

```
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 Put your name here
14  *
15  */
16 public final class CryptoUtilities {
17
18     /**
19      * Private constructor so this utility class cannot be
20      * instantiated.
21      */
22     private CryptoUtilities() {
23
24     }
25
26     /**
27      * Useful constant, not a magic number: 3.
28      */
29     private static final int THREE = 3;
30
31     /**
32      * Pseudo-random number generator.
33      */
34     private static final Random GENERATOR = new Random1L();
35
36     /**
37      * Returns a random number uniformly distributed in the
38      * interval [0, n].
39      *
40      * @param n
41      *         top end of interval
42      * @return random number in interval
43      * @requires n > 0
44      * @ensures <pre>
45      * 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
95 m) {
96     /*
97     * Use Euclid's algorithm; in pseudocode: if m = 0 then
98     GCD(n, m) = n
99     * else GCD(n, m) = GCD(m, n mod m)
100    */
101    if (!m.isZero()) {
102        NaturalNumber nModM = n.divide(m);
103        reduceToGCD(m, nModM);
104
105        // clearing m and making n equal to the GCD
106        n.transferFrom(m);
107    }
108 }
109
110 /**
111  * Reports whether n is even.
112  *
113  * @param n
114  *         the number to be checked
115  * @return true if n is even
116  * @ensures isEven = (n mod 2 = 0)
117  */
118 public static boolean isEven(NaturalNumber n) {
```

```

119         // looking at only the last digit
120         int digit = n.divideBy10();
121         n.multiplyBy10(digit);
122         return digit % 2 == 0;
123     }
124
125     /**
126      * Updates n to its p-th power modulo m.
127      *
128      * @param n
129      *         number to be raised to a power
130      * @param p
131      *         the power
132      * @param m
133      *         the modulus
134      * @updates n
135      * @requires m > 1
136      * @ensures n = #n ^ (p) mod m
137      */
138     public static void powerMod(NaturalNumber n, NaturalNumber p,
139                               NaturalNumber m) {
140         assert m.compareTo(new NaturalNumber2(1)) > 0 : "Violation
141 of: m > 1";
142         // used to restore p later
143         NaturalNumber pOriginal = new NaturalNumber2(p);
144
145         //
146         if (!p.isZero()) {
147             NaturalNumber nOriginal = new NaturalNumber2(n);
148             NaturalNumber n2 = new NaturalNumber2(n);
149             NaturalNumber two = new NaturalNumber2(2);
150
151             // if p is even, the only action needed is to multiply
152             // results of powerMod by each other
153             if (isEven(p)) {
154                 p.divide(two);
155                 powerMod(n2, p, m);
156                 powerMod(n, p, m);
157                 n.multiply(n2);
158             } else {
159                 // if p is odd, also multiply by nOriginal
160                 p.divide(two);

```

```

161         powerMod(n2, p, m);
162         powerMod(n, p, m);
163         n.multiply(n2);
164         n.multiply(nOriginal);
165     }
166     // making n = #n % m
167     n2 = n.divide(m);
168     n.transferFrom(n2);
169
170 } else {
171     // if p is zero, n should equal 1
172     n.clear();
173     n.increment();
174 }
175
176 // restoring p
177 p.transferFrom(pOriginal);
178 }
179
180 /**
181  * Reports whether w is a "witness" that n is composite, in
the sense that
182  * either it is a square root of 1 (mod n), or it fails to
satisfy the
183  * criterion for primality from Fermat's theorem.
184  *
185  * @param w
186  *         witness candidate
187  * @param n
188  *         number being checked
189  * @return true if w is a "witness" that n is composite
190  * @requires n > 2 and 1 < w < n - 1
191  * @ensures <pre>
192  * isWitnessToCompositeness =
193  *     (w ^ 2 mod n = 1) or (w ^ (n-1) mod n != 1)
194  * </pre>
195  */
196 public static boolean isWitnessToCompositeness(NaturalNumber
w,
197         NaturalNumber n) {
198     assert n.compareTo(new NaturalNumber2(2)) > 0 : "Violation
of: n > 2";
199     assert (new NaturalNumber2(1)).compareTo(w) < 0 :
"Violation of: 1 < w";

```

```
200     n.decrement();
201     assert w.compareTo(n) < 0 : "Violation of: w < n - 1";
202     n.increment();
203
204     // nMinus1 to be used for powerMod
205     NaturalNumber nMinus1 = new NaturalNumber2(n);
206     nMinus1.decrement();
207
208     // making copies and originals
209     NaturalNumber wOriginal = new NaturalNumber2(w);
210     NaturalNumber wCopy = new NaturalNumber2(w);
211     NaturalNumber two = new NaturalNumber2(2);
212
213     // w^2 mod n (if its a witness, it should equal 1)
214     powerMod(wCopy, two, n);
215
216     // w^n-1 mod n (if its a witness, it should NOT equal 1)
217     powerMod(w, nMinus1, n);
218     NaturalNumber shouldNotEqualOne = w.newInstance();
219     shouldNotEqualOne.transferFrom(w);
220
221     // restoring w
222     w.transferFrom(wOriginal);
223
224     NaturalNumber one = new NaturalNumber2(1);
225
226     // both must be true for this to be a witness to
    compositeness
227     return (wCopy.compareTo(one) == 0)
228           || (shouldNotEqualOne.compareTo(one) != 0);
229 }
230
231 /**
232  * Reports whether n is a prime; may be wrong with "low"
    probability.
233  *
234  * @param n
235  *         number to be checked
236  * @return true means n is very likely prime; false means n is
    definitely
237  *         composite
238  * @requires n > 1
239  * @ensures <pre>
240  * isPrime1 = [n is a prime number, with small probability of
```

