See the Assessment Guide for information on how to interpret this report.

ASSESSMENT SUMMARY

```
Compilation:  PASSED
API:          PASSED

SpotBugs:     FAILED (1 warning)
PMD:          FAILED (3 warnings)
Checkstyle:   FAILED (0 errors, 2 warnings)

Correctness:  40/40 tests passed
Memory:       No tests available for autograding.
Timing:       119/116 tests passed

Aggregate score: 100.52%
[ Compilation: 5%, API: 5%, Style: 0%, Correctness: 70%, Memory: 20% ]
```

ASSESSMENT DETAILS

```
The following files were submitted:
----------------------------------
1.2K Jun  5 18:18 Inversions.java
1.9K Jun  5 18:18 MaximumSquareSubmatrix.java
 858 Jun  5 18:18 Ramanujan.java


================================================================================
*  COMPILING
================================================================================


% javac Inversions.java
*------------------------------------------------------------

% javac Ramanujan.java
*------------------------------------------------------------

% javac MaximumSquareSubmatrix.java
*------------------------------------------------------------


================================================================


Checking the APIs of your programs.
*------------------------------------------------------------
Inversions:

Ramanujan:

MaximumSquareSubmatrix:

================================================================


================================================================================
*  CHECKING STYLE AND COMMON BUG PATTERNS
================================================================================


% spotbugs *.class
*------------------------------------------------------------
L P UPM_UNCALLED_PRIVATE_METHOD UPM: The private method 'printMatrix()' is never called.  At MaximumSquareSubmatrix.java:[lines 59-66]
SpotBugs ends with 1 warning.


================================================================


% pmd .
*------------------------------------------------------------
Inversions.java:27: Too many control variables in the for statement [ForLoopVariableCount]
Inversions.java:30: Avoid reassigning the loop control variable 'x' [AvoidReassigningLoopVariables]
MaximumSquareSubmatrix.java:58: Avoid unused private methods, such as 'printMatrix(int)'. [UnusedPrivateMethod]
PMD ends with 3 warnings.


================================================================


% checkstyle *.java
*------------------------------------------------------------

% custom checkstyle checks for Inversions.java
*------------------------------------------------------------
[WARN] Inversions.java:1: The number (0) of calls to 'Long.parseLong()' must equal the number (1) of long integer command-line arguments. [Co
```

```
[WARN] Inversions.java:1: The number (2) of calls to 'Integer.parseInt()' must equal the number (1) of integer command-line arguments. [Comma
Checkstyle ends with 0 errors and 2 warnings.

% custom checkstyle checks for Ramanujan.java
*-----------------------------------------------------------

% custom checkstyle checks for MaximumSquareSubmatrix.java
*-----------------------------------------------------------


================================================================


********************************************************************************
*  TESTING CORRECTNESS
********************************************************************************

Testing correctness of Inversions
*-----------------------------------------------------------
Running 16 total tests.

Test 1: check output format of main() for inputs from assignment specification
  % java-introcs Inversions 10 0
  0 1 2 3 4 5 6 7 8 9

  % java-introcs Inversions 10 1
  0 1 2 3 4 5 6 7 9 8

  % java-introcs Inversions 10 45
  9 8 7 6 5 4 3 2 1 0

  % java-introcs Inversions 10 20
  9 8 0 1 2 3 7 4 5 6

==> passed

Test 2: check correctness of main() for inputs from assignment specification
  % java-introcs Inversions 10 0
  % java-introcs Inversions 10 1
  % java-introcs Inversions 10 45
  % java-introcs Inversions 10 20
==> passed

Test 3: check generate() for fixed values of n and k
