```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 Max Timko.44
 *
 */
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;
```

```
48          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         NaturalNumber mod = n.newInstance();
 97         mod.copyFrom(n);
 98         //Adam keeps talking about recursion for this method on piazza
 99         //I didn't see any mention of it in the instructions
100
101         if (!m.isZero()) {
102             while (!mod.isZero()) {
103                 mod = n.divide(m);
104                 n.copyFrom(m);
105                 m.copyFrom(mod);
106
107             }
108         }
109
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
122         int i = n.divideBy10();
123         n.multiplyBy10(i);
124
125         return (i % 2 == 0);
126     }
127
128     /**
129      * Updates n to its p-th power modulo m.
130      *
131      * @param n
132      *            number to be raised to a power
133      * @param p
134      *            the power
135      * @param m
136      *            the modulus
137      * @updates n
138      * @requires m > 1
139      * @ensures n = #n ^ (p) mod m
140      */
141     public static void powerMod(NaturalNumber n, NaturalNumber p,
```

```
142             NaturalNumber m) {
143         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
144
145         //If we need to do more than n mod m
146         if (p.compareTo(new NaturalNumber2(1)) > 0) {
147
148             //saving values before change
149             String pString = p.toString();
150             String nString = n.toString();
151
152             boolean odd = !isEven(p);
153
154             //making the problem smaller
155             p.divide(new NaturalNumber2(2));
156
157             //cursing and restoring p
158             powerMod(n, p, m);
159             p.setFromString(pString);
160
161             /*
162              * we can know that smaller problems will have equal halves p = (p
    ^
163              * 1/2) ^ 2 (p ^ 1/2) = (p ^ 1/2)
164              */
165             n.multiply(new NaturalNumber2(n));
166
167             //if it was odd, multiply in an additional n mod m
168             if (odd) {
169                 NaturalNumber r = new NaturalNumber2(nString);
170                 n.multiply(r.divide(m));
171             }
172
173         }
174
175         /*
176          * I did look at writing: n = n.divide(m) I stopped getting correct
177          * answers when that was tested. I assume it had something to do with
178          * aliasing somewhere.
179          *
180          * All calls of powerMod should return an n mod m variant
181          */
182         n.copyFrom(n.divide(m));
183
184         //handles the 0 border case
185         if (p.isZero()) {
186             n.setFromInt(1);
187         }
```

```
188
189     }
190
191     /**
192      * Reports whether w is a "witness" that n is composite, in the sense that
193      * it fails to satisfy the (w ^ (n-1) mod n /= 1) criterion for primality
194      * from Fermat's theorem.
195      *
196      * @param w
197      *            witness candidate
198      * @param n
199      *            number being checked
200      * @return true iff w is a "witness" that n is composite
201      * @requires n > 2 and 1 < w < n - 1
202      * @ensures <pre>
203      * isWitnessToCompositeness =
204      *     (w ^ (n-1) mod n /= 1)
205      * </pre>
206      */
207     public static boolean isWitnessToCompositeness(NaturalNumber w,
208             NaturalNumber n) {
209         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
210         assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 <
    w";
211         n.decrement();
212         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
213         n.increment();
214
215         /*
216          * (w ^ (n-1) mod n /= 1)
217          */
218         NaturalNumber oneMinusN = new NaturalNumber2(n);
219         oneMinusN.decrement();
220
221         String witnessProtection = w.toString();
222         powerMod(w, oneMinusN, n);
223         boolean ans = (w.compareTo(new NaturalNumber2(1)) != 0);
224         w.setFromString(witnessProtection);
225
226         return ans;
227     }
228
229     /**
230      * Reports whether n is a prime; may be wrong with "low" probability.
231      *
232      * @param n
233      *            number to be checked
```

```
234      * @return true means n is very likely prime; false means n is definitely
235      *         composite
236      * @requires n > 1
237      * @ensures <pre>
238      * isPrime1 = [n is a prime number, with small probability of error
239      *         if it is reported to be prime, and no chance of error if it is
240      *         reported to be composite]
241      * </pre>
242      */
243     public static boolean isPrime1(NaturalNumber n) {
244         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
245         boolean isPrime;
246         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
247             /*
248              * 2 and 3 are primes
249              */
250             isPrime = true;
251         } else if (isEven(n)) {
252             /*
253              * evens are composite
254              */
255             isPrime = false;
256         } else {
257             /*
258              * odd n >= 5: simply check whether 2 is a witness that n is
259              * composite (which works surprisingly well :-)
260              */
261             NaturalNumber t = new NaturalNumber2(2);
262             isPrime = !isWitnessToCompositeness(t, 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 n > 1
275      * @ensures <pre>
276      * isPrime2 = [n is a prime number, with small probability of error
277      *         if it is reported to be prime, and no chance of error if it is
278      *         reported to be composite]
279      * </pre>
280      */
```

```
281     public static boolean isPrime2(NaturalNumber n) {
282         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
283
284         //default prime until proven false == sometimes false positives
285         boolean isPrime = true;
286
287         NaturalNumber canIGetAWitness = randomNumber(n);
288
289         NaturalNumber oneMinusN = new NaturalNumber2(n);
290         oneMinusN.decrement();
291
292         for (int i = 0; i < 50; i++) {
293             if (canIGetAWitness.compareTo(new NaturalNumber2(2)) > 0
294                     && canIGetAWitness.compareTo(oneMinusN) < 0) {
295
296                 if (isWitnessToCompositeness(canIGetAWitness, n)) {
297                     isPrime = false;
298                 }
299
300             }
301             canIGetAWitness = randomNumber(n);
302         }
303
304         return isPrime;
305     }
306
307     /**
308      * Reports whether n is a prime; may be wrong with "lower" probability.