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

```
278     * isPrime2 = [n is a prime number, with small probability of
error
279     *           if it is reported to be prime, and no chance of
error if it is
280     *           reported to be composite]
281     * </pre>
282     */
283     public static boolean isPrime2(NaturalNumber n) {
284         assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
of: n > 1";
285
286         /*
287         * Use the ability to generate random numbers (provided by
the
288         * randomNumber method above) to generate several witness
candidates --
289         * say, 10 to 50 candidates -- guessing that n is prime
only if none of
290         * these candidates is a witness to n being composite
(based on fact #3
291         * as described in the project description); use the code
for isPrime1
292         * as a guide for how to do this, and pay attention to the
requires
293         * clause of isWitnessToCompositeness
294         */
295         boolean isPrime = true;
296         NaturalNumber four = new NaturalNumber2(4);
297
298         // isPrime must be true if n < 4
299         if (n.compareTo(four) >= 0) {
300
301             // nMinusFour so that candidate can be incremented to
be always > 1,
302             // and so that n > candidate + 1, thus satisfying
303             // isWitnessToCompositeness' preconditions
304             NaturalNumber nMinusFour = new NaturalNumber2(n);
305             nMinusFour.subtract(four);
306
307             // generating 50 candidates to check, or until isPrime
= false
308             final int numCandidates = 50;
309             int i = 0;
310             while (i < numCandidates && isPrime) {
```



```
311         NaturalNumber candidate =
randomNumber(nMinusFour);
312         candidate.increment();
313         candidate.increment();
314         isPrime = !isWitnessToCompositeness(candidate, n);
315         i++;
316     }
317 }
318     return isPrime;
319 }
320
321 /**
322  * Generates a likely prime number at least as large as some
given number.
323  *
324  * @param n
325  *         minimum value of likely prime
326  * @updates n
327  * @requires n > 1
328  * @ensures n >= #n and [n is very likely a prime number]
329  */
330 public static void generateNextLikelyPrime(NaturalNumber n) {
331     assert n.compareTo(new NaturalNumber2(1)) > 0 : "Violation
of: n > 1";
332
333     /*
334     * Using isPrime2 to check numbers, starting at n and
increasing through
335     * odd numbers only until n is likely prime
336     */
337     if (isEven(n)) {
338         n.increment();
339     }
340     while (!isPrime2(n)) {
341         n.increment();
342         n.increment();
343     }
344 }
345
346 /**
347  * Main method.
348  *
349  * @param args
350  *         the command line arguments
```

```
351     */
352     public static void main(String[] args) {
353         SimpleReader in = new SimpleReader1L();
354         SimpleWriter out = new SimpleWriter1L();
355
356         /*
357         * Sanity check of randomNumber method -- just so everyone
can see how
358         * it might be "tested"
359         */
360         final int testValue = 17;
361         final int testSamples = 100000;
362         NaturalNumber test = new NaturalNumber2(testValue);
363         int[] count = new int[testValue + 1];
364         for (int i = 0; i < count.length; i++) {
365             count[i] = 0;
366         }
367         for (int i = 0; i < testSamples; i++) {
368             NaturalNumber rn = randomNumber(test);
369             assert rn.compareTo(test) <= 0 : "Help!";
370             count[rn.toInt()]++;
371         }
372         for (int i = 0; i < count.length; i++) {
373             out.println("count[" + i + "] = " + count[i]);
374         }
375         out.println("    expected value = "
376             + (double) testSamples / (double) (testValue +
1));
377
378         /*
379         * Check user-supplied numbers for primality, and if a
number is not
380         * prime, find the next likely prime after it
381         */
382         while (true) {
383             out.print("n = ");
384             NaturalNumber n = new NaturalNumber2(in.nextLine());
385             if (n.compareTo(new NaturalNumber2(2)) < 0) {
386                 out.println("Bye!");
387                 break;
388             } else {
389                 if (isPrime1(n)) {
390                     out.println(n + " is probably a prime number"
+ " according to isPrime1.");
391                 }
```

```
392         } else {
393             out.println(n + " is a composite number"
394                 + " according to isPrime1.");
395         }
396         if (isPrime2(n)) {
397             out.println(n + " is probably a prime number"
398                 + " according to isPrime2.");
399         } else {
400             out.println(n + " is a composite number"
401                 + " according to isPrime2.");
402             generateNextLikelyPrime(n);
403             out.println(" next likely prime is " + n);
404         }
405     }
406 }
407
408 /*
409  * Close input and output streams
410  */
411 in.close();
412 out.close();
413 }
414
415 }
```