```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Vishal Kumar
 *
 */
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)
             */
            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           * Use Euclid's algorithm; in pseudocode: if m = 0 then GCD(n, m) = n
97           * else GCD(n, m) = GCD(m, n mod m)
98           */
99
100         if (!(m.isZero())) {
101             // get the n mod m
102             NaturalNumber mod = new NaturalNumber2(n.divide(m));
103             // pass it into method
104             reduceToGCD(m, mod);
105             // set n equal to gcd
106             n.copyFrom(m);
107         }
108         // clear m
109         m.clear();
110     }
111
112     /**
113      * Reports whether n is even.
114      *
115      * @param n
116      *            the number to be checked
117      * @return true iff n is even
118      * @ensures isEven = (n mod 2 = 0)
119      */
120     public static boolean isEven(NaturalNumber n) {
121         // natural number to store mod
```

```
122         NaturalNumber mod = n.newInstance();
123         // natural number constant : 2
124         NaturalNumber two = n.newInstance();
125         two.setFromInt(2);
126         // copy of n for division
127         NaturalNumber nCopy = n.newInstance();
128         nCopy.copyFrom(n);
129
130         // set mod equal to n/2
131         mod.add(nCopy.divide(two));
132         return mod.isZero();
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         // natural number constants
153         NaturalNumber two = new NaturalNumber2(2);
154         NaturalNumber zero = new NaturalNumber2(0);
155         NaturalNumber one = new NaturalNumber2(1);
156         // p/2
157         NaturalNumber halfP = p.newInstance();
158         halfP.copyFrom(p);
159         halfP.divide(two);
160
161         // copy of and n
162         NaturalNumber nCopy = n.newInstance();
163         nCopy.add(n);
164
165         if (p.isZero()) {
166             n.clear();
167             n.add(one);
168         } else if (n.isZero()) {
169             n.clear();
170             n.add(zero);
171
172         } else {
173             if (isEven(p)) {
174                 // multiply n by itself
175                 n.multiply(nCopy);
176
177                 // take n to the power of p/2
178                 powerMod(n, halfP, m);
```

```
179             } else {
180                 // multiply n by itself
181                 n.multiply(nCopy);
182                 // take n to the power of p/2
183                 powerMod(n, halfP, m);
184                 // multiply n by itself again
185                 n.multiply(nCopy);
186             }
187             // n mod m
188             nCopy.copyFrom(n);
189             n.transferFrom(nCopy.divide(m));
190
191         }
192
193     }
194
195     /**
196      * Reports whether w is a "witness" that n is composite, in the sense that
197      * either it is a square root of 1 (mod n), or it fails to satisfy the
198      * criterion for primality from Fermat's theorem.
199      *
200      * @param w
201      *            witness candidate
202      * @param n
203      *            number being checked
204      * @return true iff w is a "witness" that n is composite
205      * @requires n > 2 and 1 < w < n - 1
206      * @ensures <pre>
207      * isWitnessToCompositeness =
208      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
209      * </pre>
210      */
211     public static boolean isWitnessToCompositeness(NaturalNumber w,
212             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         //declare constants
220         NaturalNumber two = new NaturalNumber2(2);
221
222         NaturalNumber one = new NaturalNumber2(1);
223
224         // t/f variable
225         boolean decision = false;
226
227         // copy of w for case 2
228         NaturalNumber wCopy = w.newInstance();
229         wCopy.add(w);
230         // power for case 2
231         NaturalNumber p = n.newInstance();
232         p.add(n);
233         p.decrement();
234
235         powerMod(w, two, n);
```

```
236         powerMod(wCopy, p, n);
237
238         if (w.compareTo(one) == 0 || wCopy.compareTo(one) != 0) {
239             decision = true;
240         }
241         return decision;
242     }
243
244     /**
245      * Reports whether n is a prime; may be wrong with "low" probability.
