Ciaran Dunne
F28PL - Question I

Question 7: Pythagorean Triples

For the Pythagorean triples question, my initial idea was to simply use 3 for loops to find a, b and c, such that $a^2 + b^2 = c^2$. As a maths student, this felt extremely inefficient. So I decided to do some research on the subject of generating Pythagorean triples.

I found that Euclid's formula states that a, b and c can be determined by two arbitrary integers, such that $m > n > 0$, m and n are coprime (in concrete terms, $gcd(m, n) = 1$), and either m or n is even. Then $a = m^2 - n^2,\ b = 2mn,\ c = m^2 + n^2$

The reason for m or n being even makes sure that the triple is *primitive,* this means that it is unique, for example, $(6,\ 8,\ 10)$ can be made by multiplying $(3,\ 4,\ 5)$ by 2, in my eyes, triples that are not primitive are pointless to calculate.

To generate the coprime tuples (m, n), I found there is also a concrete method for doing so, using two disjoint ternary trees, one starting at $(2, 1)$ (for even-odd, and odd-even pairs) and the other at $(3, 1)$ (for odd-odd) pairs. However, due to our previous definition of (m, n), one of them must be even, therefore we do not need the second tree.

Initially, I was going to try and find a tree data structure for Python, but there wasn't one built in, and as I have never used trees outside of maths, I decided against this, and started making a generator that mimicked the process and gave me the tuples I needed.

I wrote a class for a generator, with its main data structure being a list, where the coprime tuples would be stored. When initialised it would set $(2, 1)$ as the first tuple in the list. The function that does the calculation in the generator is the $branches(self, t)$ function, it takes a tuple, then returns a list of three more tuples according to the branching rules of the coprime pair tree. The main driving function of the generator is the $next()$ function, which creates a new list for the current level in the tree, and iterates through the previous level, calculating the three branches from each node, then appending that returned list to the list of coprimes on that level. The objects main list is then set to the new list, so that it is ready to be iterated through, when $next()$ is called again.

For generating the actual triples, I made a generator, when initialised, a coprime generator is initialised, and the counter is set to 0. When the $next()$ function is called (this is a different $next()$ to the one in the coprime generator), a pair is taken from the coprime generator's list of tuples, and m and n are set accordingly. If that was the last tuple in the list, then a new list of tuples will be created, and the counter set back to 0, if not, the counter will increment. a, b, c will be set by their definition of functions of $m \text{ and } n$, then the triple (a, b, c) will be returned.

In testing, the program runs exceptionally, I wrote a for loop to call $next()$ 1,000,000 times, then print the one million and first primitive Pythagorean triple. Using the *time* command in the linux terminal, I ran *time python pythag.py* to time the execution of my program. The results were as follows:

```
>time python pythag.py
(73268013, 120102916, 140687285)

real  0m4.697s
user  0m4.278s
sys   0m0.302s
```

Here is the source code for reference:

```
1. class coprimes():
2.     def __init__(self):
3.         self.coprimeList = [(2,1)]
4.
5.     def branches(self, t):
6.         m, n = t[0], t[1]
7.         return [(2 * m - n, m), (2 * m + n, m), (m + 2 * n, n)]
8.
9.     def next(self):
10.            branchList = []
11.            for node in self.coprimeList:
12.                branchList.extend(self.branches(node))
13.            self.coprimeList = branchList
14.
15.  class triples():
16.      def __init__(self):
17.          self.pair_generator = coprimes()
```

```
18.          self.i = 0
19.
20.      def next(self):
21.              pair = self.pair_generator.coprimeList[self.i]
22.              m = pair[0]
23.              n = pair[1]
24.
25.              if self.i == (len(self.pair_generator.coprimeList) - 1):
26.                  self.pair_generator.next()
27.                  self.i = 0
28.              else:
29.                  self.i = self.i + 1
30.
31.              a = m*m - n*n
32.              b = 2*m*n
33.              c = m*m + n*n
34.              return (a,b,c)
35.
36.  x = triples()
37.  for i in range(1000000):
38.      next(x)
39.
40.  print next(x)
```