Journal of Integer Sequences, Vol. 15 (2012),
Article 12.6.7

A Space-Efficient Algorithm for Calculating the Digit Distribution in the Kolakoski Sequence

Johan Nilsson
Fakultät für Mathematik
Universität Bielefeld
Postfach 100131
33501 Bielefeld
Germany
jnilsson@math.uni-bielefeld.de

Abstract

With standard algorithms for generating the classical Kolakoski sequence, the numerical calculation of the digit distribution uses a linear amount of space. Here, we present an algorithm for calculating the distribution of the digits in the classical Kolakoski sequence that uses logarithmic space and still runs in linear time. The algorithm is easily adaptable to generalized Kolakoski sequences.

1 Introduction

The classical Kolakoski sequence $K = (K_n)_{n=1}^{\infty}$ is the unique sequence over the alphabet $\{1, 2\}$, starting with 1 and having the sequence of run lengths identical with itself. The classical Kolakoski sequence was first studied in a work by Oldenburger [10], where it appears as the unique solution to the problem of a trajectory on the alphabet $\{1, 2\}$ which is identical to its exponent trajectory. The name of the Kolakoski sequences, however, originates from [7, 8]. In the On-Line Encyclopedia of Integer Sequences [14] the Kolakoski sequence has entry number A000002. The first letters of K are

$$\begin{array}{cccccccccccccccc}
K = & 1 & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{2} & 1 & 1 & \multicolumn{2}{c}{2} & 1 & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{2} & \ldots \\
 & | & \multicolumn{2}{c}{\wedge} & \multicolumn{2}{c}{\wedge} & | & | & \multicolumn{2}{c}{\wedge} & | & \multicolumn{2}{c}{\wedge} & \multicolumn{2}{c}{\wedge} & \\
K = & 1 & 2 & 2 & 1 & 1 & 2 & 1 & 2 & 2 & 1 & 2 & 2 & 1 & 1 & \ldots
\end{array} \tag{1}$$

There are several interesting questions, answered and unanswered, on the properties of the classical Kolakoski sequence; Kimberling presents several of these in [6]. One of the simplest, and yet unresolved, questions is that of the distribution of digits in K. If we let o_n be the number of 1s in K up to and including position n, that is $o_n = |\{i : K_i = 1, 1 \leq i \leq n\}|$, then the conjecture is

Conjecture 1. The limit $o := \lim_{n\to\infty} \frac{o_n}{n}$ exists and equals $\frac{1}{2}$.

Both parts of Conjecture 1 — the existence and the value — are still open. Several aspects of the conjecture (along with other properties and questions regarding the Kolakoski sequence) are considered by Dekking in [3, 4, 5]; see also the survey by Sing [13] and further references therein.

While an analytic solution to the conjecture remains elusive, one may at least estimate the value of o numerically by generating a large number of digits of the sequence K. In [15] Steinsky describes a recursion that generates the letters K_n and uses it to find one such numerical estimate, for $n = 3\cdot 10^8$. It is worth noting that a straight-forward implementation of Steinsky's recursion leads to an algorithm that either runs in exponential time or requires a linear amount of space. For some time, Steinsky's result raised doubt as to the validity of Conjecture 1; however, subsequent work by Monteil [9] suggested once again that the conjecture should hold. Monteil used a brute-force method, requiring linear time and linear space in n, to push the calculation to $n = 10^{11}$.

The brute-force, or straightforward, method to find o_n generates a prefix of length n of the sequence K, using the intuitive method suggested by (1). That is, starting from a suitable initial sequence, we step through and read off the symbols one by one, with each letter telling us what to write in the sequence beneath, and thus what to append to the end of the current sequence.

Here we present an algorithm to find o_n that runs in linear time, yet only uses logarithmic space. It is thus more efficient than those presented previously, and we have used it to push the calculation further; here we present values of o_n up to $n = 10^{13}$ (Table 1). Our calculation indicates that Conjecture 1 should hold, but once again gives no definite answer. We present our algorithm in Section 2 and state and prove the algorithm's time performance in Section 3. In Section 4, we briefly remark on our algorithm's adaptability to more general Kolakoski sequences, and finally in Section 5 we present the results of our calculations.

