New Mexico Supercomputing Challenge
Mini-Challenge Problem #1: Tiling with Pythagorean Triples

Problem Description [1]

Let (a, b, c) represent the three sides of a right triangle with integral length sides (a *Pythagorean triple*). If we place four such triangles together to form a square with sides of length c, we get the result shown in the left side of figure 1. Note that there's a square formed in the center; this center square has sides of length $b-a$. In some (but not all) cases, the larger square can be completely tiled with the smaller square (right side of figure 1) – i.e. the larger square can be filled completely, with no overlaps, overhangs, or gaps, using tiles that are the same size as the smaller square.

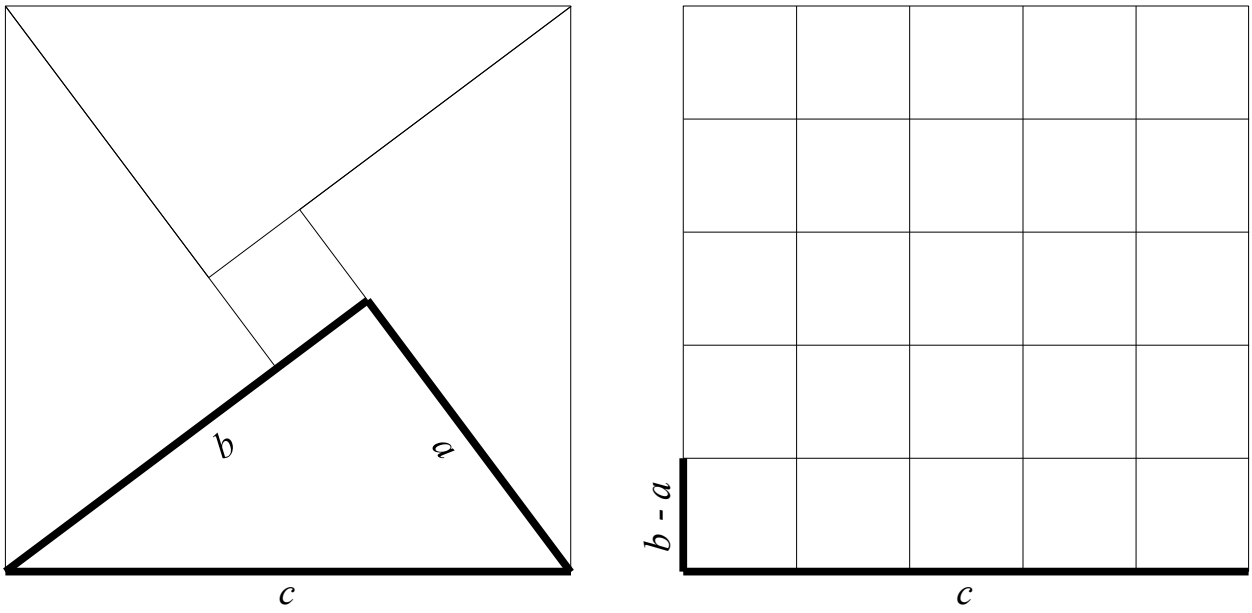

Figure 1: Tiling a square formed by the hypotenuse of a right triangle

How many Pythagorean triples allow the tiling described above, where the perimeter is less than one hundred million – i.e. where $a+b+c < 100000000$?

Answer

There are 10,057,761 Pythagorean triples which satisfy the conditions described above.

A number of different solution techniques can be used to arrive at this solution; some are described in detail in the pages that follow.

Brute Force Method

(**Note**: Implementing this approach is **NOT** recommended.)

We might try to solve the problem in a very brute force fashion, by trying all possible combinations of a and b values (within the specified perimeter limit), testing to see if the resulting c value is integral, and then checking to see whether c is evenly divisible by $b-a$.

This approach can be expressed in pseudocode, as follows:

1. u = upper limit (inclusive) on triangle perimeter
2. $t = 0$
3. For a in $\left\{1, 2, \ldots, \left\lfloor \frac{u}{2} \right\rfloor\right\}$, do:
    a. $b = a + 1$
    b. While $a + b + \sqrt{a^2 + b^2} \leq u$, do:
        i. $c = \sqrt{a^2 + b^2}$
        ii. If $\lfloor c \rfloor = c$ and $c \bmod (b - a) = 0$, then:
            (1) $t = t + 1$
        iii. $b = b + 1$
4. Finished; the final value of t is the number of triangles which satisfy the conditions.