  * n = 10, k = 0
  * n = 10, k = 1
  * n = 10, k = 45
  * n = 10, k = 20
==> passed

Test 4: check generate() for k = 0
  * n =  4, k = 0
  * n =  5, k = 0
  * n =  6, k = 0
  * n =  8, k = 0
  * n = 15, k = 0
  * n = 20, k = 0
  * n = 50, k = 0
  * n = 64, k = 0
==> passed

Test 5: check generate() for k = n * (n-1) / 2
  * n =  4, k = 6
  * n =  5, k = 10
  * n =  6, k = 15
  * n =  8, k = 28
  * n = 15, k = 105
  * n = 20, k = 190
  * n = 50, k = 1225
  * n = 64, k = 2016
==> passed

Test 6: check count() for permutations of length 0, 1, 2, and 3
  * a = { }
  * a = { 0 }
  * a = { 0, 1 }
  * a = { 1, 0 }
  * a = { 0, 1, 2 }
  * a = { 0, 2, 1 }
  * a = { 1, 0, 2 }
  * a = { 1, 2, 0 }
  * a = { 2, 0, 1 }
  * a = { 2, 1, 0 }
==> passed

Test 7: check generate() for fixed n and random k
  * 1000 random permutations of length  5, 0 <= k <= 10
  * 1000 random permutations of length  6, 0 <= k <= 15
  * 1000 random permutations of length  7, 0 <= k <= 21
  * 1000 random permutations of length 10, 0 <= k <= 45
  * 1000 random permutations of length 15, 0 <= k <= 105
  * 1000 random permutations of length 20, 0 <= k <= 190
==> passed

Test 8: check count() for fixed permutations
  * a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
  * a = { 1, 0, 2, 3, 4, 5, 6, 7, 8, 9 }
  * a = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }
```

```
    * a = { 9, 8, 0, 1, 2, 3, 7, 4, 5, 6 }
  ==> passed

  Test 9: check count() for sorted permutations of length n
    * sorted permutation of length 4
    * sorted permutation of length 5
    * sorted permutation of length 6
    * sorted permutation of length 8
    * sorted permutation of length 15
    * sorted permutation of length 20
    * sorted permutation of length 50
    * sorted permutation of length 64
  ==> passed

  Test 10: check count() for reverse-sorted permutations of length n
    * reverse-sorted permutation of length 4
    * reverse-sorted permutation of length 5
    * reverse-sorted permutation of length 6
    * reverse-sorted permutation of length 8
    * reverse-sorted permutation of length 15
    * reverse-sorted permutation of length 20
    * reverse-sorted permutation of length 50
    * reverse-sorted permutation of length 64
  ==> passed

  Test 11: check count() for permutations of length 0, 1, and 2
    * a = { }
    * a = { 0 }
    * a = { 0, 1 }
    * a = { 1, 0 }
    * a = { 0, 1, 2 }
    * a = { 1, 0, 2 }
    * a = { 0, 2, 1 }
    * a = { 2, 1, 0 }
  ==> passed

  Test 12: check count() for all permutations of length n
    * all permutations of length 4
    * all permutations of length 5
    * all permutations of length 6
    * all permutations of length 7
    * all permutations of length 8
  ==> passed

  Test 13: check count() for random permutations of length n
    * 1000 random permutations of length  9
    * 1000 random permutations of length 10
    * 1000 random permutations of length 12
    * 1000 random permutations of length 15
    * 1000 random permutations of length 20
    * 1000 random permutations of length 25
  ==> passed

  Test 14: check count() for random permutations of length n (larger n)
    * 1000 random permutations of length 50
    * 1000 random permutations of length 100
    * 1000 random permutations of length 200
    * 1000 random permutations of length 400
    * 1000 random permutations of length 1000
  ==> passed

  Test 15: check count() for permutations with huge number of inversions
    * reverse-sorted permutation of length 65536
    * reverse-sorted permutation of length 65537
    * reverse-sorted permutation of length 131072
  ==> passed

  Test 16: check that count() does not mutate argument array
    * 10 random permutations of length 3
    * 10 random permutations of length 5
    * 10 random permutations of length 7
    * 10 random permutations of length 10
  ==> passed


  Inversions Total: 16/16 tests passed!


  ================================================================
  Testing correctness of Ramanujan
  *----------------------------------------------------------------
  Running 15 total tests.