2 The Algorithm

Here we present an algorithm for calculating the number of 1s and 2s in the classical Kolakoski sequence K up to a position n. Our algorithm is more memory-efficient than the straight-forward algorithm for finding K_n; it requires only $O(\log n)$ space (Proposition 4) compared to the $\Theta(n)$ for brute-force algorithm. Here we use the standard asymptotic notation. That is, we write $f = O(g)$ if there is a constant c such that $f(n) \leq c\, g(n)$ for all sufficiently large n. Similarly $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$. (For more about asymptotic notation, see [2].) The running time of our algorithm is $O(n)$ to find o_n (Proposition 5); this is the same as for the brute-force method.

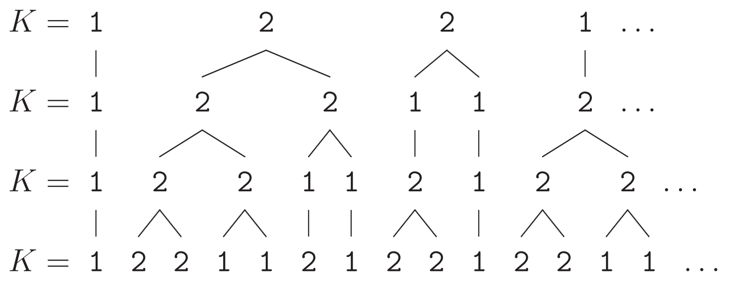
K = 1 2 2 1 ...
K = 1 2 2 1 1 2 ...
K = 1 2 2 1 1 2 1 2 2 ...
K = 1 2 2 1 1 2 1 2 2 1 2 2 1 1 ...

Figure 1: The tree structure in the Kolakoski sequence.

The idea in our algorithm is that if we set out only to find o_n, we do not have to save the complete sequence up to position n when stepping through the sequence K. As in the intuitive way of generating K, we look back at a previous position to see which symbol run to append. However, this previous position is itself determined by a letter even further back, and so on. If we keep track only of these positions that we "look back at", we can drastically reduce the amount of space needed by the algorithm.

To get a hint of how this can be done, we take as a starting point a scheme, as in (1). We see that the upper row defines (or conversely, may be defined as) the run lengths of the symbols in the lower one. We expand this scheme by adding more rows above and connecting each symbol to the symbol in the row above that has (via run length) generated it. In this way, we obtain a tree structure, as illustrated in Figure 1.

We may thus interpret the letters in the classical Kolakoski sequence K as the leaves of a tree (the leaves are the symbols in the bottom row in Figure 1). Each internal node in this tree structure is a symbol in an upper row interpreted as a run length. Each letter is connected to the letter above that has generated it (called an ancestor), and also to the letter(s) below that it generates, termed children. This tree structure continues upwards without bound as we step through the symbols of the Kolakoski sequence. However, we only need to go up in the tree until we find an ancestor, to the leaf we are currently looking at, at a left most position.

From this point on we shall consider the sequence K', defined by $K = \mathtt{12}K'$. This simplifies matters somewhat, as we do not then have to deal with the left most 1s at each height in the tree, and we do not have to deal with the leaves in the first level of the tree separately.

The algorithm for finding o_n can concisely be described as an "in-order traversal" of this tree structure, where we start from the lower left, and where we keep track of the symbols we see in the leaves during the traversal. While traversing, we add new ancestors when needed; that is we build the tree as we traverse it. To reduce the memory requirement, we dynamically generate and keep track only of the part of the tree that we currently use for the traversal. While doing so, we store the ancestors along with an indicator that tells us which of its children we have already traversed. To this end, we introduce pointers P_k, which are assigned values from the set $S = \{\mathtt{1}, \mathtt{2}, \mathtt{11}, \mathtt{22}\}$. Note that here, a run is defined as word from the set S. At any given time, the pointer P_0 holds the current run in the leaves and P_1 holds the ancestor to P_0. Similarly, any P_k that has been initiated holds the ancestor to

P_{k-1}.

We say here that pointers "hold" and not "are" a run because P_k may contain more than just the single-symbol ancestor of P_{k-1}, it may also contain a sibling of P_k. Here we refer to the single symbols (that is, 1s or 2s) of a two symbol run (11 or 22) as siblings.

The algorithm can now be described as follows.

Algorithm 2.

