38. Count and Say

Feb. 2, 2020 | 16.6K views

Average Rating: 3.67 (18 votes)

The count-and-say sequence is the sequence of integers with the first five terms as following:

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

`1` is read off as `"one 1"` or `11`.
`11` is read off as `"two 1s"` or `21`.
`21` is read off as `"one 2`, then `one 1"` or `1211`.

Given an integer n where $1 \le n \le 30$, generate the n^{th} term of the count-and-say sequence. You can do so recursively, in other words from the previous member read off the digits, counting the number of digits in groups of the same digit.

Note: Each term of the sequence of integers will be represented as a string.

Example 1:

```
Input: 1
Output: "1"
Explanation: This is the base case.
```

Example 2:

```
Input: 4
Output: "1211"
Explanation: For n = 3 the term was "21" in which we have two groups "2" and "1", "2"
```

Solution

Overview

First of all, we would like to apologize to our audiences that the description of the problem is definitely not crystal clear, as many people have raised the issue in the discussion forum.

The problem should have been rather easy, as it is labeled, should it be stated clearly. That being said, let us consider the "*mysterious*" nature of the problem description as part of the challenge.

> Imagine in a more challenging scenario, one might be given just a sequence of numbers without any explication, and one is asked to produce the next numbers, which would require one to figure out the hidden pattern about the generation of the sequence.

Now, the problem becomes more intriguing. And maybe some of you might want to pause for a moment to figure out the puzzle first before proceeding to the clarification.

Sequence Puzzle

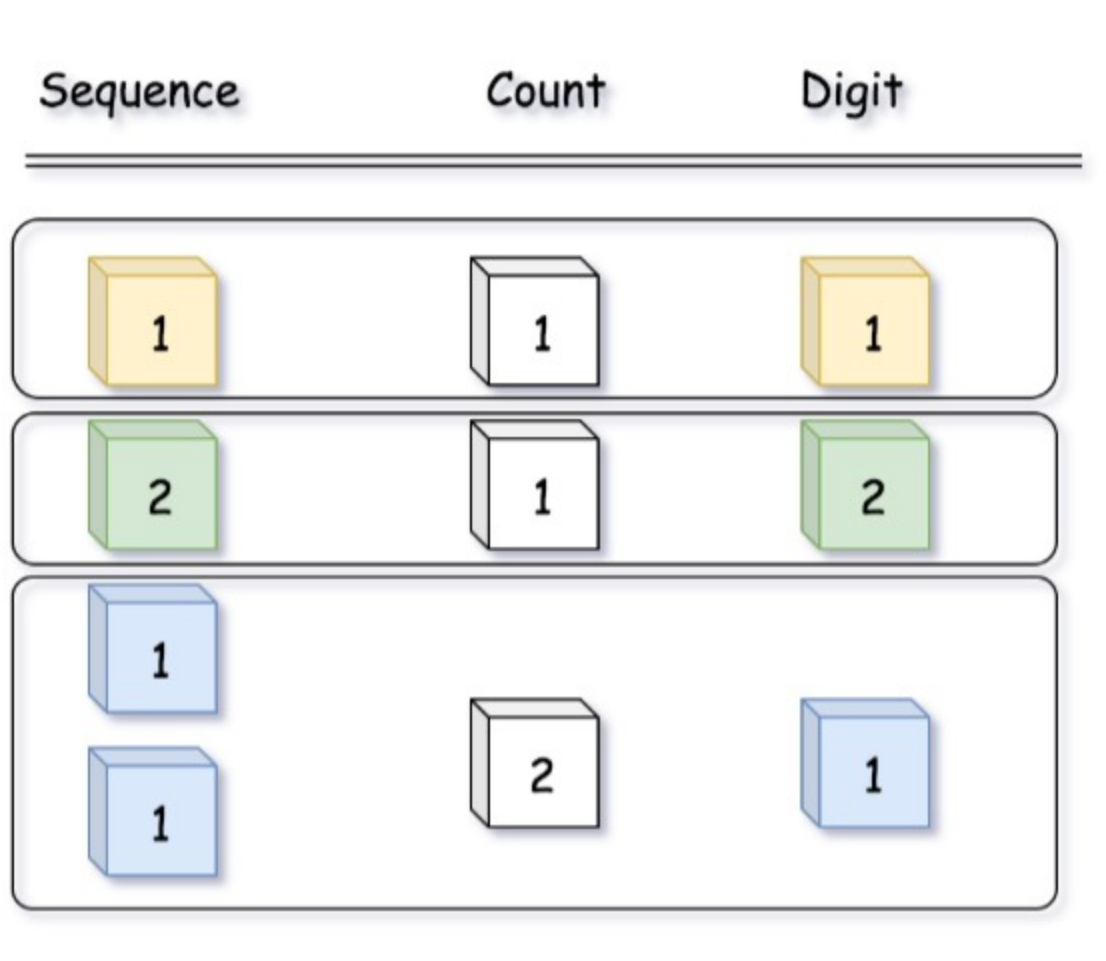
Before we proceed to the solutions, in this section, we would like to rephrase the problem.

