```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 Vaishnavi Kasabwala
 *
 */
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);
```

```
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
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 wtemp1 = new NaturalNumber2(w);
221         NaturalNumber wtemp2 = new NaturalNumber2(w);
222         NaturalNumber ntemp1 = new NaturalNumber2(n);
223         NaturalNumber ntemp2 = new NaturalNumber2(n);
224         ntemp2.decrement();
225
226         boolean witness = false;
227
228         powerMod(wtemp1, two, ntemp1);
```

```
229         if (wtemp1.compareTo(one) == 0) {
230             witness = true;
231         }
232
233         powerMod(wtemp2, ntemp2, ntemp1);
234         if (wtemp2.compareTo(one) == 0) {
235             witness = false;
236         }
237
238         return witness;
239     }
240
241     /**
242      * Reports whether n is a prime; may be wrong with "low" probability.
243      *
244      * @param n
245      *            number to be checked
246      * @return true means n is very likely prime; false means n is definitely
247      *         composite
248      * @requires n > 1
249      * @ensures <pre>
250      * isPrime1 = [n is a prime number, with small probability of error
251      *         if it is reported to be prime, and no chance of error if it is
252      *         reported to be composite]
253      * </pre>
254      */
255     public static boolean isPrime1(NaturalNumber n) {
256         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
257
258         NaturalNumber two = new NaturalNumber2(2);
259         NaturalNumber three = new NaturalNumber2(3);
260
261         boolean prime;
262
263         if (n.compareTo(three) <= 0) {
264             prime = true;
265         } else if (isEven(n)) {
266             prime = false;
267         } else {
268             prime = !isWitnessToCompositeness(two, n);
269         }
270
271         return prime;
272     }
273
274     /**
275      * Reports whether n is a prime; may be wrong with "low" probability.
276      *
277      * @param n
278      *            number to be checked
279      * @return true means n is very likely prime; false means n is definitely
280      *         composite
281      * @requires n > 1
282      * @ensures <pre>
283      * isPrime2 = [n is a prime number, with small probability of error
284      *         if it is reported to be prime, and no chance of error if it is
285      *         reported to be composite]
```

```
286      * </pre>
287      */
288     public static boolean isPrime2(NaturalNumber n) {
289         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
290
291         /*
292          * Use the ability to generate random numbers (provided by the
293          * randomNumber method above) to generate several witness candidates --
294          * say, 10 to 50 candidates -- guessing that n is prime only if none of
295          * these candidates is a witness to n being composite (based on fact #3
296          * as described in the project description); use the code for isPrime1
297          * as a guide for how to do this, and pay attention to the requires
298          * clause of isWitnessToCompositeness
299          */
300
301         boolean prime = true;
302
303         int i = 1;
304         while (prime == true && i < 20) {
305             NaturalNumber copy = new NaturalNumber2(n);
306             copy.decrement();
307
308             NaturalNumber value = n.newInstance();
309             value.increment();
310
311             NaturalNumber witness = randomNumber(n);
312
313             while (!(witness.compareTo(value) > 0
314                     && witness.compareTo(copy) < 0)) {
315                 witness = randomNumber(n);
316             }
317
318             prime = !isWitnessToCompositeness(witness, n);
319
320             i++;
321         }
322
323         return prime;
324     }
325
326     /**
327      * Generates a likely prime number at least as large as some given number.
328      *
329      * @param n
330      *            minimum value of likely prime
331      * @updates n
332      * @requires n > 1
333      * @ensures n >= #n and [n is very likely a prime number]
334      */
335     public static void generateNextLikelyPrime(NaturalNumber n) {
336         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
337
338         if (isEven(n)) {
339             n.increment();
340         }
341         while (!isPrime2(n)) {
342             n.increment();
```

```
343             n.increment();
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 }
```