- To increment (or to assign a new value to) the pointer P_k we proceed as follows. Firstly, if P_k has not been initialized, let $P_k = 22$. If P_k contains two symbols then remove one of the symbols in P_k.

  If, on the other hand, P_k contains only one symbol, then increase P_{k+1} recursively. When this increment is done, the new run to write in P_k is of the length given by the first symbol in P_{k+1} and the run to write has symbol(s) opposite to the symbol(s) previously held by P_k. Note that here we do not remove the first symbol of P_{k+1} when we return from the recursion.

- To step through the sequence K (from its second symbol onwards) and calculate o_n, we repeatedly increment the pointer P_0 and keep track of the number of 1s and 2s that we see. $\diamond$

Note that for a given run contained in P_0, the algorithm will generate only the pointers $P_1, \ldots, P_N$ to P_0, where the ancestor in P_N is at the left most position in the sequence K'. (And it is this height N that we shall shortly show is of the order of $\log n$ when P_0 holds the nth letter in the sequence). As we step through the algorithm, we shall see that the successive runs held by the pointer P_0 (and also for other P_k) are the symbols in the sequence K'.

In pseudo-code the increment of P_0, (or the step by step traverse of the leaves), would be done with the recursive call of the procedure `IncrementPointer` as presented below.

```
// Increments the pointer at height k, and returns the current symbol.
// Succesive calls to IncrementPointer(0) will return the next symbol
// in the Kolakoski sequence from the third term onward.

int IncrementPointer(int k)
{   if(P[k] has not been initiated)
    {   P[k] = 22
    }

    if(P[k] == 11)
    {   P[k] = 1
        return 1
    }else if(P[k] == 22)
    {   P[k] = 2
        return 2
    }else if(P[k] == 1)
    {   P[k] = increment(k+1) == 1 ? 2 : 22
        return 2
    }else if(P[k] == 2)
    {   P[k] = increment(k+1) == 1 ? 1 : 11
        return 1
    }
}
```

To illustrate how the algorithm works, we now present a run through of its initial steps.

Example 3. Incrementing the pointer P_0 once is done through the following procedure;

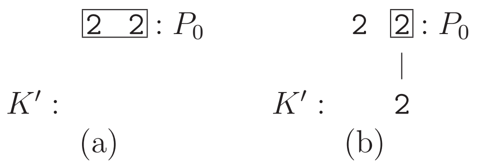

Figure 2: The first increment of the pointer P_0.

Figure 2 illustrates the first increment of the pointer P_0 in the algorithm. (a) The initiation of P_0. The framed symbols 22 are the contents of the pointer P_0. (b) The pointer P_0 holds two symbols so the first of them is discarded and the second is returned, which is the third symbol in K.

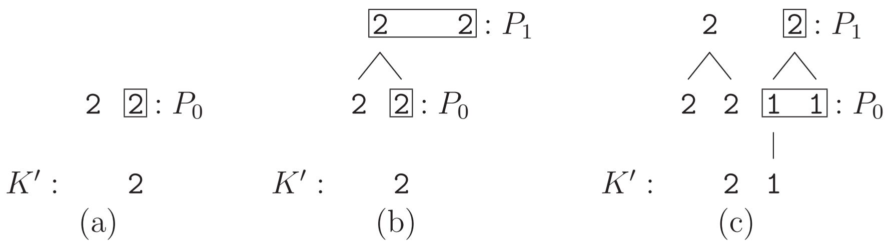

Figure 3: The second increment of the pointer P_0.

Figure 3 illustrates the second increment of the pointer P_0 in the algorithm. (a) The tree structure after the first increment of P_0. (b) To continue our traverse we must generate the next leaf. This is done by looking at the ancestor of the part of the run held by P_0. As this ancestor does not exist we have to generate it, that is we set $P_1 = 22$. (c) The first symbol of P_1 already has children (that is, it generated the initial run held by P_0). Therefore we step to the second symbol of P_1. The new run to assign to P_0 (that is, the new leaf we traverse) is then 11, since the current symbol in P_1 is 2 and P_0 currently holds a 2. Thus the symbol to return is the first in P_0, a 1.