246      *
247      * @param n
248      *            number to be checked
249      * @return true means n is very likely prime; false means n is definitely
250      *         composite
251      * @requires n > 1
252      * @ensures <pre>
253      * isPrime1 = [n is a prime number, with small probability of error
254      *         if it is reported to be prime, and no chance of error if it is
255      *         reported to be composite]
256      * </pre>
257      */
258     public static boolean isPrime1(NaturalNumber n) {
259         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
260         boolean isPrime;
261         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
262             /*
263              * 2 and 3 are primes
264              */
265             isPrime = true;
266         } else if (isEven(n)) {
267             /*
268              * evens are composite
269              */
270             isPrime = false;
271         } else {
272             /*
273              * odd n >= 5: simply check whether 2 is a witness that n is
274              * composite (which works surprisingly well :-)
275              */
276             isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
277         }
278         return isPrime;
279     }
280
281     /**
282      * Reports whether n is a prime; may be wrong with "low" probability.
283      *
284      * @param n
285      *            number to be checked
286      * @return true means n is very likely prime; false means n is definitely
287      *         composite
288      * @requires n > 1
289      * @ensures <pre>
290      * isPrime2 = [n is a prime number, with small probability of error
291      *         if it is reported to be prime, and no chance of error if it is
292      *         reported to be composite]
```

```
293      * </pre>
294      */
295     public static boolean isPrime2(NaturalNumber n) {
296         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
297
298         /*
299          * Use the ability to generate random numbers (provided by the
300          * randomNumber method above) to generate several witness candidates --
301          * say, 10 to 50 candidates -- guessing that n is prime only if none of
302          * these candidates is a witness to n being composite (based on fact #3
303          * as described in the project description); use the code for isPrime1
304          * as a guide for how to do this, and pay attention to the requires
305          * clause of isWitnessToCompositeness
306          */
307         NaturalNumber two = n.newInstance();
308         two.setFromInt(2);
309         boolean isPrime;
310         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
311             // obvious case
312             isPrime = true;
313         } else if (isEven(n)) {
314             // if n is even then it can't be prime
315             isPrime = false;
316         } else {
317             int witnessCount = 0;
318             NaturalNumber nMinusTwo = n.newInstance();
319             nMinusTwo.add(n);
320             nMinusTwo.subtract(two);
321             nMinusTwo.subtract(two);
322
323             // generate 30 random numbers from (2, n-2)
324             //and check to see if they are WitnessToCompositeness
325             for (int i = 0; i < 30; i++) {
326                 NaturalNumber random = new NaturalNumber2(
327                         randomNumber(nMinusTwo));
328                 random.add(two);
329                 if (isWitnessToCompositeness(random, n)) {
330                     witnessCount++;
331                 }
332             }
333             // if a witnessToCompositness was generated then n is not prime
334             if (witnessCount > 0) {
335                 isPrime = false;
336             } else {
337                 isPrime = true;
338             }
339         }
340         return isPrime;
341     }
342
343     /**
344      * Generates a likely prime number at least as large as some given number.
345      *
346      * @param n
347      *            minimum value of likely prime
348      * @updates n
349      * @requires n > 1
```

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

```
407             if (n.compareTo(new NaturalNumber2(2)) < 0) {
408                 out.println("Bye!");
409                 break;
410             } else {
411                 if (isPrime1(n)) {
412                     out.println(n + " is probably a prime number"
413                             + " according to isPrime1.");
414                 } else {
415                     out.println(n + " is a composite number"
416                             + " according to isPrime1.");
417                 }
418                 if (isPrime2(n)) {
419                     out.println(n + " is probably a prime number"
420                             + " according to isPrime2.");
421                 } else {
422                     out.println(n + " is a composite number"
423                             + " according to isPrime2.");
424                     generateNextLikelyPrime(n);
425                     out.println("  next likely prime is " + n);
426                 }
427             }
428         }
429 
430         /*
431          * Close input and output streams
432          */
433         in.close();
434         out.close();
435     }
436 
437 }
```