                 n.multiply(temp);
173             }
174             NaturalNumber rem = n.divide(m);
175             n.copyFrom(rem);
176         }
177     }
178
179     /**
180      * Reports whether w is a "witness" that n is composite, in the
    sense that
181      * either it is a square root of 1 (mod n), or it fails to
    satisfy the
182      * criterion for primality from Fermat's theorem.
183      *
184      * @param w
185      *            witness candidate
186      * @param n
187      *            number being checked
188      * @return true iff w is a "witness" that n is composite
189      * @requires n > 2 and 1 < w < n - 1
190      * @ensures <pre>
191      * isWitnessToCompositeness =
192      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
193      * </pre>
194      */
195     public static boolean isWitnessToCompositeness(NaturalNumber w,
196             NaturalNumber n) {
197         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation
    of: n > 2";
198         assert (new NaturalNumber2(1)).compareTo(w) < 0 :
    "Violation of: 1 < w";
199         n.decrement();
200         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
201         n.increment();
```

```
202
203         // TODO - fill in body
204         boolean isWitness = false;
205         NaturalNumber one = new NaturalNumber2(1);
206         NaturalNumber two = new NaturalNumber2(2);
207         NaturalNumber wCopy1 = new NaturalNumber2(w);
208         NaturalNumber wCopy2 = new NaturalNumber2(w);
209         NaturalNumber nCopy = new NaturalNumber2(n);
210         NaturalNumber nCopy2 = new NaturalNumber2(n);
211         powerMod(wCopy1, two, nCopy2); // sets wCopy1's value to
    w^2 mod n's value
212         int compare1 = wCopy1.compareTo(one); // checks if wCopy1 =
    1
213         if (compare1 == 0) {
214             isWitness = true;
215         } else {
216             nCopy.subtract(one); // n-1 for w^(n-1) mod n
217             powerMod(wCopy2, nCopy, nCopy2);
218             compare1 = wCopy2.compareTo(one);
219             if (compare1 != 0) {
220                 isWitness = true;
221             }
222         }
223         return isWitness;
224     }
225
226     /**
227      * Reports whether n is a prime; may be wrong with "low"
    probability.
228      *
229      * @param n
230      *            number to be checked
231      * @return true means n is very likely prime; false means n is
    definitely
232      *         composite
233      * @requires n > 1
234      * @ensures <pre>
235      * isPrime1 = [n is a prime number, with small probability of
    error
236      *         if it is reported to be prime, and no chance of
    error if it is
237      *         reported to be composite]
238      * </pre>
239      */
240     public static boolean isPrime1(NaturalNumber n) {
```

```
241         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
242         boolean isPrime;
243         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
244             /*
245              * 2 and 3 are primes
246              */
247             isPrime = true;
248         } else if (isEven(n)) {
249             /*
250              * evens are composite
251              */
252             isPrime = false;
253         } else {
254             /*
255              * odd n >= 5: simply check whether 2 is a witness that
    n is
256              * composite (which works surprisingly well :-)
257              */
258             isPrime = !isWitnessToCompositeness(new
    NaturalNumber2(2), n);
259         }
260         return isPrime;
261     }
262
263     /**
264      * Reports whether n is a prime; may be wrong with "low"
    probability.
265      *
266      * @param n
267      *            number to be checked
268      * @return true means n is very likely prime; false means n is
    definitely
269      *         composite
270      * @requires n > 1
271      * @ensures <pre>
272      * isPrime2 = [n is a prime number, with small probability of
    error
273      *         if it is reported to be prime, and no chance of
    error if it is
274      *         reported to be composite]
275      * </pre>
276      */
277     public static boolean isPrime2(NaturalNumber n) {
278         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
```