Given two adjacent sequences of number, $[S_n, S_{n+1}]$, there exists a pattern that one can produce the sequence S_{n+1} from its previous sequence S_n.

> More specifically, one can consider the sequence S_{n+1} as a sort of ***summary*** to its previous sequence S_n, *i.e.* S_{n+1} contains a list of pairs as $|\text{count}, \text{digit}|$ which ***encodes*** all the information about its previous sequence S_n.

Let us take the sequence $S_4 = 1211$ as an example, from left to right, we then can divide the sequence into three sub-groups where each sub-group contains a list of identical and adjacent digit, *i.e.* $S_4 = \{1\}\{2\}\{11\}$, as shown in the following:

We then *count* the number of digits in each sub-group, and then output the summary in the format of $|\text{count}, \text{digit}|$. As the end, we would obtain the exact sequence of S_5.



With the generated sequence S_5, we then ***recursively*** apply the above rule to generate the next sequence.

Now that the description of the problem is clear, one might dismiss it as yet another strange and artificial problem to solve. Well, it is not true in this case.

Actually, we could consider this problem as a ***naive compression algorithm*** for a sequence of numbers. Instead of storing repetitive adjacent digits as they are, we could summarize them a bit with the method presented in the problem, which could save us some space as long as there are indeed repetitive occurring patterns.

Approach 1: Sliding Window

Intuition

Now that the problem has been clarified, the solution should be intuitive.

Following the rule as we described above, in order to generate the next sequence, we could scan the current sequence with a sort of ***sliding window*** which would hold the identical and adjacent digits. With the sliding window, we would divide the original sequence into a list of sub-sequences. We then count the number of digits within each sub-sequence and output the summary as pairs of $|\text{count}, \text{digit}|$.

Algorithm

Here we define a function `nextSequence()` to generate a following sequence from a previous sequence, and we *recursively* call this function to get the desired sequence that is located at a specific index.

- Within the function, we scan the sequence with two contextual variables: `prevDigit` and `digitCnt` which refers to respectively the digit that we are expecting in the sub-sequence and the number of occurrence of the digit in the sub-sequence.
- At the end of each sub-sequence, we append the summary to the result and then we reset the above two contextual variables for the next sub-sequence.
- Note that, we use an artificial delimiter in the sequence to facilitate the iteration.

