Palindrome Fibonacci Sequences

Alessandro Luongo
alessandro@luongo.pro

March 2, 2016

Contents

1 Problem's definition

From a theoretical perspective, it is required to write a *parser* that matches words of a given language. Let's start by defining which should be the strings (or words) accepted in our language.

Definition 1.1 *A contiguous sequence of unsigned integers is considered a "Fibonacci palindrome" if two conditions hold:*

1. *the sequence is the same whether read backward or forward.*
2. - *either the sequence has fewer than three elements,*
   - *or every contiguous three-element sequence a, b, c in it satisfies at least one of these conditions:*
     - $a = c$,
     - $a + b = c$,
     - $a = b + c$.

1.1 Request

Write a python function that takes as input a non-empty sequence of unsigned integers and efficiently finds the longest Fibonacci palindrome in that sequence. If there are multiple Fibonacci palindromes of the largest length, finding any one of them is sufficient. For the given sequence of numbers (list) the return should be (startIndex, length), where startIndex is the index at which the largest Fibonacci Palindrome starts and length its length.

1.2 Observations

Note that palindromic sequences can have Fibonacci sub-sequences and vice versa. Let's define a few particular cases.

- In case the sequence is empty, we return (0, 0) [1]
- In case the sequence's length is 1, we return (0,1)
- In case the sequence is palindrome of length 2, we return (0,2)
- In case the sequence is not palindrome, but it's length is 2, we return either (0,1) or (1,1)

Otherwise, the previous rules must be respected.

We remark that, since we don't have further condition on the first number of the Fibonacci sequence, the numbers in the sequence *can be others than the known Fibonacci's numbers.* Moreover, the presence of the condition $a = c$ somehow "weaken" our language: there are words which can be "unexpected" at first sight (and therefore are good candidates for testing). For instance, the sequence:

$$[..., 1, 10, 1, 10, 1, 10, 1, ...]$$

is a word in our language, since:

1. it is palindrome
2. being longer than three elements, for each a, b, c in sequence, the condition $a == c$ holds.

[1]I prefer to have a consistent return type for each function. In this case, since our result (0,0) is no longer used as an input to other functions, it can be acceptable to return also None or (None, None), but I think it is more elegant to specify a 0 as a second value to specifically point that the longest valid sequence has length 0, that is, it does not exists.

Another important remark, is that some other sequences we expect in our language, actually aren't included. Take the following symmetric sequence of Fibonacci numbers:

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]$$

.

Now let's see what happen to the triplets from position 9 and forward:

- $[34, 55, 89]$: valid. $34 + 55 = 89$
- $[55, 89, 89]$: *not valid.*
- $[89, 89, 55]$: *not valid.*
- $[89, 55, 34]$: valid.

The problem relay in the fact that there are two "central elements" in our sequences, and our Fibonacci conditions cannot be satisfied by these kind of sub-sequences. More formally: sequences of even length are in our language only if they are trivial (i.e. all the numbers are equal). This is an important information for testing as well.

2 Algorithms and Software

Formally, we are writing a parser for a language that accept only a strings which are part of two different languages: the language of palindrome sequences and the language of Fibonacci sequences. We can think of it as a composition of two different functions: one that select the palindrome subsequences, and the other that filter sequences that are Fibonacci successions. We can either find the interception between the results of two different function (which might help code readability), or we can apply those different filter in succession (which surely will help efficiency). The software was developed using python 3.4.

2.1 Analysis

This is an average case analysis, results may vary if we have further information on the input.

The easiest algorithm for finding palindromic sub-sequences is $O(n^2)$ (i.e., check all the possible sub-sequences). There are more efficient algorithms, but I have preferred to start by filtering sub-sequences that match the Fibonacci condition, and then check if the sub-sequence is palindrome.

I have decided to run first the most efficient algorithm in order to run the slower on the least amount of input. My guess before writing the code was that Fibonacci could be done in $O(n)$ (where n is the length of the sequence). In this way, I have to check for symmetry only a limited number of sequences, and this can be done in $O(m)$ where m is the number of sequences that satisfy the Fibonacci condition. Last but not least, in this way, code can be kept simple, since checking for symmetry is really simple.

Please note that, in the case of multiple correct solution, the output of the algorithm is strictly implementation dependent. The reason behind this is that max return the first maximum element of an iterable, and this may change according to the algorithm used to create the data structure.

2.2 Solution 1: recursive style

Recursion offer often an efficient approach to solve many problems. For this algorithm, the function *find_all_fibonacci1* is called multiple times on various item of the sequence, until is not possible to unpack more than three elements from the string (that is, the sequence is finished). To help me debug this function, I have used one of my decorator in my "toolset". The decorator *@Trace* print at screen useful information for recursive function (i.e. the calling value and the return value of the recursive function).

2.3 Solution 2: functional style

Often functional programming offer an elegant way of solving certain kind of problems. In *find_all_fibonacci2* I made the following steps:

1. I start by selecting all the first elements of the triplets matching any the Fibonacci condition of 1.1.

2. Then I devised a way for cluster contiguous sequences of matching triplets. (using the *groupby* function)

3. Then, for each cluster, I create all the valid pair $(index, offset)$.

4. At last, I add all the possible sequence shorter than three elements.

2.4 Solution 3: classic style

I decided to write also a classic version of the algorithm. This is the simplest approach as possible. Perhaps it is not very "pythonic", but it can be surprisingly clear.

It iterates through the sequence looking for a match for one of the Fibonacci conditions. Once it is found, it calls the keep_matching function. The function *keep_matching* keep looking for matches in the sequence until a non-matching triplets is found. It return all the possible symmetric sub-sequences within the match. Specifically, this function is in charge of:

- keeping iterating through the string until the last match is found
- generate the sub-sequences, *starting from the longer*,
- test from symmetry.

The function *create_combinations* return all the possible slicing of a sequence using (start, offset) notation.

Finally, I take care of all sequences of length 1 and 2, taking care of symmetry in the latter case.
Since this solution has "embedded" everything, there's no need to call a function to select the palindromic sub-sequences, nor to select the maximum among the results. Everything is already inside here.

3 Correctness and Tests

There is a module that tests all different approaches, even with 1000 sequences of *random* integers. All test are marked as passed. The choice of using a testing module instead of other solution is arbitrary.

While developing this software I reached a point where all my tests were passed. I then devised a new test, where I matched the results of the *find_all_fibonacci1* with *find_all_fibonacci2* (i.e. *test_partial_1_2* function). Thanks to this tests I discovered two things. First, that Solution 1 was returning more wrong (but un-harmful) results (i.e., solutions of length 0) and than, that Solution 2 was actually losing some valid results.