```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 Yiming Cheng
 *
 */
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)
```

```
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        if (m.isZero()) {
101            //check the whether the m would get the right answer and pass it
102            m.transferFrom(n);
103        } else if (n.isZero()) {
104            m.copyFrom(n);
105        } else if (n.compareTo(m) > 0) {
106            NaturalNumber remiander = n.divide(m);
107            reduceToGCD(m, remiander);
108            n.transferFrom(remiander);
109        } else {
110            NaturalNumber remiander = m.divide(n);
111            reduceToGCD(m, remiander);
112            n.transferFrom(remiander);
113        }
114
115    }
116
117    /**
118     * Reports whether n is even.
119     *
120     * @param n
121     *            the number to be checked
122     * @return true iff n is even
123     * @ensures isEven = (n mod 2 = 0)
124     */
```

```
125     public static boolean isEven(NaturalNumber n) {
126
127         boolean even = true;
128         int number = 0;
129
130         number = n.divideBy10();
131         int num = number % 2;
132         //find the remainder of the number
133         if (num != 0) {
134             even = false;
135         }
136
137         n.multiplyBy10(number);
138         //give back the right answer.
139
140         return even;
141     }
142
143     /**
144      * Updates n to its p-th power modulo m.
145      *
146      * @param n
147      *            number to be raised to a power
148      * @param p
149      *            the power
150      * @param m
151      *            the modulus
152      * @updates n
153      * @requires m > 1
154      * @ensures n = #n ^ (p) mod m
155      */
156     public static void powerMod(NaturalNumber n, NaturalNumber p,
157             NaturalNumber m) {
158         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
159
160         /*
161          * Use the fast-powering algorithm as previously discussed in class,
162          * with the additional feature that every multiplication is followed
163          * immediately by "reducing the result modulo m"
164          */
165         NaturalNumber max = new NaturalNumber2(Integer.MAX_VALUE);
166         NaturalNumber num = new NaturalNumber2(n);
167         NaturalNumber power = new NaturalNumber2(p);
168         NaturalNumber two = new NaturalNumber2(2);
169         int number = 0;
170         //if the number is too large, the number would be divided into the smaller one.
171         if (p.compareTo(max) > 0) {
172             p.divide(two);
173             number++;
174         }
175
176         int temP = p.toInt();
177         n.power(temP);
178         while (number > 0) {
179             n.power(2);
180             number--;
181         }
182         //double the number in order to find the original one
183         if (!isEven(power) && power.compareTo(max) > 0) {
184             n.multiply(num);
185         }
186         NaturalNumber temN = n.divide(m);
```

```
187         //construct the new natural number to get the answer
188         n.copyFrom(temN);
189         //get the right answer like the postcondition that is needed.
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 nn = new NaturalNumber2(n);
218         boolean test1 = false;
219         NaturalNumber one = new NaturalNumber2(1);
220         NaturalNumber temW = new NaturalNumber2(w);
221         temW.power(2);
222         NaturalNumber remainder = new NaturalNumber2(temW.divide(n));
223         if (remainder.equals(one)) {
224             test1 = true;
225             //find the answer whether would fulfill the condition 1 or not.
226         } else {
227             nn.decrement();
228             powerMod(w, nn, n);
229             //use the powerMod to find whether the number could fit the test or not.
230             if (!w.equals(one)) {
231                 test1 = true;
232                 //find the answer whether would fulfill the condition 2 or not.
233             }
234         }
235         /*
236          * This line added just to make the program compilable. Should be
237          * replaced with appropriate return statement.
238          */
239         return test1;
240     }
241 
242     /**
243      * Reports whether n is a prime; may be wrong with "low" probability.
244      *
245      * @param n
246      *            number to be checked
247      * @return true means n is very likely prime; false means n is definitely
248      *         composite
```

```
249      * @requires n > 1
250      * @ensures <pre>
251      * isPrime1 = [n is a prime number, with small probability of error
252      *         if it is reported to be prime, and no chance of error if it is
253      *         reported to be composite]
254      * </pre>
255      */
256     public static boolean isPrime1(NaturalNumber n) {
257         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
258         boolean isPrime;
259         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
260             /*
261              * 2 and 3 are primes
262              */
263             isPrime = true;
264         } else if (isEven(n)) {
265             /*
266              * evens are composite
267              */
268             isPrime = false;
269         } else {
270             /*
271              * odd n >= 5: simply check whether 2 is a witness that n is
272              * composite (which works surprisingly well :-)
273              */
274             isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
275         }
276         return isPrime;
277     }
278
279     /**
280      * Reports whether n is a prime; may be wrong with "low" probability.