The problem with this approach is simple (and serious): it's horribly inefficient. Even if we take advantage of multiprocessing, it could easily take a single desktop or laptop computer several weeks – even months – to compute the correct answer with an implementation of the algorithm given above.

At the heart of the problem is the fact that even when the two legs of a right triangle have integral lengths, the length of the hypotenuse is usually non-integral. So this approach ends up testing many combinations of a and b values that don't result in Pythagorean triples – let alone triples that satisfy the tiling condition.

We can improve matters by ignoring some (a, b) pairs that we know in advance won't be part of a Pythagorean triple. For example, if we spent some time on the mathematics of the problem, we would see that no Pythagorean triple has odd values for both a and b – in fact, at least one of a and b must be evenly divisible by 4. Similarly, at least one of a and b must be an integral multiple of 3, and at least one of a, b, and c must be an integral multiple of 5. We could modify our algorithm to take advantage of these facts; however, these are improvements that just make a poor algorithm run faster. It would be better if we could find a fundamentally different – and better – algorithm for generating Pythagorean triples.

Generating Pythagorean Triples with Euclid's Formula

Fortunately for us, Euclid (among others) created a formula for generating Pythagorean triples directly. He stated and proved the formula in geometric terms; we'll omit the proof, and state the formula this way [2], [3], [4]:

Given m and n, where:

$m, n \in \{1, 2, 3, \ldots\}$
$m > n$
m and n have opposite parity – i.e. if one is odd, the other must be even.
m and n are *coprime* – i.e. they have no common integer factors greater than 1.

Then:

$(m^2 - n^2,\ 2mn,\ m^2 + n^2)$ is a *primitive Pythagorean triple* (a Pythagorean triple where no two of the three values have a common factor larger than 1).

All primitive Pythagorean triples can be generated via Euclid's formula.

There are also Pythagorean triples where a, b, and c have a common factor; these are called *derivative Pythagorean triples*, and they're not generated by the above formula. A derivative Pythagorean triple has the form (ka, kb, kc), where (a, b, c) is a primitive Pythagorean triple, and k is a positive integer.

If we have a primitive Pythagorean triple that satisfies the tiling condition, it's easy to show that all Pythagorean triples derived from it also satisfy the tiling condition. Conversely, no derivative Pythagorean triple will satisfy the tiling condition unless the corresponding primitive Pythagorean triple satisfies the tiling condition. Thus, we need only test primitive Pythagorean triples for the tiling condition.

When we find a primitive Pythagorean triple that satisfies the tiling condition, counting the derivative Pythagorean triples that fall within the total perimeter constraint is a simple matter of arithmetic: dividing the maximum perimeter allowed by the sum of the values in the primitive Pythagorean triple and truncating the result to an integer will return one more than the number of derivatives – i.e. the number of derivatives plus one for the primitive itself.

The perimeter of a primitive Pythagorean triple generated by Euclid's formula is $2m^2 + 2mn$. For any given value of m, the perimeter takes its minimum value of $2m^2 + 2m$ when $n = 1$. Given this, we know that we can iterate over values of m, starting at 1 and stopping when $2m^2 + 2m \geq u$. Treating this as a quadratic equation in m, we have the following:

$$m = \frac{-2 \pm \sqrt{4 + 8u}}{4}$$
$$m = \frac{-1 \pm \sqrt{1 + 2u}}{2}$$

Thus, the inclusive upper bound for m is $\left\lfloor \frac{-1 + \sqrt{1 + 2u}}{2} \right\rfloor$.

We can now write the pseudocode for a somewhat less brute-force approach:

1. u = upper limit (inclusive) on triangle perimeter
2. $t = 0$
3. For m in $\left\{2, 3, \ldots, \left\lfloor \frac{-1+\sqrt{1+2u}}{2} \right\rfloor\right\}$, do:
    a. If m is even, then:
        i. $n = 1$

        Otherwise:

        ii. $n = 2$
    b. While $n < m$ and $2m^2+2mn \leq u$, do:
        i. If m and n are coprime, then:
            (1) $a = 2mn$
            (2) $b = m^2-n^2$
            (3) $c = m^2+n^2$
            (4) If $c \bmod (b-a) = 0$, then:
                - $t = t+\left\lfloor \frac{u}{a+b+c} \right\rfloor$
        ii. $n = n+2$
4. Finished; the final value of t is the number of triangles which satisfy the conditions.