```
of: n > 1";
279
280         /*
281          * Use the ability to generate random numbers (provided by
the
282          * randomNumber method above) to generate several witness
candidates --
283          * say, 10 to 50 candidates -- guessing that n is prime
only if none of
284          * these candidates is a witness to n being composite
(based on fact #3
285          * as described in the project description); use the code
for isPrime1
286          * as a guide for how to do this, and pay attention to the
requires
287          * clause of isWitnessToCompositeness
288          */
289
290         // TODO - fill in body
291         boolean isPrime;
292         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
293             // a number <=3 is prime
294             isPrime = true;
295         } else if (isEven(n)) {
296             isPrime = false;
297         } else {
298             isPrime = true; // now witnesses must prove that n is
not prime
299             NaturalNumber[] witnesses = new NaturalNumber2[10];
300             //Array of size 10 for 10 witness candidates
301             NaturalNumber nCopy = new NaturalNumber2(n);
302             NaturalNumber one = new NaturalNumber2(1);
303             nCopy.subtract(new NaturalNumber2(2));
304             for (int i = 0; i < witnesses.length; i++) {
305                 NaturalNumber w = randomNumber(nCopy);
306                 //random number 0 <= w <= n-2
307                 witnesses[i] = new NaturalNumber2();
308                 while (w.compareTo(one) <= 0) {
309                     //allows w to fulfill requires clause of
isWitnessToCompositeness
310                     w = randomNumber(nCopy);
311                 }
312                 witnesses[i].transferFrom(w);
313             }
314             nCopy.add(new NaturalNumber2(2)); //resets nCopy to
```

```
    original value
315             for (int i = 0; i < witnesses.length; i++) {
316                 boolean isWitness =
    isWitnessToCompositeness(witnesses[i], n);
317                 if (isWitness) {
318                     isPrime = false;
319                 }
320             }
321         }
322         return isPrime;
323     }
324 
325     /**
326      * Generates a likely prime number at least as large as some
    given number.
327      *
328      * @param n
329      *            minimum value of likely prime
330      * @updates n
331      * @requires n > 1
332      * @ensures n >= #n and [n is very likely a prime number]
333      */
334     public static void generateNextLikelyPrime(NaturalNumber n) {
335         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
336 
337         /*
338          * Use isPrime2 to check numbers, starting at n and
    increasing through
339          * the odd numbers only (why?), until n is likely prime
    why?: there are
340          * no even prime numbers after 2
341          */
342         // TODO - fill in body
343         NaturalNumber nCopy = new NaturalNumber2(n);
344         boolean isPrime = isPrime2(nCopy);
345         NaturalNumber two = new NaturalNumber2(2);
346         NaturalNumber one = new NaturalNumber2(1);
347         while (!isPrime) {
348             boolean even = isEven(n);
349             if (even) {
350                 n.add(one); // makes n odd if it is an even number
351                 isPrime = isPrime2(n);
352             } else {
353                 n.add(two);
```

```
354                 isPrime = isPrime2(n);
355                 // if n is already odd, then check if the next odd
    number is prime
356             }
357         }
358     }
359
360     /**
361      * Main method.
362      *
363      * @param args
364      *             the command line arguments
365      */
366     public static void main(String[] args) {
367         SimpleReader in = new SimpleReader1L();
368         SimpleWriter out = new SimpleWriter1L();
369
370         /*
371          * Sanity check of randomNumber method -- just so everyone
    can see how
372          * it might be "tested"
373          */
374         final int testValue = 17;
375         final int testSamples = 100000;
376         NaturalNumber test = new NaturalNumber2(testValue);
377         int[] count = new int[testValue + 1];
378         for (int i = 0; i < count.length; i++) {
379             count[i] = 0;
380         }
381         for (int i = 0; i < testSamples; i++) {
382             NaturalNumber rn = randomNumber(test);
383             assert rn.compareTo(test) <= 0 : "Help!";
384             count[rn.toInt()]++;
385         }
386         for (int i = 0; i < count.length; i++) {
387             out.println("count[" + i + "] = " + count[i]);
388         }
389         out.println("  expected value = "
390                 + (double) testSamples / (double) (testValue + 1));
391
392         /*
393          * Check user-supplied numbers for primality, and if a
    number is not
394          * prime, find the next likely prime after it
395          */
```

```
396         while (true) {
397             out.print("n = ");
398             NaturalNumber n = new NaturalNumber2(in.nextLine());
399             if (n.compareTo(new NaturalNumber2(2)) < 0) {
400                 out.println("Bye!");
401                 break;
402             } else {
403                 if (isPrime1(n)) {
404                     out.println(n + " is probably a prime number"
405                             + " according to isPrime1.");
406                 } else {
407                     out.println(n + " is a composite number"
408                             + " according to isPrime1.");
409                 }
410                 if (isPrime2(n)) {
411                     out.println(n + " is probably a prime number"
412                             + " according to isPrime2.");
413                 } else {
414                     out.println(n + " is a composite number"
415                             + " according to isPrime2.");
416                     generateNextLikelyPrime(n);
417                     out.println("  next likely prime is " + n);
418                 }
419             }
420         }
421
422         /*
423          * Close input and output streams
424          */
425         in.close();
426         out.close();
427     }
428
429 }
430
```