281      *
282      * @param n
283      *            number to be checked
284      * @return true means n is very likely prime; false means n is definitely
285      *         composite
286      * @requires n > 1
287      * @ensures <pre>
288      * isPrime2 = [n is a prime number, with small probability of error
289      *         if it is reported to be prime, and no chance of error if it is
290      *         reported to be composite]
291      * </pre>
292      */
293     public static boolean isPrime2(NaturalNumber n) {
294         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
295
296         /*
297          * Use the ability to generate random numbers (provided by the
298          * randomNumber method above) to generate several witness candidates --
299          * say, 10 to 50 candidates -- guessing that n is prime only if none of
300          * these candidates is a witness to n being composite (based on fact #3
301          * as described in the project description); use the code for isPrime1
302          * as a guide for how to do this, and pay attention to the requires
303          * clause of isWitnessToCompositeness
304          */
305
306         boolean prime = false;
307         int number = 0;
308         final int randomNumber = 50;
309         //use 50 numbers to find whether it is a right number.
310
```

```
311         NaturalNumber r = new NaturalNumber2(randomNumber(n));
312         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
313
314             prime = true;
315         } else if (isEven(n)) {
316
317             prime = false;
318         } else {
319
320             prime = !isWitnessToCompositeness(r, n);
321             //use the isWitnessToCompositeness to find the answer
322         }
323         number++;
324         if (prime && number < randomNumber) {
325             isPrime2(n);
326         }
327
328         /*
329          * This line added just to make the program compilable. Should be
330          * replaced with appropriate return statement.
331          */
332         return prime;
333
334     }
335
336     /**
337      * Generates a likely prime number at least as large as some given number.
338      *
339      * @param n
340      *            minimum value of likely prime
341      * @updates n
342      * @requires n > 1
343      * @ensures n >= #n and [n is very likely a prime number]
344      */
345     public static void generateNextLikelyPrime(NaturalNumber n) {
346         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
347
348         /*
349          * Use isPrime2 to check numbers, starting at n and increasing through
350          * the odd numbers only (why?), until n is likely prime
351          */
352         while (!isPrime2(n)) {
353             n.increment();
354             //add one for finding the right number n.
355             if (!isEven(n)) {
356                 n.increment();
357             }
358         }
359
360     }
361
362     /**
363      * Main method.
364      *
365      * @param args
366      *            the command line arguments
367      */
368     public static void main(String[] args) {
369         SimpleReader in = new SimpleReader1L();
370         SimpleWriter out = new SimpleWriter1L();
371
372         /*
```

```
373          * Sanity check of randomNumber method -- just so everyone can see how
374          * it might be "tested"
375          */
376         final int testValue = 17;
377         final int testSamples = 100000;
378         NaturalNumber test = new NaturalNumber2(testValue);
379         int[] count = new int[testValue + 1];
380         for (int i = 0; i < count.length; i++) {
381             count[i] = 0;
382         }
383         for (int i = 0; i < testSamples; i++) {
384             NaturalNumber rn = randomNumber(test);
385             assert rn.compareTo(test) <= 0 : "Help!";
386             count[rn.toInt()]++;
387         }
388         for (int i = 0; i < count.length; i++) {
389             out.println("count[" + i + "] = " + count[i]);
390         }
391         out.println("  expected value = "
392                 + (double) testSamples / (double) (testValue + 1));
393 
394         /*
395          * Check user-supplied numbers for primality, and if a number is not
396          * prime, find the next likely prime after it
397          */
398         while (true) {
399             out.print("n = ");
400             NaturalNumber n = new NaturalNumber2(in.nextLine());
401             if (n.compareTo(new NaturalNumber2(2)) < 0) {
402                 out.println("Bye!");
403                 break;
404             } else {
405                 if (isPrime1(n)) {
406                     out.println(n + " is probably a prime number"
407                             + " according to isPrime1.");
408                 } else {
409                     out.println(n + " is a composite number"
410                             + " according to isPrime1.");
411                 }
412                 if (isPrime2(n)) {
413                     out.println(n + " is probably a prime number"
414                             + " according to isPrime2.");
415                 } else {
416                     out.println(n + " is a composite number"
417                             + " according to isPrime2.");
418                     generateNextLikelyPrime(n);
419                     out.println("  next likely prime is " + n);
420                 }
421             }
422         }
423 
424         /*
425          * Close input and output streams
426          */
427         in.close();
428         out.close();
429     }
430 
431 }
432 
```