Daily Coding Problem #275

Problem

This problem was asked by Epic.

The "look and say" sequence is defined as follows: beginning with the term 1, each subsequent term visually describes the digits appearing in the previous term. The first few terms are as follows:

```
1
11
21
1211
111221
```

As an example, the fourth term is 1211, since the third term consists of one 2 and one 1.

Given an integer N, print the Nth term of this sequence.

Solution

One way to go about this is to repeatedly apply the "look and say" principle to a running result, and stop once we have completed N iterations.

For a given result, we should loop through each group of identical characters, find how many of them there are, and add the count and character to a new result string. Each time we hit a new character, we reset the count to one.

```
def look_and_say(num):
    result = ''
```

```
    first = num[0]
    remainder = num[1:] + ' '
    count = 1

    for char in remainder:
        if char != first:
            result += str(count) + first
            count = 1
            first = char
        else:
            count += 1

    return result

def compute(num):
    result = '1'
    for i in range(1, num):
        result = look_and_say(result)

    return result
```

We can make this code more succinct, at some expense to readability, by using Python's `itertools` module to group similar characters.

```
import itertools

def look_and_say(num):
    result = '1'

    for _ in range(num - 1):
        tmp = ''
        for char, group in itertools.groupby(result):
            count = len(list(group))
            tmp += (str(count) + char)
        result = tmp

    return result
```

Unfortunately, there is no simple trick to quickly find the Nth term in this sequence. There

is in fact a complicated formula describing how components of each term can be considered independently, involving Conway's cosmological theorem, but this is not likely to be demanded in any coding interview.

Each iteration of `look_and_say` requires us to analyze each character of the running result, so we can consider our time complexity to be `O(N * M)`, where `N` is the number of terms and `M` is the length of the longest string analyzed. Since the length of the string grows by about thirty percent in each iteration, `M` is exponential in the number of terms, and so the overall complexity of this algorithm is $O(N * 2^N)$.
