```java
import components.naturalnumber.NaturalNumber;

/**
 * Utilities that could be used with RSA cryptosystems.
 *
 * @author Feras Akileh
 *
 */
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);
            int lastDigit = (int) (base * GENERATOR.nextDouble());
            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          // checks if m is zero
 97          if (!m.isZero()) {
 98              // initializes a variable for the recursive call
 99              NaturalNumber var = n.divide(m);
100              n.transferFrom(m);
101              // recursive call
102              reduceToGCD(n, var);
103          }
104          // clears m
105          m.clear();
106
107      }
108
109      /**
110       * Reports whether n is even.
111       *
112       * @param n
113       *              the number to be checked
114       * @return true iff n is even
115       * @ensures isEven = (n mod 2 = 0)
116       */
117      public static boolean isEven(NaturalNumber n) {
118
119          // initializes a boolean variable
120          boolean statusEven = false;
121
122          // gets the last digit of n
123          int digit = n.divideBy10();
124          // returns true if the last digit is even and false if it is odd
125          // the multiplyBy10 call in each if statements restores n
```

```
126         if (digit == 0) {
127             n.multiplyBy10(digit);
128             statusEven = true;
129         } else if (digit % 2 == 0) {
130             n.multiplyBy10(digit);
131             statusEven = true;
132         } else {
133             n.multiplyBy10(digit);
134             statusEven = false;
135         }
136
137         return statusEven;
138     }
139
140     /**
141      * Updates n to its p-th power modulo m.
142      *
143      * @param n
144      *            number to be raised to a power
145      * @param p
146      *            the power
147      * @param m
148      *            the modulus
149      * @updates n
150      * @requires m > 1
151      * @ensures n = #n ^ (p) mod m
152      */
153     public static void powerMod(NaturalNumber n, NaturalNumber p,
154             NaturalNumber m) {
155         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: m > 1";
156
157         /*
158          * Use the fast-powering algorithm as previously discussed in class,
159          * with the additional feature that every multiplication is followed
160          * immediately by "reducing the result modulo m"
161          */
162
163         // initializes natural number constant
164         NaturalNumber two = new NaturalNumber2(2);
165
166         // creates a copy of p
167         NaturalNumber pCopy = new NaturalNumber2();
168         pCopy.copyFrom(p);
169
170         // goes into powerMod if-else if-else block
171         if (p.compareTo(new NaturalNumber2()) == 0) {
172             n.setFromInt(1);
173         } else if (p.compareTo(new NaturalNumber2(1)) > 0) {
174
175             if (isEven(p)) {
176
177                 NaturalNumber nCopy = new NaturalNumber2();
178                 nCopy.copyFrom(n);
179
180                 pCopy.divide(two);
181                 powerMod(nCopy, pCopy, m);
182
183                 NaturalNumber nCopyCopy = new NaturalNumber2();
184                 nCopyCopy.copyFrom(nCopy);
```

```
185
186                 nCopy.multiply(nCopyCopy);
187
188                 n.transferFrom(nCopy.divide(m));
189
190             } else {
191
192                 NaturalNumber nCopy = new NaturalNumber2();
193                 nCopy.copyFrom(n);
194
195                 pCopy.divide(two);
196                 powerMod(nCopy, pCopy, m);
197
198                 NaturalNumber nCopyCopy = new NaturalNumber2();
199                 nCopyCopy.copyFrom(nCopy);
200
201                 nCopy.multiply(nCopyCopy);
202                 nCopy.multiply(n);
203
204                 n.transferFrom(nCopy.divide(m));
205
206             }
207
208         }
209     }
210
211     /**
212      * Reports whether w is a "witness" that n is composite, in the sense that
213      * either it is a square root of 1 (mod n), or it fails to satisfy the
214      * criterion for primality from Fermat's theorem.