Java | Python | Copy

```python
class Solution(object):
    def countAndSay(self, n):
        """
        :type n: int
        :rtype: str
        """
        return ''.join(self.nextSequence(n, ['1', 'E']))

    def nextSequence(self, n, prevSeq):
        if n == 1:
            return prevSeq[:-1]

        nextSeq = []
        prevDigit = prevSeq[0]
        digitCnt = 1
        for digit in prevSeq[1:]:
            if digit == prevDigit:
                digitCnt += 1
            else:
                # the end of a sub-sequence
                nextSeq.extend([str(digitCnt), prevDigit])
                prevDigit = digit
                digitCnt = 1

        # add a delimiter for the next sequence
        nextSeq.append('E')

```

Complexity

- Time Complexity: $\mathcal{O}(2^n)$ where n is the index of the desired sequence.
  - First of all, we would invoke the function `nextSequence()` $n-1$ times to get the desired sequence.
  - For each invocation of the function, we would scan the current sequence whose length is difficult to determine though.
  - Let us image in the worst scenario where any two adjacent digit in the sequence are not identical, then its next sequence would ***double*** the length, rather than having a *reduced* length. As a result, we could assume that in the worst case, the length of the sequence would grow *exponentially*.
  - As a result, the overall time complexity of the algorithm would be $\mathcal{O}(\sum_{i=0}^{n-1} 2^i) = \mathcal{O}(2^n)$.
- Space Complexity: $\mathcal{O}(2^{n-1})$.
  - Within each invocation of the `nextSequence()` function, we are using a container to keep the result of the next sequence. The memory consumption of the container is proportional to the length of the sequence that the function needs to process, *i.e* 2^{n-1}.
  - Though we were applying the recursion function, which typically incurs some additional memory consumption in call stack. In our case though, the recursion is implemented in the form of ***tail recursion***, and we assume that the compiler could optimize its execution which would not incur additional memory consumption.
  - One could also easily replace the recursion with the iteration in this case.
  - As a result, the overall space complexity of the algorithm would be dominated by the space needed to hold the final sequence, *i.e.* $\mathcal{O}(2^{n-1})$.