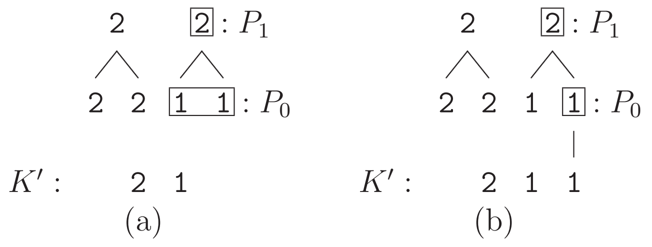

Figure 4: The third increment of the pointer P_0.

Figure 4 illustrates the third increment of the pointer P_0 in the algorithm. (a) The tree structure after the second increment of P_0. (b) The pointer P_0 holds two symbols so the first of them is discarded and the second is returned, which is the fifth symbol in K.

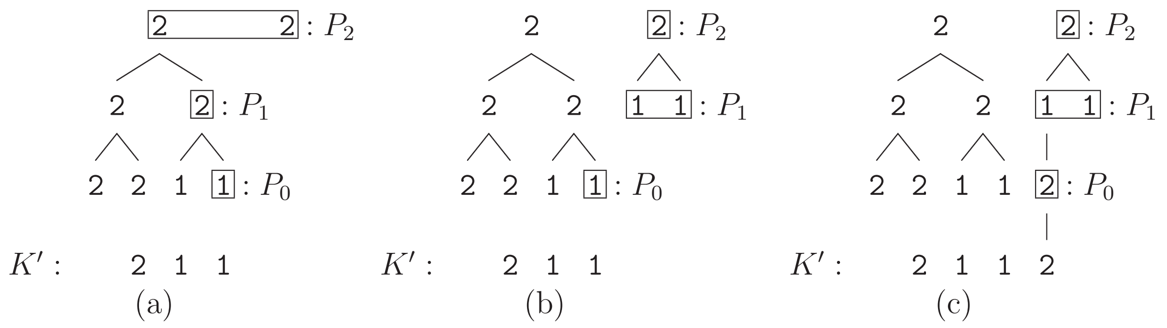
$\boxed{2\ \ \ 2}$: P_2
2 $\boxed{2}$: P_1
2 2 1 $\boxed{1}$: P_0
K' : 2 1 1
(a)
2 $\boxed{2}$: P_2
2 2 $\boxed{1\ 1}$: P_1
2 2 1 $\boxed{1}$: P_0
K' : 2 1 1
(b)
2 $\boxed{2}$: P_2
2 2 $\boxed{1\ 1}$: P_1
2 2 1 1 $\boxed{2}$: P_0
K' : 2 1 1 2
(c)

Figure 5: The fourth increment of the pointer P_0.

Figure 5 illustrates the fourth increment of the pointer P_0 in the algorithm. (a) The status of the pointers after the third increment of P_0. (b) We have used the symbols held by the pointers P_0 and P_1, but only the first in the run held by P_2. To generate the new contents of P_1, we discard the first symbol of P_2 and use the second one to create the new run in P_1, which is 11, since $P_2 = 2$ and P_1 previously held a 2. (c) The new run length of the run to assign to P_0 is now 1 since the first symbol in P_1 is a 1, and the symbol to write is 2 since P_0 previously held a 2, which is also the symbol which returned as the sixth symbol in K.

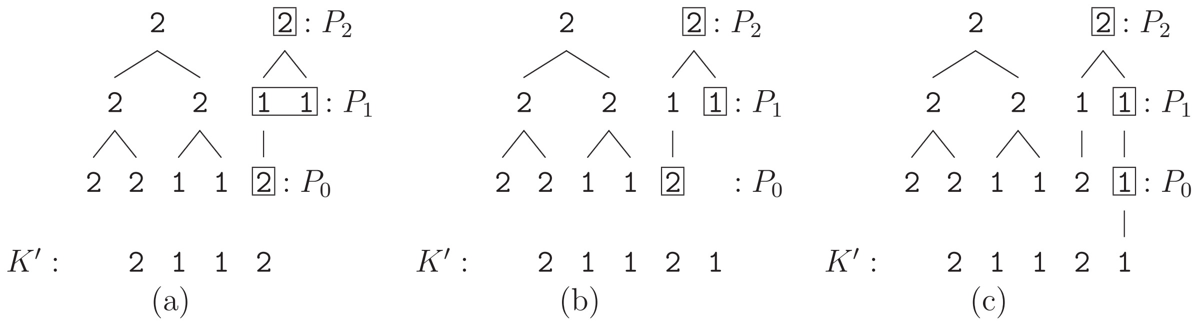

Figure 6: The fifth increment of the pointer P_0.

