```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Jane Weissberg
 *
 */
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;
```

```
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
101         // Base case: if m is zero, n is already the GCD
102         if (!m.isZero()) {
103             // Calculate remainder of n divided by m
104             NaturalNumber remainder = n.divide(m);
105
106             reduceToGCD(m, remainder);
107
108             // Copy m into n since m is the GCD now
109             n.copyFrom(m);
110
111             // Clear m to satisfy contract
112             m.clear();
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         // Divide n by 10 and store remainder in lastDigit
128         int lastDigit = n.divideBy10();
129
130         // Restore n
131         n.multiplyBy10(lastDigit);
132
133         // Return true if last digit is even and false if it is
    odd
134         return (lastDigit % 2 == 0);
135
136     }
137
138     /**
139      * Updates n to its p-th power modulo m.
140      *
141      * @param n
142      *            number to be raised to a power
143      * @param p
144      *            the power
145      * @param m
146      *            the modulus
147      * @updates n
148      * @requires m > 1
149      * @ensures n = #n ^ (p) mod m
150      */
151     public static void powerMod(NaturalNumber n, NaturalNumber p,
    NaturalNumber m) {
152         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: m > 1";
153
154         /*
155          * Use the fast-powering algorithm as previously discussed
    in class,
156          * with the additional feature that every multiplication
    is followed
157          * immediately by "reducing the result modulo m"
158          */
159
160         // TODO - fill in body
161         final NaturalNumber two = new NaturalNumber2(2);
162         NaturalNumber originalN = new NaturalNumber2(n);
163
164         // Base case: if p is 0, set n to 1
```

```
165         if (p.isZero()) {
166             n.setFromInt(1);
167         } else {
168 
169             // Calculate p / 2 and store it in halfPower
170             NaturalNumber halfPower = new NaturalNumber2();
171             halfPower.copyFrom(p);
172             halfPower.divide(two);
173 
174             // Recursion to raise n to the half power mod m
175             powerMod(n, halfPower, m);
176 
177             // Multiply n by itself then do mod m
178             NaturalNumber copy = new NaturalNumber2(n);
179             copy.multiply(n);
180             copy.transferFrom(copy.divide(m));
181 
182             if (p.divide(two).isZero()) {
183                 n.copyFrom(copy);
184             } else { // If p is odd, multiply the result by n
    again
185                 copy.multiply(originalN);
186                 copy.transferFrom(copy.divide(m));
187                 n.copyFrom(copy);
188             }
189 
190             // Restore p
191             p.multiply(new NaturalNumber2(2));
192         }
193 
194     }
195 
196     /**
197      * Reports whether w is a "witness" that n is composite, in
    the sense that
198      * either it is a square root of 1 (mod n), or it fails to
    satisfy the
199      * criterion for primality from Fermat's theorem.
200      *
201      * @param w
202      *            witness candidate
203      * @param n
204      *            number being checked
205      * @return true iff w is a "witness" that n is composite
```

```
206      * @requires n > 2 and 1 < w < n - 1
207      * @ensures <pre>
208      * isWitnessToCompositeness =
209      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
210      * </pre>
211      */
212     public static boolean isWitnessToCompositeness(NaturalNumber
    w, NaturalNumber n) {
213         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation
    of: n > 2";
214         assert (new NaturalNumber2(1)).compareTo(w) < 0 :
    "Violation of: 1 < w";
215         n.decrement();
216         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
217         n.increment();
218
219         boolean isWitness = false;
220         NaturalNumber w2 = new NaturalNumber2(0);
221
222         w2.copyFrom(w);
223
224         // Check if w^2 mod n == 1 (potential witness)
225         powerMod(w, new NaturalNumber2(2), n);
226         if (w.compareTo(new NaturalNumber2(1)) == 0) {
227             // Set to true if it is a witness
228             isWitness = true;
229         }
230
231         // Restore w
232         w.copyFrom(w2);
233
234         // Check if w^(n-1) mod n != 1 (failure)
235         NaturalNumber n2 = new NaturalNumber2(n);
236         n2.decrement();
237         powerMod(w, n2, n);
238         if (w.compareTo(new NaturalNumber2(1)) != 0) {
239
240             // If w^(n-1) mod n is not 1, set isWitness to true
241             isWitness = true;
242         }
243
244         // Restore w
245         w.copyFrom(w2);
246         return isWitness;
```

```
247     }
248
249     /**
250      * Reports whether n is a prime; may be wrong with "low"
    probability.
251      *
252      * @param n
253      *            number to be checked
254      * @return true means n is very likely prime; false means n is
    definitely
255      *         composite
256      * @requires n > 1
257      * @ensures <pre>
258      * isPrime1 = [n is a prime number, with small probability of
    error
259      *         if it is reported to be prime, and no chance of
    error if it is
260      *         reported to be composite]
261      * </pre>
262      */
263     public static boolean isPrime1(NaturalNumber n) {
264         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
265         boolean isPrime;
266         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
267             /*
268              * 2 and 3 are primes
269              */
270             isPrime = true;
271         } else if (isEven(n)) {
272             /*
273              * evens are composite
274              */
275             isPrime = false;
276         } else {
277             /*
278              * odd n >= 5: simply check whether 2 is a witness
    that n is
279              * composite (which works surprisingly well :-)
280              */
281             isPrime = !isWitnessToCompositeness(new
    NaturalNumber2(2), n);
282         }
283         return isPrime;
```

```
284     }
285
286     /**
287      * Reports whether n is a prime; may be wrong with "low"
    probability.
