General

Infinite loops

Take care to avoid infinite loops. An infinite loop is a loop that never stops executing: in most of the cases it concerns a `while`-loop where the statements inside the loop never take care to make the `while`-condition `False` after some time. As an example, take a look at the following code snippet

```
>>> i = 0
>>> a = 0
>>> while i < 4:
...     a += 1
```

Because the statement `a += 1` will never cause the initial value of the variable `i` to become larger than or equal to 4, the condition `i < 4` will evaluate to `True` forever.

Tip: If you work with Eclipse, you known that a program that was started is still running if you observe a red square in the top menu of the Console. If you click the red square, you force the program to stop.

Monkeys and coconuts

General information

Counting starts at zero

Computer scientists by default start counting from zero, not from one, and Python follows this tradition in many of its design decisions. As an example, the built-in function `range` generates a sequences of successive integers that starts at zero, if you only pass a single argument to the function. Here's how you count to 5 in Python

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
```

If you want counting to start at another value, you can pass this value as an extra argument to the `range` function.

```
>>> for i in range(1, 6):
...     print(i)
...
1
2
3
4
5
```

However, it is considered a more *Pythonic* solution to write the above as

```
>>> for i in range(5):
...     print(i + 1)
...
1
2
3
4
5
```

Specific information

There are three possibilities to describe an amount of coconuts

- no nuts
- 1 nut
- n nuts

where n is a number greater then 1. Each of these possibilities can be split into two components that are separated from each other by a space:

- amount of nuts (`no`, 1 or n)
- singular or plural form of nut ('nut' of 'nuts')

According to the **divide-and-conquer** principle, we should compute these two components separately. If we assume that the variable `n` references an integer object whose value corresponds to the amount of nuts, we can compose the string describing the amount of coconuts with the correct singular or plural form in the following way:

```
>>> amount = 'no' if n == 0 else str(n)
>>> nuts = 'nut' if n == 1 else 'nuts'
>>> f'{amount} {nuts}'
```

The following sessions illustrate this for a couple of cases:

```
>>> n = 0
>>> amount = 'no' if n == 0 else str(n)
>>> nuts = 'nut' if n == 1 else 'nuts'
>>> f'{amount} {nuts}'
'no nuts'
```

```
>>> n = 1
>>> amount = 'no' if n == 0 else str(n)
>>> nuts = 'nut' if n == 1 else 'nuts'
>>> f'{amount} {nuts}'
'1 nut'
```

```
>>> n = 4
>>> amount = 'no' if n == 0 else str(n)
>>> nuts = 'nut' if n == 1 else 'nuts'
>>> f'{amount} {nuts}'
'4 nuts'
```

Pythagorean triples

Specific information

A first attempt to tackle this assignment is to use the *brute force* technique. With this strategy, you simply try all possible values for a, b and c to see which combinations leads to a valid solution.

However, if you naively try all possible combinations (see it as an exercise to compute how many combinations there are), you will run against the time limit that we have set in Dodona. In this case this does not necessarily mean that you program got stuck in an infinite loop. It simply takes too long to evaluate all possible combinations.

You should therefore optimize the *brute force* strategy, by not evaluation those combination that can not (or no longer) lead to a valid solution. With less combinations to be evaluated, the program will automatically find the solutions of the test cases in a more efficient (faster) way. The technical term used when trying to avoid combinations that cannot lead to a solution is called *pruning*. To do so, you can use all information that is given in the problem statement !