Figure 6 illustrates the fifth increment of the pointer P_0 in the algorithm. (a) The status of the pointers after the fourth increment of P_0. (b) Since we already used the first symbol held by P_1, we step to the second one. (c) The new run to write in P_0 is now 1, since $P_1 = 1$ and P_0 previously held the symbol 2, which is also the next symbol in K.

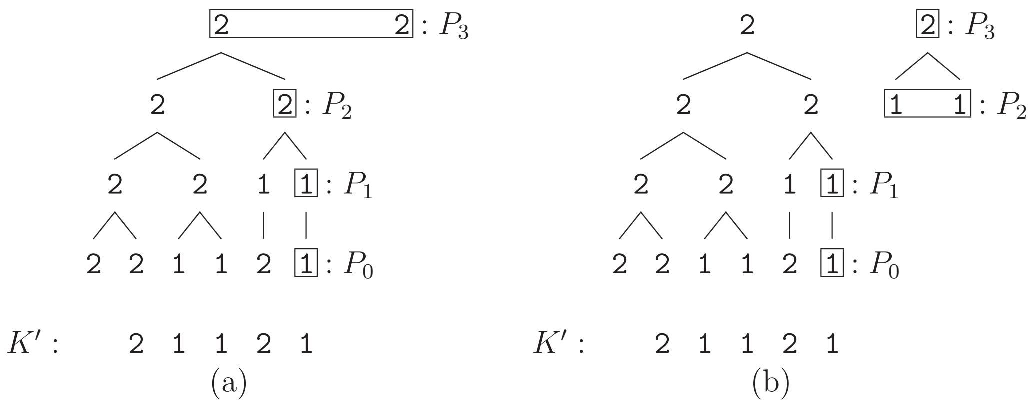

Figure 7: The first part of the sixth increment of the pointer P_0.

Figure 7 illustrates the first part of the sixth increment of the pointer P_0 in the algorithm. (a) To increase P_0 we have to look at the ancestors of the run held by P_0. We see that we have used all symbols in all of the ancestors, therefore we have to initiate the new pointer $P_3 = 22$. (b) We have already used the first symbol held by P_3 and therefore we step to its second symbol. The new run to assign to P_2 is now 11 since $P_3 = 2$ and $P_2 = 22$.

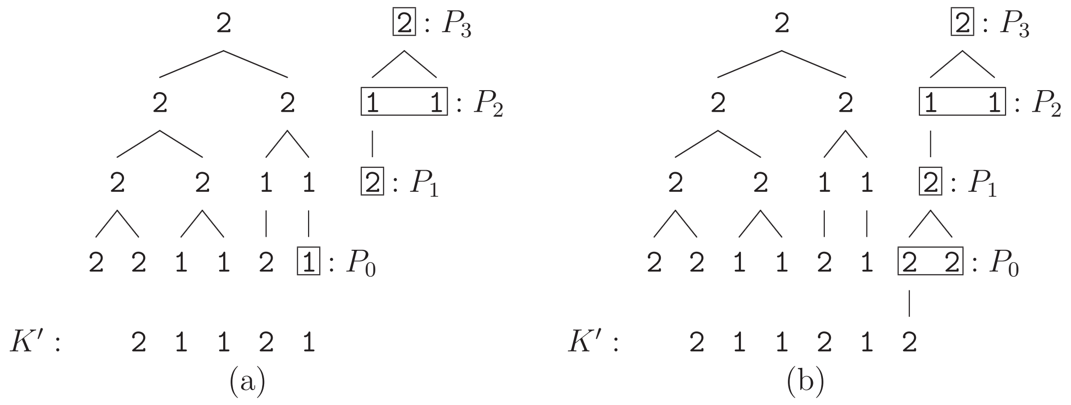

Figure 8: The second part of the sixth increment of the pointer P_0.