  Test 1: check output format of main() with inputs from assignment specification
    % java Ramanujan 1729
    true

    % java Ramanujan 3458
    false

    % java Ramanujan 4104
    true

    % java Ramanujan 216125
    true

  ==> passed
```

```
Test 2: check correctness of main() with inputs from assignment specification
  % java-introcs Ramanujan 1729
  % java-introcs Ramanujan 3458
  % java-introcs Ramanujan 4104
  % java-introcs Ramanujan 216125
==> passed

Test 3: check isRamanujan() with arguments from assignment specification
  * isRamanujan(1729)
  * isRamanujan(3458)
  * isRamanujan(4104)
  * isRamanujan(216125)
==> passed

Test 4: check isRamanujan() with Ramanujan numbers
  * isRamanujan(13832)
  * isRamanujan(20683)
  * isRamanujan(32832)
  * isRamanujan(39312)
  * isRamanujan(40033)
  * isRamanujan(46683)
  * isRamanujan(64232)
  * isRamanujan(65728)
  * isRamanujan(110656)
  * isRamanujan(110808)
  * isRamanujan(134379)
  * isRamanujan(149389)
  * isRamanujan(165464)
  * isRamanujan(171288)
  * isRamanujan(195841)
  * isRamanujan(216027)
==> passed

Test 5: check isRamanujan() with non-Ramanujan numbers
  * isRamanujan(1)
  * isRamanujan(3)
  * isRamanujan(4)
  * isRamanujan(5)
  * isRamanujan(93)
  * isRamanujan(346)
  * isRamanujan(51214)
==> passed

Test 6: check isRamanujan() with non-Ramanujan numbers
        (sum of two cubes in only 1 way)
  * isRamanujan(2)
  * isRamanujan(35)
  * isRamanujan(65)
  * isRamanujan(91)
  * isRamanujan(128)
  * isRamanujan(189)
  * isRamanujan(341)
==> passed

Test 7: check isRamanujan() with non-positive integers
  * isRamanujan(0)
  * isRamanujan(-1)
  * isRamanujan(-2)
  * isRamanujan(-91)
  * isRamanujan(-189)
  * isRamanujan(-1729)
  * isRamanujan(-123456789)
==> passed

Test 8: check isRamanujan() with Ramanujan triples
        (sum of two cubes in 3 different ways)
  * isRamanujan(87539319)
  * isRamanujan(119824488)
  * isRamanujan(143604279)
  * isRamanujan(175959000)
  * isRamanujan(327763000)
  * isRamanujan(700314552)
  * isRamanujan(804360375)
  * isRamanujan(958595904)
==> passed

Test 9: check isRamanujan() with Ramanujan quadruples
        (sum of two cubes in 4 different ways)
  * isRamanujan(6963472309248)
  * isRamanujan(12625136269928)
  * isRamanujan(21131226514944)
  * isRamanujan(26059452841000)
==> passed

Test 10: check isRamanujan() with random Ramanujan numbers
  * 100 random Ramanujan numbers between 1 and 10000
  * 10000 random Ramanujan numbers between 1 and 1000000
  * 10000 random Ramanujan numbers between 1 and 10000000
  * 10000 random Ramanujan numbers between 1 and 100000000
  * 10000 random Ramanujan numbers between 1 and 1000000000
==> passed

Test 11: check isRamanujan() with random non-Ramanujan numbers
  * 100 random non-Ramanujan numbers between 1 and 1000
  * 10000 random non-Ramanujan numbers between 1 and 1000000
  * 10000 random non-Ramanujan numbers between 1 and 10000000
  * 10000 random non-Ramanujan numbers between 1 and 100000000
  * 10000 random non-Ramanujan numbers between 1 and 1000000000
```

```
==> passed

Test 12: check isRamanujan() with random Ramanujan numbers (large integers)
  * 100 random Ramanujan numbers between 1 and 10000000000
  * 100 random Ramanujan numbers between 1 and 100000000000
  * 100 random Ramanujan numbers between 1 and 1000000000000
  * 100 random Ramanujan numbers between 1 and 10000000000000
  * 100 random Ramanujan numbers between 1 and 100000000000000
  * 100 random Ramanujan numbers between 1 and 1000000000000000
==> passed

Test 13: check isRamanujan() with random non-Ramanujan numbers (large integers)
  * 100 random non-Ramanujan numbers between 1 and 10000000000
  * 100 random non-Ramanujan numbers between 1 and 100000000000
  * 100 random non-Ramanujan numbers between 1 and 1000000000000
  * 100 random non-Ramanujan numbers between 1 and 10000000000000
  * 100 random non-Ramanujan numbers between 1 and 100000000000000
  * 100 random non-Ramanujan numbers between 1 and 1000000000000000
==> passed

Test 14: check isRamanujan() with Ramanujan numbers near 2^63 - 1
  * isRamanujan(9223261564912520489)
  * isRamanujan(9223278330318728221)
==> passed

Test 15: check that main() is consistent with isRamanujan()
  * 10000 random Ramanujan numbers between 1 and 1000000
  * 10000 random Ramanujan numbers between 1 and 10000000
  * 10000 random Ramanujan numbers between 1 and 100000000
  * 10000 random Ramanujan numbers between 1 and 1000000000
==> passed


Ramanujan Total: 15/15 tests passed!


