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.

Unused variables

In case you setup Eclipse to make use of pylint (Window > Preferences > PyDev > Pylint > Use pylint?), additional markers are placed in the right margin of the code editor that indicate bad programming style. One type of markers indicates that variables have been defined that are not used in the source code. This does not make your Python code invalid, but may cause this variable to accidentally be misused in your code. Therefore it is considered bad programming style.

The above sitation may occur, for example, when you combine a `for`-loop in combination with the `range` function to execute a sequence of statements a fixed number of times. Because the syntax of the `for`-loop requires you to always use a loop variable, you may have to use a variable that is not necessarily used in your source code. If this is the case, it is commonplace to use an underscore (`_`) as the variable name, which implicitly expresses in the Python community that *this is a variable that will not be used in the source code*. Pylint will take this into account, and will never indicate that an underscore is a variable that is not used in the code. The code snippet

```
>>> count = 0
>>> for i in range(4):
...     count = 2 * count + 1
...
```

may thus better be rewritten as

```
>>> count = 0
>>> for _ in range(4):
...     count = 2 * count + 1
...
```

Heat Wave

General information

Premature abortion of loops

In Python you can use the statements `break` and `continue` to abort a loop before it has come to completion. In general, however, these statements are considered bad programming style.

One situation where you may want a premature abortion of a loop occurs when you want to find a solution by trying all possible cases, and stop as soon as one solution has been found. Instead of using `break` or `continue` in this case, it is better to use an additional Boolean variable that indicates whether or not the solution has already been found.

```
>>> found = False
>>> while not found:
...     if (solution found): #solution found represents a condition
...         found = True
...
```

As soon as the solution has been found (represented here by the fact that the condition *solution found* evaluates to `True`), the variable `found` is assigned the value `True`. As a result, the `while`-loop ends the next time the `while`-conditions is evaluated after the current iteration.

Specific information

In this assignment it is a good idea to count the number of successive **summer days** that have been observed in the passed period, and how many **tropical days** have been observed in the passed period of successive summer days.

Three wise men

General information

Avoid floating point computations (if possible)

Because computer may always make rounding errors while computing with *floating point* numbers, it is recommended to avoid using *floating point* numbers whenever you can do the same computations with integers.

For example, if you know beforehand that all real-valued numbers a program needs to process will have two decimal digits (e.g. euro or dollar values), all it requires is to multiply these numbers by one hundred to convert them into integers. After you have done the computations in integer space, you can convert the result back to floating point numbers.

```
>>> a_float = 5.33
>>> b_float = 2.5
>>> a_float + b_float
7.83
>>> a_int = round(100 * a_float)
>>> b_int = round(100 * b_float)
```

```
>>> sum_int = a_int + b_int
>>> sum_int
783
>>> sum_int / 100
7.83

>>> a_float * b_float
13.325
>>> product_int = a_int * b_int
>>> product_int
133250
>>> product_int / 100         # wrong conversion of result to float !!!
1332.5
>>> product_int / (100 ** 2)
13.325
```

Note that we have use the built-in function `round` in the above example to convert real-valued numbers to integers, and not the built-in function `int`, for the specific reason that we want to take into account rounding errors. These rounding errors are caused by the fact that computers cannot accurately represent real-valued numbers. Rounding errors especially are dangerous when you start multiplying real-valued number.

```
>>> number = 69.96
>>> 100 * number
6995.999999999999
>>> int(100 * number)
6995
>>> round(100 * number)
6996
```

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 !