Figure 8 illustrates the second part of the sixth increment of the pointer P_0 in the algorithm. (a) We have not yet used any of the symbols held by P_2 and therefore the current one is the first. Then the new symbol in P_1 is 2 since the current symbol in P_2 is 1 and $P_1 = 1$. (b) The new run to assign to P_0 is now 22 since the first symbol in P_1 is 2 and $P_0 = 1$, and therefore we return a 2 as the next symbol in K. $\square$

Note that the algorithm does not need to keep track of the tree structure that it steps through. The algorithm only keeps track of the current contents of the pointers P_k and how many of each symbol we have got in return from each increment of the pointer P_0.

3 Run Time Analysis of the Algorithm

Let t_n be the number of 2s in K up to and including position n. That is,

$$t_n = |\{i : K_i = 1, 1 \leq i \leq n\}|.$$

Recall that we have already similarly defined o_n as the number of ones. By considering the subwords of K of length five with the maximal and minimal possible letter distributions, that is the words

$$\texttt{11211} \quad \text{and} \quad \texttt{22122},$$

we see that we have the bounds

$$\frac{1}{4} \leq \frac{o_n}{t_n} \leq 4 \qquad (2)$$

for $n \geq 2$. For the analysis, let $P(n)$ be the number of pointers used by Algorithm 2 to calculate o_n.

Proposition 4. *The amount of space used by Algorithm 2 to find o_n is logarithmic in n. That is, $P(n) = O(\log n)$.*

Proof. Let $w_0 = \texttt{122}$ and $w_1 = \texttt{12211}$ and similarly let the letters in w_k define the runs in w_{k+1}. Then w_k is a prefix of the sequence K for all $k \geq 0$. (The collection of the words w_k is known as the Kolakoski fan.) By the frequency bound (2) it follows that

$$\frac{6}{5} \leq \frac{|w_{k+1}|}{|w_k|} \leq \frac{9}{5}$$

whenever $k \geq 1$ and where $|\cdot|$ denotes the length of a word.

This implies that if pointer P_0 holds the symbol at position n in K' then the pointer P_1 is at most at position $\frac{5}{6}n$ and at least at position $\frac{5}{9}n$ in the Kolakoski sequence. Accordingly, pointer P_2 is at most at position $(\frac{5}{6})^2 n$ and at least at position $(\frac{5}{9})^2 n$, and so on at higher levels in the tree. We only have to continue upwards in the tree until the pointer P_k holds the first position in K', that is $\left[(\frac{5}{6})^k n\right] = 1$. This gives that the bound of the number of pointers needed to find o_n is given by

$$P(n) \leq \left\lceil \frac{1}{\log \frac{6}{5}} \log n \right\rceil = O(\log n),$$

which completes the proof. $\square$

If Conjecture 1 were shown to be true, it would follow that the number of pointers needed to find o_n is $P(n) \approx \frac{1}{\log \frac{3}{2}} \log n$.

Proposition 5. *Algorithm 2 runs in (amortized) linear time. That is, to find o_n we have to do an amount of work of order $O(n)$.*

Proof. Let us consider the maximal amount of work we have to do to make n increments of the pointer P_0 (to generate n symbols).

Let $p_k(n)$ be the number of times we change the contents of pointer P_k under these n increments. Then the sum of the p_ks will give us the total amount of work we have to do. It is clear that $p_0(n) = n$, since we change P_0 at each increment. The other pointers do not change every time; for $k \geq 1$ we make a change to P_k only when P_{k-1} consists of a single symbol.

Let $a_k(n)$ be the number of times the pointer P_k holds a single symbol under n increments of P_0. Similarly let $b_k(n)$ be the number of times that P_k holds two symbols under the n increments of P_0. From the algorithm we see that to find the maximal amount of work, we have to look for the maximal number of single-symbol pointer contents, since this is what forces us to go recursively higher in the tree. For the pointer P_0 it follows from (2) that we have the bounds

$$\frac{1}{4} \leq \frac{a_0(n)}{b_0(n)} \leq 4$$

for $n \geq 1$. For pointers higher up, we have that the number of times P_k holds a single symbol is at most four times the number of times it holds two symbols plus the number of times it holds two symbols, since in the latter case P_k will hold a single symbol in the next step of the algorithm. This gives

$$a_k(n) \leq 4b_k(n) + b_k(n) = 5b_n(k)$$

Therefore, our upper bound on the number of times a pointer holds a single symbol gives the bound on the amount of work we have to do with a pointer P_k compared to the amount of work for the pointer holding the children of P_k. This is

$$p_{k+1}(n) \leq \frac{5}{6}\, p_k(n) \tag{3}$$

for $k \geq 1$. The total amount of work we now have to do to increment the pointer P_0 n times is therefore bounded by the initial amount of work plus the convergent geometric series obtained from (3) We have

$$P(n) = \sum_{i=0}^{\infty} p_i(n) \leq n + c \log n + n \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i \leq (7 + C)n, \tag{4}$$

where $c \log n$ is the initial amount of work for each pointer before we can apply our estimates above. □

