An Analysis of the Kolakoski Sequence

Hangu Andrei Calin

March 17, 2025

Abstract

The Kolakoski sequence is a self-referential sequence that only contains the numbers 1 and 2, and it is defined by its own run-length encoding. This paper explores the properties of the sequence, its historical background, and the different ways of generating it. I implemented and analyzed two Python approaches: a basic extension-based method and Nilsson's more efficient algorithm. Additionally, I introduce an optimized approach that uses lookup tables and row-by-row expansion to improve performance. This method reduces redundant calculations and speeds up the sequence generation by precomputing the transitions between segments. I discuss these methods, their connections to combinatorics and symbolic dynamics, and suggest some future research ideas on improving sequence computation.

1 Introduction

The Kolakoski sequence (1) is an infinite sequence of 1s and 2s, where each term in the sequence defines the length of the runs of 1s or 2s. What makes the Kolakoski sequence unique is that it is defined by its own run-length encoding. In simpler terms, the sequence describes itself in a way that each term indicates the number of times the previous number should repeat. This makes the sequence both self-referential and recursive, which gives it interesting mathematical properties.

The sequence begins as:

$$k = 1, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2, 2, 1, \ldots \tag{1}$$

Each run length is determined by the sequence itself, which creates a unique and unpredictable pattern. The Kolakoski sequence is also known to be aperiodic, meaning it does not repeat in any regular way, and its long-term behavior is still an area of active research.

2 Historical Background

The Kolakoski sequence was first introduced by Rufus Oldenburger in 1939 but became widely known through the work of William Kolakoski in 1965. It has

been studied in various fields of mathematics, including combinatorics, number theory, and symbolic dynamics. One of the major unsolved questions about the sequence is whether the proportion of 1s in the sequence approaches 1/2 as the sequence grows, which remains an open problem.

Over time, different methods have been used to study the sequence, including probabilistic methods and computational approaches. Johan Nilsson's work in 2012 introduced a more memory-efficient way to generate the sequence, which made it easier to compute large portions of the sequence.

3 Mathematical Properties

The Kolakoski sequence has several interesting mathematical properties and conjectures associated with it:

- The sequence does not repeat in a regular cycle, making it aperiodic.
- It is a self-referential sequence, meaning it defines itself in a way that each term depends on the previous terms.
- It is conjectured that the density of 1s in the sequence approaches 1/2, but this remains unproven.
- The sequence is cube-free, meaning it does not contain any repeated substring of the form XXX.
- The discrepancy function $\delta(n)$ is defined as:

$$\delta(n) = \sum_{j=1}^{n} (-1)^{k_j}, \tag{2}$$

  which measures how far the sequence deviates from a balanced distribution of 1s and 2s.

These properties make the Kolakoski sequence an interesting object for study, and computational methods are important for testing different conjectures about its behavior.

4 Implementation Strategies

To explore the Kolakoski sequence, we need efficient methods for generating the sequence. I tested two Python algorithms for this purpose:

4.1 Basic Extension Algorithm

The first method I implemented is based on a simple extension approach:

```
def generate_kolakoski(n):
    if n <= 0:
        return []
    if n == 1:
        return [1]
    if n == 2:
        return [1, 2]

    arr = [1, 2, 2]
    i = 2
    current_value = 1
    while len(arr) < n:
        run_length = arr[i]
        arr.extend([current_value] * run_length)
        current_value = 3 - current_value
        i += 1
    return arr[:n]
```

This method constructs the sequence iteratively by appending values based on the current sequence. While simple to implement, this method can be inefficient because it uses the 'extend()' function to add large blocks of values, which results in excessive memory usage and slower performance for large values of n.

