```
 1 import components.naturalnumber.NaturalNumber;
 2 import components.naturalnumber.NaturalNumber2;
 3 import components.random.Random;
 4 import components.random.Random1L;
 5 import components.simplereader.SimpleReader;
 6 import components.simplereader.SimpleReader1L;
 7 import components.simplewriter.SimpleWriter;
 8 import components.simplewriter.SimpleWriter1L;
 9
10 /**
11  * Utilities that could be used with RSA cryptosystems.
12  *
13  * @author Ansh Pachauri
14  *
15  */
16 public final class CryptoUtilities {
17
18     /**
19      * Private constructor so this utility class cannot be
    instantiated.
20      */
21     private CryptoUtilities() {
22     }
23
24     /**
25      * Useful constant, not a magic number: 3.
26      */
27     private static final int THREE = 3;
28
29     /**
30      * Pseudo-random number generator.
31      */
32     private static final Random GENERATOR = new Random1L();
33
34     /**
35      * Returns a random number uniformly distributed in the
    interval [0, n].
36      *
37      * @param n
```

```
38      *            top end of interval
39      * @return random number in interval
40      * @requires n > 0
41      * @ensures <pre>
42      * randomNumber = [a random number uniformly distributed in
   [0, n]]
43      * </pre>
44      */
45     public static NaturalNumber randomNumber(NaturalNumber n) {
46         assert !n.isZero() : "Violation of: n > 0";
47         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         /*
 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         // TODO - fill in body
102         NaturalNumber zero = new NaturalNumber2(0);
```

```
103         if (!m.isZero()) {
104             // Create new natural number k to hold value of n
    mod m
105             NaturalNumber k = new NaturalNumber2(n.divide(m));
106             reduceToGCD(m, k);
107             // transferFrom m in order to update n and clear m
108             n.transferFrom(m);
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         // TODO - fill in body
123         NaturalNumber temp = new NaturalNumber2(n);
124         NaturalNumber two = new NaturalNumber2(2);
125         boolean choice = false;
126         //checking if n mod 2 is equal to 0
127         if (temp.divide(two).isZero()) {
128             choice = true;
129         }
130
131         return choice;
132     }
133
134     /**
135      * Updates n to its p-th power modulo m.
136      *
137      * @param n
138      *            number to be raised to a power
139      * @param p
140      *            the power
```

```
141     * @param m
142     *            the modulus
143     * @updates n
144     * @requires m > 1
145     * @ensures n = #n ^ (p) mod m
146     */
147    public static void powerMod(NaturalNumber n, NaturalNumber
   p,
148            NaturalNumber m) {
149        assert m.compareTo(new NaturalNumber2(1)) > 0 :
   "Violation of: m > 1";
150
151        /*
152         * Use the fast-powering algorithm as previously
   discussed in class,
153         * with the additional feature that every
   multiplication is followed
154         * immediately by "reducing the result modulo m"
155         */
156        NaturalNumber one = new NaturalNumber2(1);
157        NaturalNumber two = new NaturalNumber2(2);
158        NaturalNumber nTemp = new NaturalNumber2(n);
159        NaturalNumber pTemp = new NaturalNumber2(p);
160
161        if (p.isZero()) {
162            n.copyFrom(one);
163        } else {
164            pTemp.divide(two);
165            powerMod(n, pTemp, m);
166            n.multiply(new NaturalNumber2(n));
167            if (!isEven(p)) {
168                n.multiply(nTemp);
169            }
170            n.transferFrom(n.divide(m));
171        }
172    }
173
174    /**
175     * Reports whether w is a "witness" that n is composite, in
```

```
   the sense that
176     * either it is a square root of 1 (mod n), or it fails to
   satisfy the
177     * criterion for primality from Fermat's theorem.
178     *
179     * @param w
180     *            witness candidate
181     * @param n
182     *            number being checked
183     * @return true iff w is a "witness" that n is composite
184     * @requires n > 2 and 1 < w < n - 1
185     * @ensures <pre>
186     * isWitnessToCompositeness =
187     *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
188     * </pre>
189     */
190    public static boolean
   isWitnessToCompositeness(NaturalNumber w,
191            NaturalNumber n) {
192        assert n.compareTo(new NaturalNumber2(2)) > 0 :
   "Violation of: n > 2";
193        assert (new NaturalNumber2(1)).compareTo(w) < 0 :
   "Violation of: 1 < w";
194        n.decrement();
195        assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
196        n.increment();
197
198        // TODO - fill in body
199        boolean witness = false;
200        NaturalNumber two = new NaturalNumber2(2);
201        NaturalNumber one = new NaturalNumber2(1);
202        NaturalNumber nMinusOne = new NaturalNumber2(n);
203        NaturalNumber wTemp1 = new NaturalNumber2(w);
204        NaturalNumber wTemp2 = new NaturalNumber2(w);
205        nMinusOne.decrement();
206
207        powerMod(wTemp1, two, n);
208        powerMod(wTemp2, nMinusOne, n);
209
```

```
210         if (wTemp1.compareTo(one) == 0 ||
    wTemp2.compareTo(one) != 0) {
211             witness = true;
212         }
213
214         return witness;
215     }
216
217     /**
218      * Reports whether n is a prime; may be wrong with "low"
    probability.