One operation that might be a bit hazy is in step #3bi. How can we find out whether two numbers are coprime? One simple method – which is also fairly efficient – is Euclid's algorithm for finding the greatest common divisor (GCD) of two numbers[5]; if the GCD is 1, then the two numbers are coprime.

Whereas an implementation of the previous algorithm would take several weeks to solve our problem, one incorporating this algorithm is much more efficient: it can come up with the correct answer in a matter of seconds.

(For implementation, see `Euclid.java`, appendix p. iv.)

Adding Constraints for More Efficient Generation of Primitive Pythagorean Triples

We've already seen that we really only need to examine primitive Pythagorean triples. Given that, there are more improvements we can make to our algorithm.

The area of a right triangle with legs a and b is $ab/2$. Thus, the combined area of the four triangles in figure 1 is $2ab$; the area of the center square is $(b-a)^2$. The area of the larger square is equal to the sum of these two; this is visually obvious, but it can also be seen from the Pythagorean theorem:

$$\begin{aligned} a^2+b^2 &= c^2 \\ a^2+b^2-2ab &= c^2-2ab \\ (b-a)^2 &= c^2-2ab \\ (b-a)^2+2ab &= c^2 \end{aligned}$$

Since the area of the smaller square (i.e. the tile) is $(b-a)^2$, the area of the larger square, c^2, must be evenly divisible by $(b-a)^2$. From the last equation above, it follows that $2ab$ must also be evenly divisible by $(b-a)^2$.

In a primitive Pythagorean triple, a and b have no common factors greater than 1; also, exactly one of a and b must be odd, and the other even. Therefore, $b-a$ is odd – and so is $(b-a)^2$, while $2ab$ is necessarily even.

Since a and b are coprime, $b-a$ isn't a factor of either a or b – except when $b-a = \pm 1$. Thus, in a primitive Pythagorean triple, $2ab$ is evenly divisible by $(b-a)^2$ if and only if $b-a = \pm 1$.

Given the above, we don't need to test whether c is evenly divisible by $b-a$; instead, we can simply test for $b-a = \pm 1$. But we can still do much better than this slight improvement.

In our current algorithm, we're iterating over a range of n values for each value of m. But for any given m, we're really only interested in those values of n that result in $b-a = \pm 1$ - since those are the only values that may satisfy the tiling condition.

Let's review how a, b, and c are computed, with Euclid's formula:

$$\begin{aligned} a &= m^2-n^2 \\ b &= 2mn \\ c &= m^2+n^2 \end{aligned}$$

For a given value of m, we'll focus on those values of n that result in $b-a = \pm 1$:

$$\begin{aligned} b-a &= \pm 1 \\ 2mn-m^2+n^2 &= \pm 1 \\ n^2+2mn-(m^2\pm 1) &= 0 \end{aligned}$$

If we take m as a given, we now have a quadratic equation in n, and we can apply the quadratic formula:

$$\begin{aligned} n &= \frac{-2m \pm \sqrt{4m^2 + 4(m^2 \pm 1)}}{2} \\ n &= \frac{-2m \pm \sqrt{8m^2 \pm 4}}{2} \\ n &= \frac{-2m \pm 2\sqrt{2m^2 \pm 1}}{2} \\ n &= -m \pm \sqrt{2m^2 \pm 1} \end{aligned}$$

Clearly, we're only interested in positive values of m, so we can simplify the last equation a bit:

$$n = \sqrt{2m^2 \pm 1} - m$$

This means that for a given m, our algorithm only needs to iterate over the values of n where:

$$\sqrt{2m^2 - 1} - m \leq n \leq \sqrt{2m^2 + 1} - m$$

For all $m \geq 2$, this range includes a maximum of one integer – in fact, for most values of m, the range includes no integer values at all). If the range does include an integer, it's the value given by $\left\lceil \sqrt{2m^2 - 1} \right\rceil - m$.