================================================================
Testing correctness of MaximumSquareSubmatrix
*----------------------------------------------------------
Running 9 total tests.

Test 1: check output format for inputs from assignment specification
  % java-introcs MaximumSquareSubmatrix < square5.txt
  2

  % java-introcs MaximumSquareSubmatrix < square6.txt
  3

  % java-introcs MaximumSquareSubmatrix < square7.txt
  4

  % java-introcs MaximumSquareSubmatrix < square10.txt
  2

==> passed

Test 2: check that main() reads all data from standard input
  % java-introcs MaximumSquareSubmatrix < square5.txt
  % java-introcs MaximumSquareSubmatrix < square6.txt
  % java-introcs MaximumSquareSubmatrix < square7.txt
  % java-introcs MaximumSquareSubmatrix < square10.txt
==> passed

Test 3: check main() with input files from assignment specification
  % java-introcs MaximumSquareSubmatrix < square5.txt
  % java-introcs MaximumSquareSubmatrix < square6.txt
  % java-introcs MaximumSquareSubmatrix < square7.txt
  % java-introcs MaximumSquareSubmatrix < square10.txt
==> passed

Test 4: check size() on input files
  * filename = square5.txt
  * filename = square6.txt
  * filename = square7.txt
  * filename = square10.txt
==> passed

Test 5: check size() on random n-by-n matrices
  * 1000 random 5-by-5 matrices, with each entry 1 with probability 0.5
  * 1000 random 6-by-6 matrices, with each entry 1 with probability 0.1
  * 1000 random 7-by-7 matrices, with each entry 1 with probability 0.9
  * 1000 random 8-by-8 matrices, with each entry 1 with probability 0.99
  * 1000 random 10-by-10 matrices, with each entry 1 with probability 0.5
  * 1000 random 20-by-20 matrices, with each entry 1 with probability 0.1
  * 1000 random 25-by-25 matrices, with each entry 1 with probability 0.9
  * 1000 random 30-by-30 matrices, with each entry 1 with probability 0.99
==> passed

Test 6: check size() on n-by-n matrix of all 0s
  * 1-by-1 matrix of all 0s
  * 3-by-3 matrix of all 0s
  * 5-by-5 matrix of all 0s
  * 10-by-10 matrix of all 0s
  * 15-by-15 matrix of all 0s
  * 20-by-20 matrix of all 0s
==> passed

Test 7: check size() on n-by-n matrix of all 1s
```

```
   * 1-by-1 matrix of all 1s
   * 3-by-3 matrix of all 1s
   * 5-by-5 matrix of all 1s
   * 10-by-10 matrix of all 1s
   * 15-by-15 matrix of all 1s
   * 20-by-20 matrix of all 1s
==> passed

Test 8: check that size(a) does not mutate a[][] for random n-by-n matrices
  * 10 random 3-by-3 matrices
  * 10 random 5-by-5 matrices
  * 10 random 7-by-7 matrices
  * 10 random 10-by-10 matrices
==> passed

Test 9: check that main() is consistent with size()
  * 1000 random 4-by-4 matrices
  * 1000 random 5-by-5 matrices
  * 1000 random 8-by-8 matrices
  * 1000 random 10-by-10 matrices
==> passed


MaximumSquareSubmatrix Total: 9/9 tests passed!


================================================================
********************************************************************************
*  TIMING
********************************************************************************

Timing Ramanujan
*----------------------------------------------------------
Running 44 total tests.