Approach 2: Regular Expression

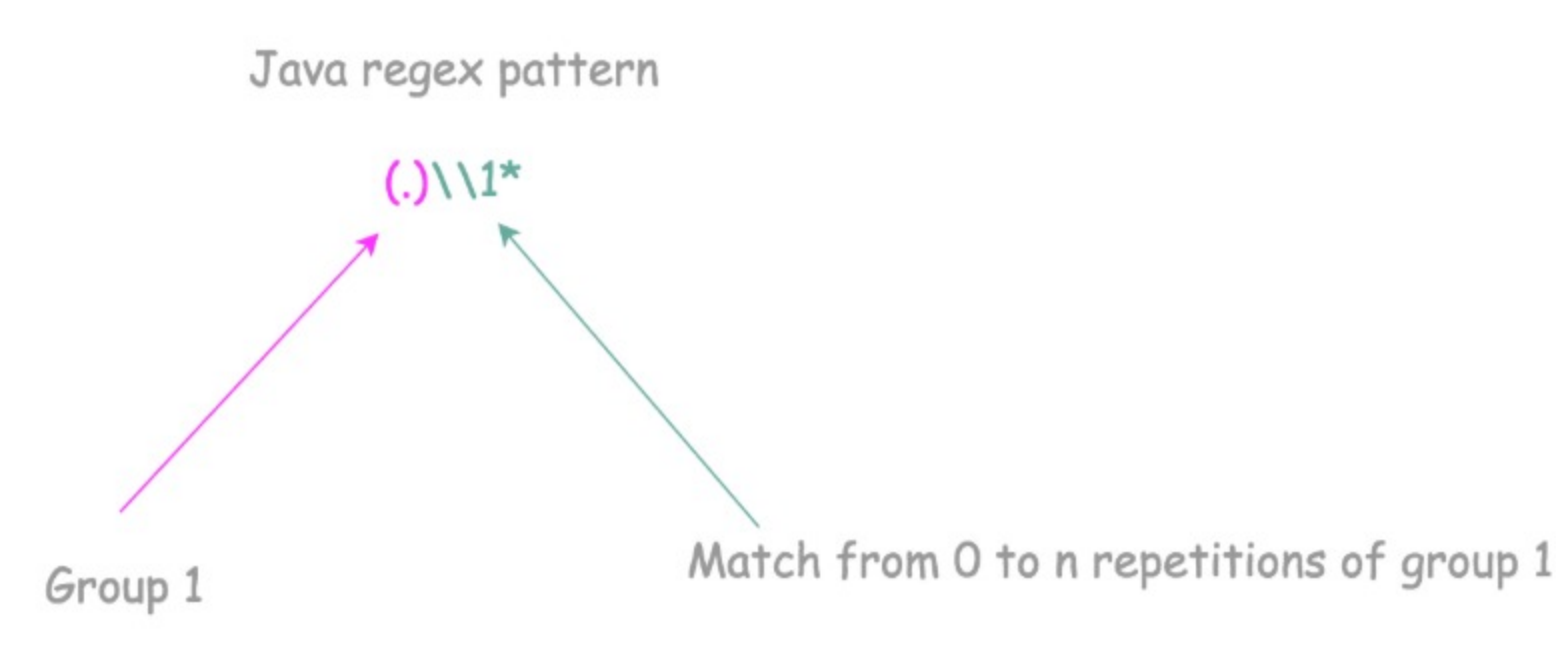
Intuition

This problem could be a good exercise to apply ***pattern matching***, where in our case we need to *find* out all those *repetitive* groups of digits.

A regular expression (*a.k.a regex*) is a sequence of characters that defines a ***search pattern***. The regex serves as a common tool for many pattern matching problems. And many programming languages provides regex capacities either with build-in constructs or via some libraries.

Note that, although the syntax of regex is mostly universal across all programming languages, there might exist some subtile differences. Here we show two examples in Java and Python respectively.