4 Generalized Kolakoski Sequences

In this this section we remark that our algorithm is also applicable to a general Kolakoski sequence. By a generalized Kolakoski sequence we mean a sequence that is defined as its symbols' run length, as for the classical Kolakoski sequence, but the symbols may be taken from any alphabet $\{r, s\}$, where r and s are natural numbers, as discussed in [4]. We denote a generalized Kolakoski sequence over r and s with $K(r, s)$ and shall assume that $K(r, s)$ starts with the symbol r. The classical Kolakoski sequence is then $K = K(\mathtt{1}, \mathtt{2})$.

It is known that if $r + s$ is an even number, then the letter frequency in $K(r, s)$ can be calculated; see [1, 11, 12, 13]. When $r + s$ is odd, the existence and the value of the letter frequencies are still unknown, but are believed to exist and equal $\frac{1}{2}$.

Our algorithm easily adopts to count the letters in a generalized Kolakoski sequence; we may only have to change the initiation of new pointers. By applying the same idea as in the proof of Proposition 4 we see that the algorithm in this case with a generalized Kolakoski sequence uses fewer pointers than for the classical Kolakoski sequence, and therefore the space requirement must again be at most logarithmic.

Similarly, by looking at the proof of Proposition 5 we see that the number of times we use a pointer for a general Kolakoski sequence before having to consider its ancestor is longer than for the classical Kolakoski sequence. Therefore the bounding factor for the quoted amount of work between two consecutive levels (3) must be smaller than the $\frac{5}{6}$ given for the classical Kolakoski sequence. This gives then, by summing up as in (4), that the total amount of work for the generalized Kolakoski sequence is also linear in n.

5 Calculations

In Table 1 we present a short output from an implementation in Java of our Algorithm 2 for calculating the number of `1`s in the classical Kolakoski sequence. The program was run on a standard PC. In Table 2 we present results of a calculation of the number of 2s in the generalized Kolakoski sequence $K(2,3)$, the sequence A071820 in the On-Line Encyclopedia of Integer Sequences [14].

We denote for the classical Kolakoski sequence the maximal deviation of the proportion of 1s from $\frac{1}{2}$ in a logarithmic decade by

$$D(n) = \max_{\frac{1}{10}n<i\leq n} \left| \frac{1}{2} - \frac{o_i}{i} \right|,$$

where o_i is the number of 1s up to position i. We can similarly define the deviation for the generalized Kolakoski sequence $K(2,3)$.

n	Number of 1s	$P(n)$	$D(n)$
1	1		
10^1	5	4	$1.667 \cdot 10^{-1}$
10^2	49	10	$8.333 \cdot 10^{-2}$
10^3	502	16	$1.351 \cdot 10^{-2}$
10^4	4 996	22	$3.588 \cdot 10^{-3}$
10^5	49 972	27	$5.481 \cdot 10^{-4}$
10^6	499 986	33	$2.800 \cdot 10^{-4}$
10^7	5 000 046	39	$3.892 \cdot 10^{-5}$
10^8	50 000 675	44	$2.054 \cdot 10^{-5}$
10^9	500 001 223	50	$8.586 \cdot 10^{-6}$
10^{10}	4 999 997 671	56	$2.152 \cdot 10^{-6}$
10^{11}	50 000 001 587	61	$4.453 \cdot 10^{-7}$
10^{12}	500 000 050 701	67	$2.140 \cdot 10^{-7}$
10^{13}	5 000 000 008 159	73	$6.774 \cdot 10^{-8}$

Table 1: The output of the calculation of the number of 1s in the classical Kolakoski sequence. Our result here exceeds the calculations made by Steinsky up to $n = 3 \cdot 10^8$, and Monteil up to $n = 10^{11}$, as mentioned earlier. The column with the number of 1s is the sequence A195206 in the On-Line Encyclopedia of Integer Sequences [14].