Tests 1-15: time isRamanujan(), for random non-Ramanujan numbers between 1 and 10^k,
            for k = 4 to 18

The approximate order-of-growth is n ^ (log ratio)

                 k  seconds  log ratio
               --------------------------
  => passed        4    0.000
  => passed        5    0.000
  => passed        6    0.000
  => passed        7    0.000
  => passed        8    0.001
  => passed        9    0.002
  => passed       10    0.003       0.22
  => passed       11    0.005       0.18
  => passed       12    0.007       0.18
  => passed       13    0.014       0.30
  => passed       14    0.030       0.34
  => passed       15    0.067       0.34
  => passed       16    0.142       0.33
  => passed       17    0.304       0.33
  => passed       18    0.652       0.33

Test 16-30: time isRamanujan(), for random Ramanujan numbers between 1 and 10^k,
            for k = 4 to 18

The approximate order-of-growth is n ^ (log ratio)

                 k  seconds  log ratio
               --------------------------
  => passed        4    0.000
  => passed        5    0.000
  => passed        6    0.000
  => passed        7    0.000
  => passed        8    0.000
  => passed        9    0.001
  => passed       10    0.002
  => passed       11    0.003       0.34
  => passed       12    0.008       0.36
  => passed       13    0.015       0.28
  => passed       14    0.030       0.31
  => passed       15    0.067       0.35
  => passed       16    0.142       0.33
  => passed       17    0.306       0.33
  => passed       18    0.661       0.34

Test 31-37: time isRamanujan() method with random Ramanujan numbers
            between 1 and 1000000
     -  reference solution calls per second:  180456.00
     -  student   solution calls per second:  367870.00
     -  reference / student ratio:                0.49

  => passed      student <= 10000x reference
  => passed      student <=  5000x reference
  => passed      student <=  1000x reference
  => passed      student <=   500x reference
  => passed      student <=   100x reference
  => passed      student <=    50x reference
  => passed      student <=    10x reference
  => BONUS       student <=     2x reference
  => BONUS       student <=   0.5x reference

Test 38-44: time isRamanujan() method with random non-Ramanujan numbers
```

```
            between 1 and 1000000
      -  reference solution calls per second:  112224.00
      -  student   solution calls per second:  147321.50
      -  reference / student ratio:                 0.76

   => passed      student <= 10000x reference
   => passed      student <=  5000x reference
   => passed      student <=  1000x reference
   => passed      student <=   500x reference
   => passed      student <=   100x reference
   => passed      student <=    50x reference
   => passed      student <=    10x reference
   => BONUS       student <=     2x reference


  Ramanujan Total: 47/44 tests passed!


  ================================================================


  ********************************************************************************
  *  TIMING
  ********************************************************************************

  Timing Inversions
  *-----------------------------------------------------------
  Running 57 total tests.

  Tests 1-10: calling generate(n, k) with k = 0 and n ranging
              from 8192 to 4194304, going up by a factor of 2

  The approximate order-of-growth is n ^ (log ratio)

                n  seconds  log ratio
          ---------------------------
             8192    0.000
            16384    0.000
            32768    0.001
            65536    0.001
           131072    0.002        1.0
           262144    0.004        1.0
           524288    0.009        1.0
          1048576    0.018        1.0
          2097152    0.034        0.9
          4194304    0.066        1.0
  ==> passed

  Tests 11-20: calling generate(n, k) with k = n(n-1)/2 and n ranging
              from 8192 to 4194304, going up by a factor of 2

  The approximate order-of-growth is n ^ (log ratio)

                n  seconds  log ratio
          ---------------------------
             8192    0.000
            16384    0.001
            32768    0.001
            65536    0.003        1.1
           131072    0.005        0.7
           262144    0.010        1.0
           524288    0.019        1.0
          1048576    0.040        1.0
          2097152    0.080        1.0
          4194304    0.161        1.0
  ==> passed

  Tests 21-30: calling generate(n, k) with k = random and n ranging
              from 8192 to 4194304, going up by a factor of 2

  The approximate order-of-growth is n ^ (log ratio)

                n  seconds  log ratio
          ---------------------------
             8192    0.000
            16384    0.000
            32768    0.001
            65536    0.001
           131072    0.003        1.5
           262144    0.004        0.5
           524288    0.017        2.1
          1048576    0.018        0.1
          2097152    0.037        1.0
          4194304    0.073        1.0
  ==> passed

  Tests 31-39: calling count() with sorted permutations of length n, with n
              ranging from 512 to 8192, going up by a factor of sqrt(2)

  The approximate order-of-growth is n ^ (log ratio)

               n  seconds  log ratio
          --------------------------
             512    0.002
             724    0.004        1.9
            1024    0.008        2.0
            1448    0.014        1.8
```

```
          2048    0.028        2.0
          2896    0.057        2.1
          4096    0.114        2.0
          5793    0.226        2.0
          8192    0.453        2.0
==> passed

Tests 40-48: calling count() with reverse-sorted permutations of length n, with n
             ranging from 512 to 8192, going up by a factor of sqrt(2)

The approximate order-of-growth is n ^ (log ratio)

             n  seconds  log ratio
        --------------------------
           512    0.003   Infinity
           724    0.005        2.0
          1024    0.011        2.0
          1448    0.021        2.0
          2048    0.043        2.1
          2896    0.085        1.9
          4096    0.172        2.0
          5793    0.344        2.0
          8192    0.683        2.0
==> passed

Tests 49-57: calling count() with random permutations of length n, with n
             ranging from 512 to 8192, going up by a factor of sqrt(2)

The approximate order-of-growth is n ^ (log ratio)

             n  seconds  log ratio
        --------------------------
           512    0.003   Infinity
           724    0.006        2.3
          1024    0.011        1.9
          1448    0.020        1.8
          2048    0.041        2.0
          2896    0.082        2.0
          4096    0.165        2.0
          5793    0.330        2.0
          8192    0.660        2.0
==> passed


Inversions Total: 57/57 tests passed!