Now that we have a much better understanding of the range of n values that we need to test for any given m value, we can also narrow the range of m values:

$$\begin{aligned} 2m^2 + 2mn &\leq u \\ 2m^2 + 2m(\left\lceil \sqrt{2m^2 - 1} \right\rceil - m) &\leq u \\ 2m^2 + 2m\left\lceil \sqrt{2m^2 - 1} \right\rceil - 2m^2 &\leq u \\ 2m\left\lceil \sqrt{2m^2 - 1} \right\rceil &\leq u \\ 4m^2(2m^2 - 1) &\leq u^2 \\ 8m^4 - 4m^2 &\leq u^2 \end{aligned}$$

Once again, we can use the quadratic formula (this time in m^2) to find an upper bound for m:

$$m^2 = \frac{4 \pm \sqrt{16 + 32u^2}}{16}$$

$$m^2 = \frac{1 \pm \sqrt{1 + 2u^2}}{4}$$

Taking only the positive root, and taking the square root again, we get an upper bound for m :

$$m \leq \left\lfloor \frac{\sqrt{1+\sqrt{1+2u^2}}}{2} \right\rfloor$$

(In practice, it probably makes sense to approximate this upper bound with a simpler expression.)

Incorporating all of the above, we have a new algorithm:

1. u = upper limit (inclusive) on triangle perimeter
2. $t = 0$
3. For m in $\left\{2, 3, \ldots, \left\lfloor \frac{\sqrt{1+\sqrt{1+2u^2}}}{2} \right\rfloor\right\}$, do:
   a. $n = \left\lfloor \sqrt{2m^2-1} \right\rfloor - m$
   b. If m and n are coprime and $2mn-m^2+n^2 = \pm 1$, then:
      i. $t = t + \left\lfloor \frac{u}{2m^2+2mn} \right\rfloor$
4. Finished; the final value of t is the number of triangles which satisfy the conditions.

This algorithm computes just a single value of n for each m , then tests m and n for relative primality, and finally tests the tiling condition. As noted above, the range of possible n values for a given m may not even include an integer (in fact, it usually doesn't); we could test for this, but the best improvement in efficiency that we could get that way probably wouldn't be large enough to warrant the additional complexity of the algorithm. As it is, an implementation of this algorithm is significantly more efficient than the previous one, solving the problem in a matter of milliseconds.

(For implementation, see `Constrained.java`, appendix p. v.)

Solution via Pell's Equation

In the constrained approach, we saw that when using Euclid's formula, the tiling condition can be expressed as:

$$2mn - m^2 + n^2 = \pm 1$$

We can rearrange and group the terms, and define two new variables, to transform the equation:

$$\begin{aligned} m^2 - 2mn + n^2 - 2n^2 &= \mp 1 \\ (m-n)^2 - 2n^2 &= \mp 1 \\ x &= m-n \\ y &= n \\ x^2 - 2y^2 &= \mp 1 \end{aligned}$$

Since m and n are positive integers, with $m > n$, it follows from the above that x and y will also be positive integers. Given that, the last version of the tiling condition fits the *Pell's equation* form of *Diophantine equations* [6], [7]. A Diophantine equation is a polynomial equation with the added constraint that only integer solutions are allowed; Pell's equation is a specific form of Diophantine equation which has been the subject of much study, and which has a number of powerful solution techniques.

The solution technique we'll use starts with a fundamental solution (where x and y both have positive integral values, but where the value of x is minimized), and then uses a recurrence relation to generate each successive solution. Fortunately, this specific Pell's equation has a well known fundamental solution, $(1,1)$, and a well understood recurrence relation:

$$\begin{aligned} x_{i+1} &= x_1 x_i + 2 y_1 y_i \\ y_{i+1} &= x_1 y_i + x_i y_1 \\ x_1 &= 1 \\ y_1 &= 1 \end{aligned}$$

We can reverse the previous transformation, to express this recurrence relation in m and n:

$$\begin{aligned} n_{i+1} &= (m_1 - n_1) n_i + (m_i - n_i) n_1 \\ &= m_1 n_i + m_i n_1 - 2 n_1 n_i \\ m_{i+1} - n_{i+1} &= (m_1 - n_1)(m_i - n_i) + 2 n_1 n_i \\ m_{i+1} &= m_1 m_i - m_1 n_i - m_i n_1 + n_1 n_i + 2 n_1 n_i + m_1 n_i + m_i n_1 - 2 n_1 n_i \\ &= m_1 m_i + n_1 n_i \end{aligned}$$

We can do the same thing with the fundamental solution:

$$\begin{aligned} m_1 - n_1 &= 1 \\ n_1 &= 1 \\ m_1 &= 2 \end{aligned}$$

Putting it all together, we now have a recurrence relation and a fundamental solution in terms of m and n:

$$\begin{aligned} m_{i+1} &= m_1 m_i + n_1 n_i \\ n_{i+1} &= m_1 n_i + m_i n_1 - 2 n_1 n_i \\ m_1 &= 2 \\ n_1 &= 1 \end{aligned}$$