215      *
216      * @param w
217      *            witness candidate
218      * @param n
219      *            number being checked
220      * @return true iff w is a "witness" that n is composite
221      * @requires n > 2 and 1 < w < n - 1
222      * @ensures <pre>
223      * isWitnessToCompositeness =
224      *     (w ^ 2 mod n = 1)  or  (w ^ (n-1) mod n /= 1)
225      * </pre>
226      */
227     public static boolean isWitnessToCompositeness(NaturalNumber w,
228             NaturalNumber n) {
229         assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation of: n > 2";
230         assert (new NaturalNumber2(1)).compareTo(w) < 0 : "Violation of: 1 < w";
231         n.decrement();
232         assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
233         n.increment();
234
235         // creates boolean variable
236         boolean status = false;
237
238         // creates 2 copies of w
239         NaturalNumber wCopy = new NaturalNumber2();
240         wCopy.copyFrom(w);
241         NaturalNumber wCopy2 = new NaturalNumber2();
242         wCopy2.copyFrom(w);
243
```

```
244         // creates copy of n
245         NaturalNumber nCopy = new NaturalNumber2();
246         nCopy.copyFrom(n);
247
248         // natural number constants
249         NaturalNumber two = new NaturalNumber2(2);
250         NaturalNumber one = new NaturalNumber2(1);
251
252         // decrements nCopy
253         nCopy.decrement();
254
255         // calls to powerMod
256         powerMod(wCopy, two, n);
257         powerMod(wCopy2, nCopy, n);
258
259         if (wCopy.compareTo(one) == 0 || !(wCopy.compareTo(one) == 0)) {
260             status = true;
261         }
262         return status;
263
264     }
265
266     /**
267      * Reports whether n is a prime; may be wrong with "low" probability.
268      *
269      * @param n
270      *            number to be checked
271      * @return true means n is very likely prime; false means n is definitely
272      *         composite
273      * @requires n > 1
274      * @ensures <pre>
275      * isPrime1 = [n is a prime number, with small probability of error
276      *         if it is reported to be prime, and no chance of error if it is
277      *         reported to be composite]
278      * </pre>
279      */
280     public static boolean isPrime1(NaturalNumber n) {
281         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
282         boolean isPrime;
283         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
284             /*
285              * 2 and 3 are primes
286              */
287             isPrime = true;
288         } else if (isEven(n)) {
289             /*
290              * evens are composite
291              */
292             isPrime = false;
293         } else {
294             /*
295              * odd n >= 5: simply check whether 2 is a witness that n is
296              * composite (which works surprisingly well :-)
297              */
298             isPrime = !isWitnessToCompositeness(new NaturalNumber2(2), n);
299         }
300         return isPrime;
301     }
302
```

```
303     /**
304      * Reports whether n is a prime; may be wrong with "low" probability.
305      *
306      * @param n
307      *            number to be checked
308      * @return true means n is very likely prime; false means n is definitely
309      *         composite
310      * @requires n > 1
311      * @ensures <pre>
312      * isPrime2 = [n is a prime number, with small probability of error
313      *         if it is reported to be prime, and no chance of error if it is
314      *         reported to be composite]
315      * </pre>
316      */
317     public static boolean isPrime2(NaturalNumber n) {
318         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
319
320         /*
321          * Use the ability to generate random numbers (provided by the
322          * randomNumber method above) to generate several witness candidates --
323          * say, 10 to 50 candidates -- guessing that n is prime only if none of
324          * these candidates is a witness to n being composite (based on fact #3
325          * as described in the project description); use the code for isPrime1
326          * as a guide for how to do this, and pay attention to the requires
327          * clause of isWitnessToCompositeness
328          */
329
330         // creates natural number constants
331         NaturalNumber three = new NaturalNumber2(3);
332         NaturalNumber two = new NaturalNumber2(2);
333
334         // initializes the number of guess for the loop
335         int guessNum = 50;
336
337         // creates boolean variable that will be returned
338         boolean isPrime2 = true;
339
340         // first if statement check
341         if (n.compareTo(new NaturalNumber2(THREE)) <= 0) {
342             isPrime2 = true;
343             // second if statement check runs isEven on N
344         } else if (isEven(n)) {
345             // returns false because prime numbers are never even
346             isPrime2 = false;
347         } else {
348             // creates the upper limit of the interval
349             NaturalNumber upperLimit = new NaturalNumber2();
350             upperLimit.copyFrom(n);
351             upperLimit.subtract(three);
352
353             // runs for loop that goes through the guesses
354             for (int i = 0; i <= guessNum && isPrime2; i++) {
355                 NaturalNumber guess = randomNumber(upperLimit);
356                 // accounts for the interval
357                 guess.add(two);
358                 // calls isWitnessToCompositeness with guess as a parameter
359                 isPrime2 = !isWitnessToCompositeness(guess, n);
360             }
361
```

```
362         }
363
364         return isPrime2;
365
366     }
367
368     /**
369      * Generates a likely prime number at least as large as some given number.