309      *
310      * @param n
311      *            number to be checked
312      * @return true means n is very likely prime; false means n is definitely
313      *         composite
314      * @requires n > 1
315      * @ensures <pre>
316      * isPrime3 = [n is a prime number, with small probability of error
317      *         if it is reported to be prime, and no chance of error if it is
318      *         reported to be composite]
319      * </pre>
320      */
321     public static boolean isPrime3(NaturalNumber n) {
322         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
323
324         boolean isPrime = true;
325
326         if (isEven(n)) {
327             isPrime = false;
```

```
328         } else {
329             NaturalNumber canIGetAWitness = randomNumber(n);
330 
331             NaturalNumber oneMinusN = new NaturalNumber2(n);
332             oneMinusN.decrement();
333 
334             NaturalNumber d = new NaturalNumber2(n);
335 
336             d.divide(new NaturalNumber2(2));
337 
338             int i = 0;
339             while (isPrime && i < 50) {
340                 if (canIGetAWitness.compareTo(new NaturalNumber2(2)) > 0
341                         && canIGetAWitness.compareTo(oneMinusN) < 0) {
342 
343                     //(w ^ (n/2) ) mod n != +/-1 mod n
344                     powerMod(canIGetAWitness, d, n);
345                     if (!(canIGetAWitness.compareTo(new NaturalNumber2(1)) ==
    0
346                             || canIGetAWitness.compareTo(oneMinusN) == 0)) {
347                         isPrime = false;
348                     }
349 
350                 }
351                 canIGetAWitness = randomNumber(n);
352 
353                 i++;
354             }
355 
356         }
357         return isPrime;
358     }
359 
360     /**
361      * Generates a likely prime number at least as large as some given number.
362      *
363      * @param n
364      *            minimum value of likely prime
365      * @updates n
366      * @requires n > 1
367      * @ensures n >= #n and [n is very likely a prime number]
368      */
369     public static void generateNextLikelyPrime(NaturalNumber n) {
370         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
371 
372         if (isEven(n)) {
373             n.decrement();
```

```
374         }
375         n.increment();
376         n.increment();
377         while (!isPrime2(n)) {
378             n.increment();
379             n.increment();
380         }
381
382     }
383
384     /**
385      * Main method.
386      *
387      * @param args
388      *            the command line arguments
389      */
390     public static void main(String[] args) {
391         SimpleReader in = new SimpleReader1L();
392         SimpleWriter out = new SimpleWriter1L();
393
394         /*
395          * Sanity check of randomNumber method -- just so everyone can see how
396          * it might be "tested"
397          */
398         final int testValue = 17;
399         final int testSamples = 100000;
400         NaturalNumber test = new NaturalNumber2(testValue);
401         int[] count = new int[testValue + 1];
402         for (int i = 0; i < count.length; i++) {
403             count[i] = 0;
404         }
405         for (int i = 0; i < testSamples; i++) {
406             NaturalNumber rn = randomNumber(test);
407             assert rn.compareTo(test) <= 0 : "Help!";
408             count[rn.toInt()]++;
409         }
410         for (int i = 0; i < count.length; i++) {
411             out.println("count[" + i + "] = " + count[i]);
412         }
413         out.println("  expected value = "
414                 + (double) testSamples / (double) (testValue + 1));
415
416         /*
417          * Check user-supplied numbers for primality, and if a number is not
418          * prime, find the next likely prime after it
419          */
420         while (true) {
```

```
421             out.print("n = ");
422             NaturalNumber n = new NaturalNumber2(in.nextLine());
423             if (n.compareTo(new NaturalNumber2(2)) < 0) {
424                 out.println("Bye!");
425                 break;
426             } else {
427                 if (isPrime1(n)) {
428                     out.println(n + " is probably a prime number"
429                             + " according to isPrime1.");
430                 } else {
431                     out.println(n + " is a composite number"
432                             + " according to isPrime1.");
433                 }
434                 if (isPrime2(n)) {
435                     out.println(n + " is probably a prime number"
436                             + " according to isPrime2.");
437                 } else {
438                     out.println(n + " is a composite number"
439                             + " according to isPrime2.");
440 
441                 }
442                 if (isPrime3(n)) {
443                     out.println(n + " is probably a prime number"
444                             + " according to isPrime3.");
445                 } else {
446                     out.println(n + " is a composite number"
447                             + " according to isPrime3.");
448                     generateNextLikelyPrime(n);
449                     out.println("  next likely prime is " + n);
450                 }
451             }
452         }
453 
454         /*
455          * Close input and output streams
456          */
457         in.close();
458         out.close();
459     }
460 
461 }
```