This recurrence relation leads to a simple – but very powerful – algorithm:

1. u = upper limit (inclusive) on triangle perimeter
2. $t = 0$
3. $m_1 = 2$
4. $n_1 = 1$
5. $m = m_1$
6. $n = n_1$
7. While $2m^2 + 2mn \le u$, do:
   a. $t = t + \left\lfloor \frac{u}{2m^2 + 2mn} \right\rfloor$
   b. $m' = m_1 m + n_1 n$
   c. $n = m_1 n + m n_1 - 2 n_1 n$
   d. $m = m'$
8. Finished; the final value of t is the number of triangles which satisfy the conditions.

(For implementation, see `Pell.java`, appendix p. v.)

Summary

With the exception of the brute force technique described initially, all of the algorithms presented here represent reasonable approaches to solving the Pythagorean triple tiling problem. Additionally, any one of them can be implemented with just a few dozen lines of code. (Interestingly, the additional cost in code complexity incurred in trying to make the brute force solution run faster resulted in that approach requiring more lines of unique code than any of the other solutions.) They vary widely in performance, and in the mathematical background expected of the programmer – but most of the concepts are well within the grasp of a high school student with solid math skills.

Possibly the most important observation here is that for a student with fair math skills, but without previous exposure to Euclid's formula, an hour of research and study would probably have been sufficient to move from an extremely impractical approach (the brute force method) to one capable of completing the task at hand literally millions of times faster. A little bit of research and math exercise doesn't always pay off so dramatically, but it's virtually always worth the effort.

References

[1] "Problem 139," *Project Euler.net,* Jan 27, 2007. Available: http://projecteuler.net/index.php?section=problems&id=139. [Accessed: Jan. 18, 2010].

[2] E.W. Weisstein, "Pythagorean Triple," *Wolfram MathWorld*. Available: http://mathworld.wolfram.com/PythagoreanTriple.html. [Accessed: Jan. 18, 2010].

[3] "Pythagorean triple," *Wikipedia*, Jan. 11, 2010. Available: http://en.wikipedia.org/wiki/Pythagorean_triple. [Accessed: Jan. 18, 2010].

[4] Euclid, "Proposition 29." in *Elements*, Book X.

[5] P.E. Black, "Euclid's algorithm," *Dictionary of Algorithms and Data Structures*, May 14, 2007. Available: http://www.itl.nist.gov/div897/sqg/dads/HTML/euclidalgo.html. [Accessed: Jan. 18, 2010].

[6] "Diophantine equation," *Wikipedia*, Dec. 21, 2009. Available: http://en.wikipedia.org/wiki/Diophantine_equation. [Accessed: Jan. 18, 2010].

[7] "Pell's equation," *Wikipedia*, Dec. 29, 2009. Available: http://en.wikipedia.org/wiki/Pell%27s_equation. [Accessed: Jan. 18, 2010].

Java Implementation Notes

The implementations described previously in this document aren't implemented as separate standalone programs, but as a set of "solver" classes, instantiated and invoked by a single Java program class. In addition, there's another class, `Common`, which defines static methods used by several of the solvers. The `Common` class also defines a nested interface, `Common.Solver`, which is the interface implemented by each of the solver classes.

The Java implementation consists of six files, containing six classes and one interface:

- `Main.java`
  - `Main`

    This is the Java program that invokes the solver classes and displays the results. Each solver class produces the same result; the difference between them is apparent in the time required to solve the problem.

    This class can be invoked with a command line argument, overriding the exclusive upper bound on the perimeter of the Pythagorean triangles. By default, this upper bound is 100,000,000.

- `Common.java`
  - `Common`

    This class includes methods for determining whether a pair of numbers is coprime, and for computing the number of derivative Pythagorean triples within a specified perimeter limit.

  - *`Common.Solver`*

    This interface defines a single function, `countTileableTriples`. Each of the implementing classes (i.e. the solver classes) implement the relevant solution algorithm with this method.

- `Euclid.java`
  - `Euclid`

    Implements the algorithm described in "Generating Pythagorean Triples with Euclid's Formula" (page 3).

- `Constrained.java`
  - `Constrained`

    Implements the algorithm described in "Adding Constraints for More Efficient Generation of Primitive Pythagorean Triples" (page 5).

- `Pell.java`
  - `Pell`

    Implements the algorithm described in “Solution via Pell's Equation” (page 8).