4.2 Nilsson's Optimized Algorithm

Nilsson's approach improves the sequence generation by reducing unnecessary list operations. Instead of appending large blocks of values at once, the algorithm adds one value at a time based on the current run length, which helps minimize memory usage. This method is faster than the basic approach because it processes the sequence more efficiently and uses less memory:

```
def generate_kolakoski_nilsson(n):
    if n <= 0:
        return []
    if n == 1:
        return [1]
    if n == 2:
        return [1, 2]

    arr = [1, 2, 2]
    i, j = 3, 2
    while i < n:
        repeat = arr[j]
        value = 3 - arr[-1]
        for _ in range(repeat):
            if i >= n:
                break
```

```
            arr.append(value)
            i += 1
        j += 1
    return arr
```

This algorithm processes the sequence more incrementally and only adds the necessary number of values, resulting in a more memory-efficient solution with a space complexity of $O(\log n)$.

5 Comparison of Performance

By comparing the two algorithms, we can see the benefits of Nilsson's approach:

- The basic method relies on direct list extensions, which leads to inefficient memory use.
- Nilsson's method reduces memory waste and processes the sequence more incrementally.
- The basic method runs in $O(n)$ time but has higher space complexity due to list expansion.
- Nilsson's method also runs in $O(n)$ time but with a lower space complexity of $O(\log n)$.

Nilsson's approach is clearly more efficient for larger computations, as it reduces memory usage and performs better overall.

6 Optimized Generation Using Lookup Tables

Another method for generating the Kolakoski sequence more efficiently is by using precomputed segments and lookup tables. This approach helps reduce redundant calculations and speeds up the process by skipping over previously computed parts of the sequence.

6.1 Improvement Over Linear Methods

Traditional methods generate the sequence by processing each element sequentially, leading to redundant computations and inefficient memory use. The optimized method improves upon this by:

- Using a lookup table to store precomputed segments of the sequence.
- Using stored values to directly determine the next run length, avoiding unnecessary recalculations.
- Skipping over sections of the sequence instead of processing every element.

6.2 Mathematical Framework

Let $k = (k_1, k_2, k_3, ...)$ be the Kolakoski sequence, where each k_j defines the length of the run at that position. Instead of processing the sequence element by element, we use a lookup function T_d that maps a segment of length d to a jump length ℓ:

$$T_d(k_j, ..., k_{j+d}) = \ell, \tag{3}$$

allowing us to skip ahead and generate multiple terms in a single step.

By choosing an optimal depth for the lookup table, we can reduce the overall computational complexity from $O(n)$ to:

$$O\left(\left(\frac{2}{3}\right)^{d_{\max}} n + 2^{d_{\max}}\right), \tag{4}$$

which is much faster than the basic method.

6.3 Practical Considerations

For large values of n, memory efficiency becomes important. The lookup table requires $O(2^{d_{\max}})$ space, so the choice of $d_{\max}$ is critical. In practice, choosing $d_{\max} \approx 30$ allows computations up to $n \approx 10^{17}$ without exceeding memory limits.

This approach makes it possible to compute large parts of the Kolakoski sequence more quickly and efficiently, which is useful for applications that require handling large sequences.

7 Conclusion and Future Work

The Kolakoski sequence continues to be an interesting object of study in mathematics. Nilsson's algorithm provides an efficient way to compute the sequence, but there may still be room for further improvements. Future research could focus on:

- Investigating parallel computing methods to speed up sequence generation.
- Exploring probabilistic models to better understand the long-term behavior of the sequence.
- Using machine learning techniques to uncover hidden patterns in the sequence.

Understanding the Kolakoski sequence requires a combination of theoretical mathematics and computational methods, which is key to exploring its deep properties.

References

[1] R. P. Brent and J. Osborn, "Fast Algorithms for the Kolakoski Sequence," *ANU Research*, 2016.

[2] J. Nilsson, "A Space-Efficient Algorithm for the Kolakoski Sequence," *J. Integer Sequences*, 2012.

[3] "Kolakoski Sequence", *OEIS*, Available at: `https://oeis.org/A000002`.

[4] "Kolakoski Sequence", *Wikipedia*, Available at: `https://en.wikipedia.org/wiki/Kolakoski_sequence`.