```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Micah Casey-Fusco
 *
 */
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 <pre>
     * {@code n > 0}
     * </pre>
     * @ensures <pre>
     *          {@code 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)
```

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

```
122      *            the number to be checked
123      * @return true iff n is even
124      * @ensures <pre> {@code isEven = (n mod 2 = 0)}
125      * </pre>
126      */
127     public static boolean isEven(NaturalNumber n) {
128
129         //create new nn to keep from changing n's value and nn for 2
130         NaturalNumber nCopy = new NaturalNumber2(n);
131         NaturalNumber two = new NaturalNumber2(2);
132
133         //if remainder is 0 when divided by 2, n is even
134         if (nCopy.divide(two).isZero()) {
135             return true;
136         } else {
137             return false;
138         }
139     }
140
141     /**
142      * Updates n to its p-th power modulo m.
143      *
144      * @param n
145      *            number to be raised to a power
146      * @param p
147      *            the power
148      * @param m
149      *            the modulus
150      * @updates n
151      * @requires m > 1
152      * @ensures n = #n ^ (p) mod m
153      */
154     public static void powerMod(NaturalNumber n, NaturalNumber p,
155             NaturalNumber m) {
156         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
157
158         /*
159          * Use the fast-powering algorithm as previously discussed in class,
160          * with the additional feature that every multiplication is followed
161          * immediately by "reducing the result modulo m"
162          */
163
164         // new NN for use in the fast-powering algorithm
165         NaturalNumber powerNum = new NaturalNumber2(1);
166
167         // use algorithm till the p-th power (reduce to mod m every iteration)
168         for (int i = 0; i < p.toInt(); i++) {
169             powerNum.multiply(n);
170             powerNum = powerNum.divide(m);
171         }
172
173         //steal value from powerNum to update n and save storage space
174         n.transferFrom(powerNum);
175     }
176
177     /**
178      * Reports whether w is a "witness" that n is composite, in the sense that
```

```
179      * either it is a square root of 1 (mod n), or it fails to satisfy the
180      * criterion for primality from Fermat's theorem.
181      *
182      * @param w
183      *            witness candidate
184      * @param n
185      *            number being checked
186      * @return true iff w is a "witness" that n is composite
187      * @requires n > 2 and 1 < w < n - 1
188      * @ensures <pre>
189      * isWitnessToCompositeness =
190      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
191      * </pre>
192      */
193     public static boolean isWitnessToCompositeness(NaturalNumber w,
194             NaturalNumber n) {
195         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
196         assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 < w";
197         n.decrement();
198         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
199         n.increment();
200
201         //parameter copies & temporary variables to prevent alias conflicts
202         NaturalNumber ensures1 = new NaturalNumber2();
203         NaturalNumber ensures2 = new NaturalNumber2();
204         NaturalNumber wCopy = new NaturalNumber2(w);
205         NaturalNumber wCopy2 = new NaturalNumber2(w);
206         NaturalNumber one = new NaturalNumber2(1);
207         int nMinus = 0;
208
209         //first ensures clause (w ^ 2 mod n = 1)
210         wCopy.power(2);
211         ensures1 = wCopy.divide(n);
212
213         //assign n-1 for next step
214         n.decrement();
215         nMinus = n.toInt();
216         n.increment();
217
218         //second ensures clause (w ^ (n-1) mod n /= 1)}
219         wCopy2.power(nMinus);
220         ensures2 = wCopy2.divide(n);
221
222         //checking compositeness and returning the corresponding boolean
223         if (ensures1.equals(one) || !ensures2.equals(one)) {
224             return true;
225         } else {
226             return false;
227         }
228     }
229
230     /**
231      * Reports whether n is a prime; may be wrong with "low" probability.
232      *
233      * @param n
234      *            number to be checked
235      * @return true means n is very likely prime; false means n is definitely
```

```
236      *         composite
237      * @requires n > 1
238      * @ensures <pre>
239      * isPrime1 = [n is a prime number, with small probability of error
240      *         if it is reported to be prime, and no chance of error if it is
241      *         reported to be composite]
242      * </pre>
243      */
244     public static boolean isPrime1(NaturalNumber n) {
245         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
246         boolean isPrime;
247         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
248             /*
249              * 2 and 3 are primes
250              */
251             isPrime = true;
252         } else if (isEven(n)) {
253             /*
254              * evens are composite
255              */
256             isPrime = false;
257         } else {
258             /*
259              * odd n >= 5: simply check whether 2 is a witness that n is
260              * composite (which works surprisingly well :-)
261              */
262             isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
263         }
264         return isPrime;
265     }
266
267     /**
268      * Reports whether n is a prime; may be wrong with "low" probability.
269      *
270      * @param n
271      *            number to be checked
272      * @return true means n is very likely prime; false means n is definitely
273      *         composite
274      * @requires <pre>
275      * {@code n > 1}
276      * </pre>
277      * @ensures <pre>
278      * {@code isPrime1 = [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         //initialize for later
297         NaturalNumber one = new NaturalNumber2(1);
298         NaturalNumber random = new NaturalNumber2(1);
299
300         // looping 50 times to check for witnesses
301         for (int i = 0; i < 50; i++) {
302
303             // generate random number between 1 and n (non-inclusive)
304             while (!(one.compareTo(random) < 0)) {
305                 n.decrement();
306                 n.decrement();
307                 random = randomNumber(n);
308                 n.increment();
309                 n.increment();
310
311             }
312
313             //check if random number is a witness
314             System.out.println(random);
315             if (isWitnessToCompositeness(random, n)) {
316                 return false;
317             }
318
319         }
320         return true;
321     }
322
323     /**
324      * Generates a likely prime number at least as large as some given number.
325      *
326      * @param n
327      *            minimum value of likely prime
328      * @updates n
329      * @requires <pre>
330      * {@code n > 1}
331      * </pre>
332      * @ensures <pre>
333      * {@code n >= #n  and  [n is very likely a prime number]}
334      * </pre>
335      */
336     public static void generateNextLikelyPrime(NaturalNumber n) {
337         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
338
339         /*
340          * Use isPrime2 to check numbers, starting at n and increasing through
341          * the odd numbers only (why?), until n is likely prime
342          */
343
344         while (!isPrime2(n)) {
345             if (isEven(n)) {
346                 n.increment();
347             } else {
348                 n.increment();
349                 n.increment();
```

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

```
407                 } else {
408                     out.println(n + " is a composite number"
409                             + " according to isPrime2.");
410                     generateNextLikelyPrime(n);
411                     out.println("  next likely prime is " + n);
412                 }
413             }
414         }
415 
416         /*
417          * Close input and output streams
418          */
419         in.close();
420         out.close();
421     }
422 
423 }
424 //holy sh*t im never leaving a project for the night before again :,) its 6am
425 
```