288      *
289      * @param n
290      *            number to be checked
291      * @return true means n is very likely prime; false means n is
    definitely
292      *         composite
293      * @requires n > 1
294      * @ensures <pre>
295      * isPrime2 = [n is a prime number, with small probability of
    error
296      *         if it is reported to be prime, and no chance of
    error if it is
297      *         reported to be composite]
298      * </pre>
299      */
300     public static boolean isPrime2(NaturalNumber n) {
301         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
302
303         /*
304          * Use the ability to generate random numbers (provided by
    the
305          * randomNumber method above) to generate several witness
    candidates --
306          * say, 10 to 50 candidates -- guessing that n is prime
    only if none of
307          * these candidates is a witness to n being composite
    (based on fact #3
308          * as described in the project description); use the code
    for isPrime1
309          * as a guide for how to do this, and pay attention to the
    requires
310          * clause of isWitnessToCompositeness
311          */
312
313         boolean isPrime;
314         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
315             /*
316              * 2 and 3 are primes
```

```
317             */
318             isPrime = true;
319         } else if (isEven(n)) {
320             /*
321              * evens are composite
322              */
323             isPrime = false;
324         } else {
325             /*
326              * odd n >= 5: simply check whether 2 is a witness
    that n is
327              * composite (which works surprisingly well :-)
328              */
329             int j = 0;
330             isPrime = true;
331             final int limit = 49;
332 
333             // Loop until isPrime is true and the counter is below
    the limit
334             while (isPrime && j < limit) {
335 
336                 n.subtract(new NaturalNumber2(2));
337                 NaturalNumber random = randomNumber(n);
338 
339                 // Ensure random is within the range 2 < random <
    n - 1
340                 while (random.compareTo(new NaturalNumber2(2)) <
    0) {
341                     random = randomNumber(n);
342                 }
343 
344                 // Restore n
345                 n.add(new NaturalNumber2(2));
346 
347                 // Check if 'random' is a witness to n's
    compositeness
348                 isPrime = !isWitnessToCompositeness(random, n);
349                 j++;
350             }
351         }
352         return isPrime;
353     }
354 
355     /**
```

```
356      * Generates a likely prime number at least as large as some
    given number.
357      *
358      * @param n
359      *            minimum value of likely prime
360      * @updates n
361      * @requires n > 1
362      * @ensures n >= #n and [n is very likely a prime number]
363      */
364     public static void generateNextLikelyPrime(NaturalNumber n) {
365         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
366
367         /*
368          * Use isPrime2 to check numbers, starting at n and
    increasing through
369          * the odd numbers only (why?), until n is likely prime
370          */
371
372         // TODO - fill in body
373         boolean result = false;
374         NaturalNumber two = new NaturalNumber2(2);
375
376         if (isEven(n)) {
377             n.increment();
378         } else {
379             n.add(two);
380         }
381
382         while (!result) {
383             if (isPrime2(n)) {
384                 result = true;
385             } else {
386                 n.add(two);
387             }
388         }
389
390     }
391
392     /**
393      * Main method.
394      *
395      * @param args
396      *            the command line arguments
```

```
397      */
398     public static void main(String[] args) {
399         SimpleReader in = new SimpleReader1L();
400         SimpleWriter out = new SimpleWriter1L();
401
402         /*
403          * Sanity check of randomNumber method -- just so everyone
    can see how
404          * it might be "tested"
405          */
406         final int testValue = 17;
407         final int testSamples = 100000;
408         NaturalNumber test = new NaturalNumber2(testValue);
409         int[] count = new int[testValue + 1];
410         for (int i = 0; i < count.length; i++) {
411             count[i] = 0;
412         }
413         for (int i = 0; i < testSamples; i++) {
414             NaturalNumber rn = randomNumber(test);
415             assert rn.compareTo(test) <= 0 : "Help!";
416             count[rn.toInt()]++;
417         }
418         for (int i = 0; i < count.length; i++) {
419             out.println("count[" + i + "] = " + count[i]);
420         }
421         out.println(
422                 "  expected value = " + (double) testSamples /
    (double) (testValue + 1));
423
424         /*
425          * Check user-supplied numbers for primality, and if a
    number is not
426          * prime, find the next likely prime after it
427          */
428         while (true) {
429             out.print("n = ");
430             NaturalNumber n = new NaturalNumber2(in.nextLine());
431             if (n.compareTo(new NaturalNumber2(2)) < 0) {
432                 out.println("Bye!");
433                 break;
434             } else {
435                 if (isPrime1(n)) {
436                     out.println(n + " is probably a prime number"
437                             + " according to isPrime1.");
```

```
438                 } else {
439                     out.println(n + " is a composite number" + "
    according to isPrime1.");
440                 }
441                 if (isPrime2(n)) {
442                     out.println(n + " is probably a prime number"
443                             + " according to isPrime2.");
444                 } else {
445                     out.println(n + " is a composite number" + "
    according to isPrime2.");
446                     generateNextLikelyPrime(n);
447                     out.println("  next likely prime is " + n);
448                 }
449             }
450         }
451 
452         /*
453          * Close input and output streams
454          */
455         in.close();
456         out.close();
457     }
458 
459 }
460 
```