219      *
220      * @param n
221      *             number to be checked
222      * @return true means n is very likely prime; false means n
    is definitely
223      *         composite
224      * @requires n > 1
225      * @ensures <pre>
226      * isPrime1 = [n is a prime number, with small probability
    of error
227      *         if it is reported to be prime, and no chance of
    error if it is
228      *         reported to be composite]
229      * </pre>
230      */
231     public static boolean isPrime1(NaturalNumber n) {
232         assert n.compareTo(new NaturalNumber2(1)) > 0 :
    "Violation of: n > 1";
233         boolean isPrime;
234         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
235             /*
236              * 2 and 3 are primes
237              */
238             isPrime = true;
239         } else if (isEven(n)) {
240             /*
241              * evens are composite
242              */
```

```
243             isPrime = false;
244         } else {
245             /*
246              * odd n >= 5: simply check whether 2 is a witness
    that n is
247              * composite (which works surprisingly well :-)
248              */
249             isPrime = !isWitnessToCompositeness(new
    NaturalNumber2(2), n);
250         }
251         return isPrime;
252     }
253
254     /**
255      * Reports whether n is a prime; may be wrong with "low"
    probability.
256      *
257      * @param n
258      *             number to be checked
259      * @return true means n is very likely prime; false means n
    is definitely
260      *         composite
261      * @requires n > 1
262      * @ensures <pre>
263      * isPrime2 = [n is a prime number, with small probability
    of error
264      *         if it is reported to be prime, and no chance of
    error if it is
265      *         reported to be composite]
266      * </pre>
267      */
268     public static boolean isPrime2(NaturalNumber n) {
269         assert n.compareTo(new NaturalNumber2(1)) > 0 :
    "Violation of: n > 1";
270
271         /*
272          * Use the ability to generate random numbers (provided
    by the
273          * randomNumber method above) to generate several
```

```
   witness candidates --
274         * say, 10 to 50 candidates -- guessing that n is prime
   only if none of
275         * these candidates is a witness to n being composite
   (based on fact #3
276         * as described in the project description); use the
   code for isPrime1
277         * as a guide for how to do this, and pay attention to
   the requires
278         * clause of isWitnessToCompositeness
279         */
280
281        // TODO - fill in body
282        boolean prime = false;
283        NaturalNumber nTemp = new NaturalNumber2(n);
284        NaturalNumber one = new NaturalNumber2(1);
285        NaturalNumber two = new NaturalNumber2(2);
286
287        if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
288            /*
289             * 2 and 3 are primes
290             */
291            prime = true;
292        } else if (isEven(n)) {
293            /*
294             * evens are composite
295             */
296            prime = false;
297        } else {
298            NaturalNumber nMinusOne = new
   NaturalNumber2(nTemp);
299            nMinusOne.decrement();
300            for (int i = 0; i <= 40; i++) {
301                NaturalNumber witness = randomNumber(nTemp);
302                //isWitnessToCompositeness requires: n > 2 and
   1 < w < n - 1
303                if (n.compareTo(two) > 0 &&
   witness.compareTo(one) > 0
304                        && witness.compareTo(nMinusOne) < 0) {
```

```
305                     prime = !isWitnessToCompositeness(witness,
    n);
306                 }
307             }
308         }
309         /*
310          * This line added just to make the program compilable.
    Should be
311          * replaced with appropriate return statement.
312          */
313         return prime;
314     }
315 
316     /**
317      * Generates a likely prime number at least as large as
    some given number.
318      *
319      * @param n
320      *            minimum value of likely prime
321      * @updates n
322      * @requires n > 1
323      * @ensures n >= #n and [n is very likely a prime number]
324      */
325     public static void generateNextLikelyPrime(NaturalNumber n)
    {
326         assert n.compareTo(new NaturalNumber2(1)) > 0 :
    "Violation of: n > 1";
327 
328         /*
329          * Use isPrime2 to check numbers, starting at n and
    increasing through
330          * the odd numbers only (why?), until n is likely prime
331          */
332 
333         // TODO - fill in body
334         boolean prime = false;
335         NaturalNumber two = new NaturalNumber2(2);
336         //even number cannot be a prime
337         if (isEven(n)) {
```

```
338             n.increment();
339         }
340         prime = isPrime2(n);
341         while (!prime) {
342             n.add(two);
343             prime = isPrime2(n);
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 }
417
```