6 Acknowledgement

The author wishes to thank M. Baake, P. Bugarin, V. Terauds and P. Zeiner at Bielefeld University, Germany, for our discussions of the problem and for reading drafts of the manuscript. This work was supported by the German Research Council (DFG), via CRC 701.

n	Number of 2s	$P(n)$	$D(n)$
1	1		
10^1	5	3	$2.143 \cdot 10^{-1}$
10^2	51	6	$8.333 \cdot 10^{-2}$
10^3	502	9	$2.459 \cdot 10^{-2}$
10^4	4 995	11	$3.318 \cdot 10^{-3}$
10^5	49 999	14	$6.353 \cdot 10^{-4}$
10^6	499 980	16	$8.448 \cdot 10^{-5}$
10^7	4 999 995	19	$2.464 \cdot 10^{-5}$
10^8	50 000 202	21	$7.936 \cdot 10^{-6}$
10^9	499 999 731	24	$3.279 \cdot 10^{-6}$
10^{10}	5 000 005 565	26	$8.382 \cdot 10^{-7}$
10^{11}	50 000 013 114	29	$5.606 \cdot 10^{-7}$
10^{12}	499 999 997 503	31	$1.430 \cdot 10^{-7}$
10^{13}	4 999 999 971 938	34	$3.744 \cdot 10^{-8}$

Table 2: The output of the calculation of the number of 2s in the generalized Kolakoski sequence $K(2,3)$. The column with the number of 2s is the sequence A195211 in the On-Line Encyclopedia of Integer Sequences [14].

References

[1] M. Baake and B. Sing, Kolakoski(3; 1) is a (deformed) model set, *Canad. Math. Bull.* **47** (2004), 168–190.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 2001.

[3] F. M. Dekking, Regularity and irregularity of sequences generated by automata, Exposé no. 9, *Sém. Th. Nombres Bordeaux*, 1979–1980, pp. 9.01–9.10.

[4] F. M. Dekking, On the structure of self generating sequences, Exposé no. 31, *Sém. Th. Nombres Bordeaux*, 1980–1981, pp. 31.01–31.06.

[5] F. M. Dekking, What is the long range order in the Kolakoski sequence?, *The Mathematics of Long-Range Aperiodic Order*, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci. **489**, Kluwer Acad. Publ., (1997), pp. 115–125.

[6] C. Kimberling, Integer sequences and arrays, `http://faculty.evansville.edu/ck6/integer/index.html`.

[7] W. Kolakoski, Problem 5304: self generating runs, *Amer. Math. Monthly* **72** (1965), 674.

[8] W. Kolakoski, Problem 5304, *Amer. Math. Monthly* **73** (1966), 681–682.

[9] T. Monteil, Brute force Kolakoski, `http://www.lirmm.fr/~monteil/blog/BruteForceKolakoski/`.

[10] R. Oldenburger, Exponent trajectories in dynamical systems, *Trans. Amer. Math. Soc.* **46** (1939), 453–466.

[11] B. Sing, *Spektrale Eigenschaften der Kolakoski-Sequenzen*, Diploma thesis, Universität Tübingen, 2002.

[12] B. Sing, Kolakoski-($2m$; $2n$) are limit-periodic model sets, *J. Math. Phys.* **44** (2003), 899–912.

[13] B. Sing, More Kolakoski sequences, *INTEGERS*, **11B** (2011), Paper A14.

[14] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences. Published electronically at `http://oeis.org/`.

[15] B. Steinsky, A recursive formula for the Kolakoski sequence A000002, *J. Integer Sequences*, **9** (2006), Article 06.3.7.

2010 *Mathematics Subject Classification*: Primary 68R15; Secondary 68Q25.
Keywords: combinatorics on words, analysis of algorithms, linear space, Kolakoski sequence.

(Concerned with sequences A000002, A071820, A195206, and A195211.)

Received October 12 2011; revised versions received March 9 2012; June 25 2012. Published in *Journal of Integer Sequences*, June 26 2012.

Return to Journal of Integer Sequences home page.