```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Shyam Sai Bethina
 *
 */
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         //if m is not zero, then the GCD of n and m will be GCD of
    m and n mod m
102         if (!m.isZero()) {
103             NaturalNumber remainder = n.divide(m);
104             reduceToGCD(m, remainder);
105
106             /*
107              * since m will be the value of n we want to return,
    used
108              * transferFrom to transfer the value
109              */
110             n.transferFrom(m);
111         }
112
113         //clears m to comply with precondition
114         m.clear();
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 isEven = (n mod 2 = 0)
125      */
126     public static boolean isEven(NaturalNumber n) {
127
128         boolean isEven = false;
129
130         //gets the last digit of n
131         int lastDigit = n.divideBy10();
132
133         //if the last digit is divisible by 2, then return true
134         if (lastDigit % 2 == 0) {
135             isEven = true;
136         }
137
138         //restores n to its original value
139         n.multiplyBy10(lastDigit);
140
141         return isEven;
142     }
143
144     /**
145      * Updates n to its p-th power modulo m.
146      *
147      * @param n
148      *            number to be raised to a power
149      * @param p
150      *            the power
151      * @param m
152      *            the modulus
153      * @updates n
154      * @requires m > 1
155      * @ensures n = #n ^ (p) mod m
156      */
157     public static void powerMod(NaturalNumber n, NaturalNumber p,
158             NaturalNumber m) {
159         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: m > 1";
160
161         /*
162          * Use the fast-powering algorithm as previously discussed
    in class,
163          * with the additional feature that every multiplication
    is followed
164          * immediately by "reducing the result modulo m"
```

```
165          */
166         NaturalNumber two = new NaturalNumber2(2);
167         NaturalNumber zero = new NaturalNumber2(0);
168         NaturalNumber one = new NaturalNumber2(1);
169 
170         //n ^ 0 is one
171         if (p.compareTo(zero) == 0) {
172             n.copyFrom(one);
173         } else {
174             NaturalNumber tempN = new NaturalNumber2(n);
175             NaturalNumber tempP = new NaturalNumber2(p);
176 
177             /*
178              * if p is even, then divide p by two, square n, and then get the
179              * powerMod of n
180              */
181             if (isEven(p)) {
182                 tempP.divide(two);
183                 n.multiply(tempN);
184                 powerMod(n, tempP, m);
185             } else {
186 
187                 /*
188                  * if p is not even, then do the same thing as if p was even,
189                  * then multiply by the original n according to the equation n(n
190                  * ^ p/2)^2
191                  */
192                 tempP.divide(two);
193                 n.multiply(tempN);
194                 powerMod(n, tempP, m);
195                 n.multiply(tempN);
196             }
197 
198             //transfers the reminder from n divided by m to n
199             n.transferFrom(n.divide(m));
200         }
201     }
202 
203     /**
204      * Reports whether w is a "witness" that n is composite, in the sense that
```

```
205      * either it is a square root of 1 (mod n), or it fails to
    satisfy the
206      * criterion for primality from Fermat's theorem.
207      *
208      * @param w
209      *            witness candidate
210      * @param n
211      *            number being checked
212      * @return true iff w is a "witness" that n is composite
213      * @requires n > 2 and 1 < w < n - 1
214      * @ensures <pre>
215      * isWitnessToCompositeness =
216      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
217      * </pre>
218      */
219     public static boolean isWitnessToCompositeness(NaturalNumber
    w,
220             NaturalNumber n) {
221         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation
    of: n > 2";
222         assert (new NaturalNumber2(1)).compareTo(w) < 0 :
    "Violation of: 1 < w";
223         n.decrement();
224         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
225         n.increment();
226
227         NaturalNumber tempW = new NaturalNumber2(w);
228         NaturalNumber tempN = new NaturalNumber2(n);
229         tempN.decrement();
230
231         NaturalNumber two = new NaturalNumber2(2);
232         NaturalNumber one = new NaturalNumber2(1);
233         boolean isWitness = false;
234
235         /*
236          * first case: (w ^ 2 mod n = 1) does powerMod to get the
    w^2 mod n part
237          * of the equation
238          */
239         powerMod(tempW, two, n);
240
241         //compares resulting value from powerMod to 1 to check the
    first case
242         if (tempW.compareTo(one) == 0) {
```

```
243             isWitness = true;
244         }
245 
246         //if first case does not pass, then resets tempW to
    original value of w
247         tempW.copyFrom(w);
248 
249         /*
250          * second case: (w ^ (n-1) mod n /= 1) uses powerMod to
    get the w^(n-1)
251          * mod n part of the equation
252          */
253         powerMod(tempW, tempN, n);
254 
255         /*
256          * if resulting value of tempW does not equal to one,
    returns true since
257          * it is a witness
258          */
259         if (tempW.compareTo(one) != 0) {
260             isWitness = true;
261         }
262 
263         //returns false if none of the cases passed
264         return isWitness;
265 
266     }
267 
268     /**
269      * Reports whether n is a prime; may be wrong with "low"
    probability.
270      *
271      * @param n
272      *            number to be checked
273      * @return true means n is very likely prime; false means n is
    definitely
274      *         composite
275      * @requires n > 1
276      * @ensures <pre>
277      * isPrime1 = [n is a prime number, with small probability of
    error
278      *         if it is reported to be prime, and no chance of
    error if it is
279      *         reported to be composite]
```

```
280      * </pre>
281      */
282     public static boolean isPrime1(NaturalNumber n) {
283         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
284         boolean isPrime;
285         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
286             /*
287              * 2 and 3 are primes
288              */
289             isPrime = true;
290         } else if (isEven(n)) {
291             /*
292              * evens are composite
293              */
294             isPrime = false;
295         } else {
296             /*
297              * odd n >= 5: simply check whether 2 is a witness
    that n is
298              * composite (which works surprisingly well :-)
299              */
300             isPrime = !isWitnessToCompositeness(new
    NaturalNumber2(2), n);
301         }
302         return isPrime;
303     }
304
305     /**
306      * Reports whether n is a prime; may be wrong with "low"
    probability.