================================================================



Timing MaximumSquareSubmatrix
*-----------------------------------------------------------
Running 15 total tests.

Tests 1-15: Calling size() with random n-by-n matrices, with n ranging from
            16 to 2048, going up by a factor of sqrt(2).

The approximate order-of-growth is n ^ (log ratio)


                 n  seconds  log ratio
              --------------------------
  => passed       16    0.000
  => passed       23    0.000
  => passed       32    0.000
  => passed       45    0.000
  => passed       64    0.000
  => passed       91    0.001
  => passed      128    0.002
  => passed      181    0.003        2.0
  => passed      256    0.006        2.0
  => passed      362    0.013        2.0
  => passed      512    0.028        2.2
  => passed      724    0.052        1.8
  => passed     1024    0.105        2.0
  => passed     1448    0.207        2.0
  => passed     2048    0.415        2.0

MaximumSquareSubmatrix Total: 15/15 tests passed!


================================================================



********************************************************************************
*  TIMING
********************************************************************************

Timing Ramanujan
*-----------------------------------------------------------
Running 44 total tests.

Tests 1-15: time isRamanujan(), for random non-Ramanujan numbers between 1 and 10^k,
            for k = 4 to 18

The approximate order-of-growth is n ^ (log ratio)
```

```
                  k  seconds  log ratio
               -------------------------
  => passed        4    0.000
  => passed        5    0.000
  => passed        6    0.000
  => passed        7    0.000
  => passed        8    0.001
  => passed        9    0.001
  => passed       10    0.003       0.34
  => passed       11    0.005       0.24
  => passed       12    0.007       0.19
  => passed       13    0.014       0.30
  => passed       14    0.030       0.32
  => passed       15    0.065       0.33
  => passed       16    0.141       0.34
  => passed       17    0.304       0.33
  => passed       18    0.659       0.34

Test 16-30: time isRamanujan(), for random Ramanujan numbers between 1 and 10^k,
            for k = 4 to 18

The approximate order-of-growth is n ^ (log ratio)

                  k  seconds  log ratio
               -------------------------
  => passed        4    0.000
  => passed        5    0.000
  => passed        6    0.000
  => passed        7    0.000
  => passed        8    0.000
  => passed        9    0.001
  => passed       10    0.002
  => passed       11    0.003       0.34
  => passed       12    0.007       0.30
  => passed       13    0.014       0.31
  => passed       14    0.032       0.36
  => passed       15    0.066       0.32
  => passed       16    0.143       0.34
  => passed       17    0.304       0.33
  => passed       18    0.660       0.34

Test 31-37: time isRamanujan() method with random Ramanujan numbers
            between 1 and 1000000
     -  reference solution calls per second:  181178.00
     -  student   solution calls per second:  369663.00
     -  reference / student ratio:                0.49

  => passed      student <= 10000x reference
  => passed      student <=  5000x reference
  => passed      student <=  1000x reference
  => passed      student <=   500x reference
  => passed      student <=   100x reference
  => passed      student <=    50x reference
  => passed      student <=    10x reference
  => BONUS       student <=     2x reference
  => BONUS       student <=   0.5x reference

Test 38-44: time isRamanujan() method with random non-Ramanujan numbers
            between 1 and 1000000
     -  reference solution calls per second:  114256.50
     -  student   solution calls per second:  149959.50
     -  reference / student ratio:                0.76

  => passed      student <= 10000x reference
  => passed      student <=  5000x reference
  => passed      student <=  1000x reference
  => passed      student <=   500x reference
  => passed      student <=   100x reference
  => passed      student <=    50x reference
  => passed      student <=    10x reference
  => BONUS       student <=     2x reference


Ramanujan Total: 47/44 tests passed!


================================================================
```