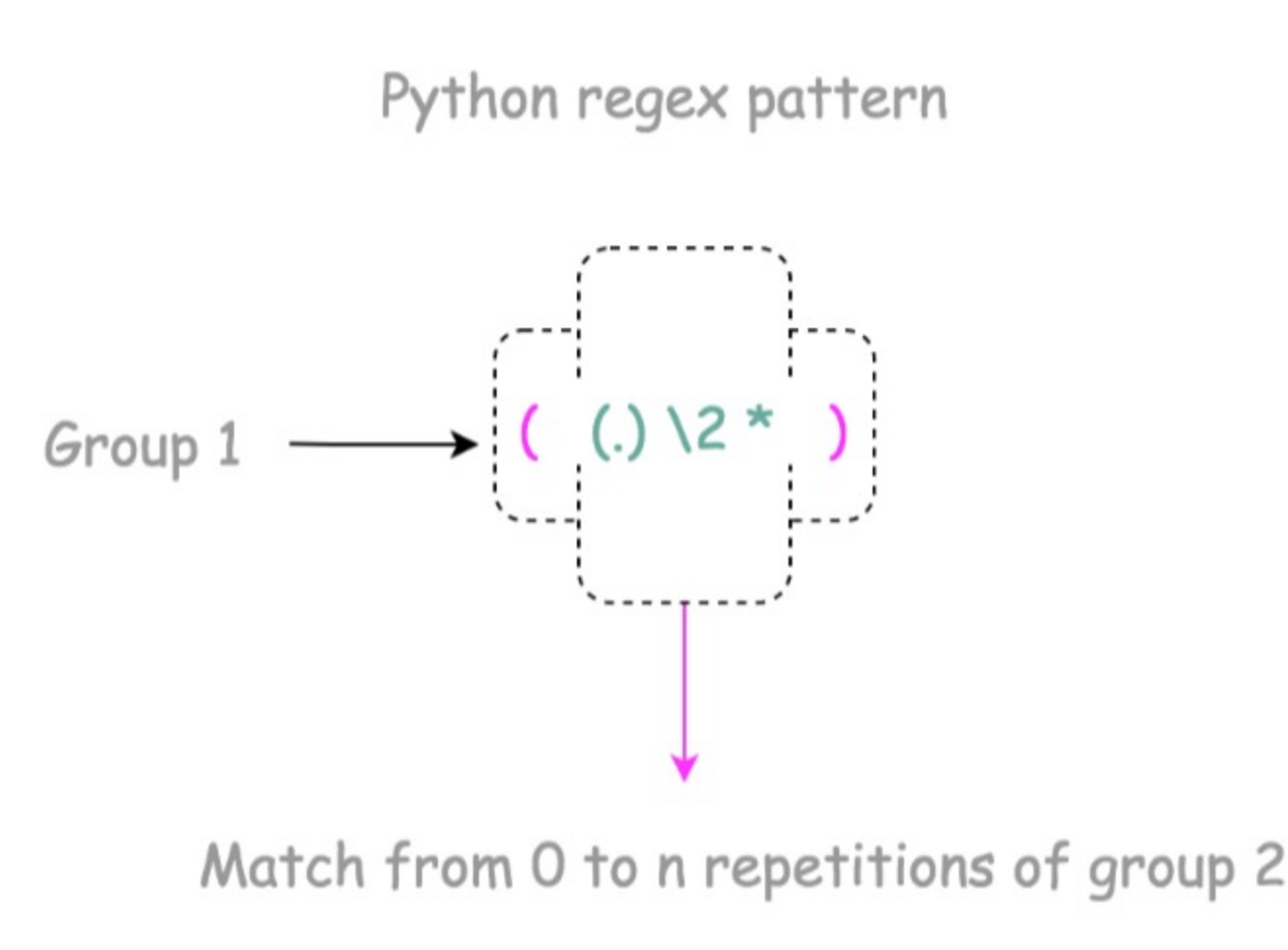
> Java: `regex = "(.)\\1*"`



We could break down the above regex expression into three 3 parts:

- "(.)": it defines a *group* that contains a single character that could be of anything.
- "\1": this part refers to the defined group with the index of 1.
- "*": this qualifier followed by the group reference `\\1`, indicates that we would like to see the group repeats itself *zero* or more times.

> Python: `regex = "((.)\2*)"`



Slightly different than the above regex in Java, here we define two *groups* instead of one.

- "(.)": again, this is a *group* that contains a single character that could be of anything.
- "\2": this part refers to the second group (*i.e.* `(.)`) that we define.
- "((.)\2*)": the outer bracket defines the scope of the *first* group, which contains the repetitive appearance of the second group above.

Algorithm

Based on the above definitions of regex, we then find all the matches to the regex and then concatenate the results together.

Java | Python | Copy

```python
class Solution(object):
    def countAndSay(self, n):
        """
        :type n: int
        :rtype: str
        """
        import re

        currSeq = '1'
        pattern = r'((.)\2*)'

        for i in range(n-1):
            nextSeq = []
            for g1, g2 in re.findall(pattern, currSeq):
                # append the pair of <count, digit>
                nextSeq.append(str(len(g1)))
                nextSeq.append(g2)
            # prepare for the next iteration
            currSeq = ''.join(nextSeq)

        return currSeq
```

Complexity

- Time Complexity: $\mathcal{O}(2^n)$ where n is the index of the desired sequence.
  - Similar with our sliding window approach, the overall algorithm consists of a nested loop.
  - We could assume that the time complexity of the regex matching is linear to the length of the input string.
  - As a result, the overall time complexity of the algorithm would be $\mathcal{O}(\sum_{i=0}^{n-1} 2^i) = \mathcal{O}(2^n)$.
- Space Complexity: $\mathcal{O}(2^{n-1})$.
  - Within the function, we are using a container to keep the result of the next sequence. The memory consumption of the container is proportional to the length of the sequence that the function needs to process, *i.e* 2^{n-1}.
  - As a result, the space complexity of the algorithm would be dominated by the space needed to hold the final sequence, *i.e.* $\mathcal{O}(2^{n-1})$.