307      *
308      * @param n
309      *            number to be checked
310      * @return true means n is very likely prime; false means n is
    definitely
311      *         composite
312      * @requires n > 1
313      * @ensures <pre>
314      * isPrime2 = [n is a prime number, with small probability of
    error
315      *         if it is reported to be prime, and no chance of
    error if it is
316      *         reported to be composite]
```

```
317      * </pre>
318      */
319     public static boolean isPrime2(NaturalNumber n) {
320         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
321
322         /*
323          * Use the ability to generate random numbers (provided by
    the
324          * randomNumber method above) to generate several witness
    candidates --
325          * say, 10 to 50 candidates -- guessing that n is prime
    only if none of
326          * these candidates is a witness to n being composite
    (based on fact #3
327          * as described in the project description); use the code
    for isPrime1
328          * as a guide for how to do this, and pay attention to the
    requires
329          * clause of isWitnessToCompositeness
330          */
331         boolean isPrime = true;
332
333         //checks if n is less than or equal to 3 since 2 and 3 are
    prime
334         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
335             isPrime = true;
336         } else if (isEven(n)) {
337             //if n is even, then it is not prime
338             isPrime = false;
339         } else {
340             NaturalNumber one = new NaturalNumber2(1);
341
342             n.decrement();
343             NaturalNumber tempN = new NaturalNumber2(n);
344             n.increment();
345
346             //tried using 50 candidates
347             int numCands = 50;
348             for (int i = 0; i < numCands; i++) {
349
350                 //gets a random number
351                 NaturalNumber random = randomNumber(n);
352
```

```
353                 /*
354                  * if the random number does not pass the
    preconditions of
355                  * isWitnessToCompositeness, then it keeps
    generating a new
356                  * random number until it does pass the
    preconditions
357                  */
358                 while (random.compareTo(one) <= 0
359                         || random.compareTo(tempN) >= 0) {
360                     random = randomNumber(n);
361                 }
362
363                 /*
364                  * uses the random number to check if it is a
    witness to n's
365                  * compositeness
366                  */
367                 if (isWitnessToCompositeness(random, n)) {
368                     isPrime = false;
369                 }
370
371             }
372
373         }
374
375         return isPrime;
376     }
377
378     /**
379      * Generates a likely prime number at least as large as some
    given number.
380      *
381      * @param n
382      *            minimum value of likely prime
383      * @updates n
384      * @requires n > 1
385      * @ensures n >= #n and [n is very likely a prime number]
386      */
387     public static void generateNextLikelyPrime(NaturalNumber n) {
388         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
    of: n > 1";
389
390         /*
```

```
391          * Use isPrime2 to check numbers, starting at n and
    increasing through
392          * the odd numbers only (why?), until n is likely prime
393          */
394
395         NaturalNumber two = new NaturalNumber2(2);
396
397         //since an even number is not prime, increments n by one
398         if (isEven(n)) {
399             n.increment();
400         }
401
402         //keeps adding two to n until it n becomes a prime number
403         while (!isPrime2(n)) {
404             n.add(two);
405         }
406
407     }
408
409     /**
410      * Main method.
411      *
412      * @param args
413      *            the command line arguments
414      */
415     public static void main(String[] args) {
416         SimpleReader in = new SimpleReader1L();
417         SimpleWriter out = new SimpleWriter1L();
418
419         /*
420          * Sanity check of randomNumber method -- just so everyone
    can see how
421          * it might be "tested"
422          */
423         final int testValue = 17;
424         final int testSamples = 100000;
425         NaturalNumber test = new NaturalNumber2(testValue);
426
427         int[] count = new int[testValue + 1];
428         for (int i = 0; i < count.length; i++) {
429             count[i] = 0;
430         }
431         for (int i = 0; i < testSamples; i++) {
432             NaturalNumber rn = randomNumber(test);
```

```
433             assert rn.compareTo(test) <= 0 : "Help!";
434             count[rn.toInt()]++;
435         }
436         for (int i = 0; i < count.length; i++) {
437             out.println("count[" + i + "] = " + count[i]);
438         }
439         out.println("  expected value = "
440                 + (double) testSamples / (double) (testValue +
    1));
441 
442         /*
443          * Check user-supplied numbers for primality, and if a
    number is not
444          * prime, find the next likely prime after it
445          */
446         while (true) {
447             out.print("n = ");
448             NaturalNumber n = new NaturalNumber2(in.nextLine());
449             if (n.compareTo(new NaturalNumber2(2)) < 0) {
450                 out.println("Bye!");
451                 break;
452             } else {
453                 if (isPrime1(n)) {
454                     out.println(n + " is probably a prime number"
455                             + " according to isPrime1.");
456                 } else {
457                     out.println(n + " is a composite number"
458                             + " according to isPrime1.");
459                 }
460                 if (isPrime2(n)) {
461                     out.println(n + " is probably a prime number"
462                             + " according to isPrime2.");
463                 } else {
464                     out.println(n + " is a composite number"
465                             + " according to isPrime2.");
466                     generateNextLikelyPrime(n);
467                     out.println("  next likely prime is " + n);
468                 }
469             }
470         }
471 
472         /*
473          * Close input and output streams
474          */
```

```
475         in.close();
476         out.close();
477     }
478
479 }
480
```