370      *
371      * @param n
372      *            minimum value of likely prime
373      * @updates n
374      * @requires n > 1
375      * @ensures n >= #n and [n is very likely a prime number]
376      */
377     public static void generateNextLikelyPrime(NaturalNumber n) {
378         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation of: n > 1";
379
380         /*
381          * Use isPrime2 to check numbers, starting at n and increasing through
382          * the odd numbers only (why?), until n is likely prime
383          */
384
385         // creates natural number constant
386         NaturalNumber two = new NaturalNumber2(2);
387
388         // if n is even, it gets incremented to be odd
389         if (isEven(n)) {
390             n.increment();
391         }
392
393         //  checks if n is prime and will increment two
394         while (isPrime2(n)) {
395             n.add(two);
396         }
397
398     }
399
400     /**
401      * Main method.
402      *
403      * @param args
404      *            the command line arguments
405      */
406     public static void main(String[] args) {
407         SimpleReader in = new SimpleReader1L();
408         SimpleWriter out = new SimpleWriter1L();
409
410         /*
411          * Sanity check of randomNumber method -- just so everyone can see how
412          * it might be "tested"
413          */
414         final int testValue = 17;
415         final int testSamples = 100000;
416         NaturalNumber test = new NaturalNumber2(testValue);
417         int[] count = new int[testValue + 1];
418         for (int i = 0; i < count.length; i++) {
419             count[i] = 0;
420         }
```

```
421         for (int i = 0; i < testSamples; i++) {
422             NaturalNumber rn = randomNumber(test);
423             assert rn.compareTo(test) <= 0 : "Help!";
424             count[rn.toInt()]++;
425         }
426         for (int i = 0; i < count.length; i++) {
427             out.println("count[" + i + "] = " + count[i]);
428         }
429         out.println("  expected value = "
430                 + (double) testSamples / (double) (testValue + 1));
431
432         /*
433          * Check user-supplied numbers for primality, and if a number is not
434          * prime, find the next likely prime after it
435          */
436         while (true) {
437             out.print("n = ");
438             NaturalNumber n = new NaturalNumber2(in.nextLine());
439             if (n.compareTo(new NaturalNumber2(2)) < 0) {
440                 out.println("Bye!");
441                 break;
442             } else {
443                 if (isPrime1(n)) {
444                     out.println(n + " is probably a prime number"
445                             + " according to isPrime1.");
446                 } else {
447                     out.println(n + " is a composite number"
448                             + " according to isPrime1.");
449                 }
450                 if (isPrime2(n)) {
451                     out.println(n + " is probably a prime number"
452                             + " according to isPrime2.");
453                 } else {
454                     out.println(n + " is a composite number"
455                             + " according to isPrime2.");
456                     generateNextLikelyPrime(n);
457                     out.println("  next likely prime is " + n);
458                 }
459             }
460         }
461
462         /*
463          * Close input and output streams
464          */
465         in.close();
466         out.close();
467     }
468
469 }
```