Two files are attached to this document:

- `Tiling.jar`

  This is the executable .jar file, containing the compiled class files. It can be executed from a command line, by typing the following (case-sensitive) command:

  ```
  java -jar Tiling.jar
  ```

  To override the default perimeter upper bound, pass the desired value as a command line parameter. For example, to specify that the Pythagorean triangle perimeters should be less than 1000, use this command:

  ```
  java -jar Tiling.jar 1000
  ```

- `Tiling-src.zip`

  This archive contains the Java source code and the Javadoc-generated documentation.

The code listings that follow don't include any code comments. Also, some formatting and other non-critical details have been modified slightly, to simplify the task of including the code in this document. In the event that there are questions on the program operation, the attached source code files should be consulted in addition to the code listings here.

Main.java

```
package org.supercomputingchallenge.minichallenge.tiling;

public class Main {

    private static final long DEFAULT_BOUND = 100000000;
    private static final String PREAMBLE = "Searching for Pythagorean " +
        "triples satisfying tiling condition, with a perimeter less than %d:\n";
    private static final String RESULT =
        "\tClass %s found %d triples, in %d ms.\n";

    public static void main(String[] args) {
        long bound;
        if (args.length > 0) {
            bound = Long.parseLong(args[0]);
        }
        else {
            bound = DEFAULT_BOUND;
        }
        System.out.printf(PREAMBLE, bound);
        solveAndDisplay(new Euclid(), bound);
        solveAndDisplay(new Constrained(), bound);
        solveAndDisplay(new Pell(), bound);
    }

    private static void solveAndDisplay(Common.Solver solver, long bound) {
        long start = System.currentTimeMillis();
        System.out.printf(RESULT, solver.getClass().getSimpleName(),
                solver.countTileableTriples(bound),
                System.currentTimeMillis() - start);
    }

}
```

Common.java

```
package org.supercomputingchallenge.minichallenge.tiling;

public class Common {

    public static boolean coprime(long num1, long num2) {
        long lower = Math.min(num1, num2);
        long higher = Math.max(num1, num2);
        while (lower > 1) {
            long swap = lower;
            lower = higher % lower;
            higher = swap;
        }
        return (lower != 0);
    }

    public static long countDerivatives(long m, long n, long bound) {
        long perimeter = 2 * m * (m + n);
        return (bound - 1) / perimeter;
    }

    public static interface Solver {
        long countTileableTriples(long bound);
    }

}
```

Euclid.java

```
package org.supercomputingchallenge.minichallenge.tiling;

public class Euclid implements Common.Solver {

    public long countTileableTriples(long bound) {
        long count = 0;
        for (long m = 2; m <= (long) Math.sqrt(bound / 2); m++) {
            for (long n = 1 + (m & 1); (n < m) && (2 * m * (m + n) < bound);
                    n += 2) {
                if (Common.coprime(m, n)
                        && (((2 * m * (m + n))
                            % Math.abs(m * m - n * n - 2 * m * n)) == 0)) {
                    long derivatives = Common.countDerivatives(m, n, bound);
                    count += derivatives;
                }
            }
        }
        return count;
    }

}
```

Constrained.java

```
package org.supercomputingchallenge.minichallenge.tiling;

public class Constrained implements Common.Solver {

    private static final double LIMIT_FACTOR = Math.sqrt(Math.sqrt(2)) / 2;

    public long countTileableTriples(long bound) {
        long count = 0;
        long limit = 1 + (long) (LIMIT_FACTOR * Math.sqrt(bound – 1));
        for (long m = 2; m <= limit; m++) {
            long n = (long) Math.ceil(Math.sqrt(2 * m * m - 1)) - m;
            if (Common.coprime(m, n)
                    && (Math.abs(m * m - n * n - 2 * m * n) == 1)) {
                long derivatives = Common.countDerivatives(m, n, bound);
                count += derivatives;
            }
        }
        return count;
    }

}
```

Pell.java

```
package org.supercomputingchallenge.minichallenge.tiling;

public class Pell implements Common.Solver {

    public long countTileableTriples(long bound) {
        long count = 0;
        long initialM = 2;
        long initialN = 1;
        long m = initialM;
        long n = initialN;
        while (2 * m * (m + n) < bound) {
            long nextM = initialM * m + initialN * n;
            count += Common.countDerivatives(m, n, bound);
            n = initialM * n + initialN * m - 2 * initialN * n;
            m = nextM;        }
        return count;
    }

}
```