The Kolakoski Sequence and Some Fast Algorithms, Part 2

Richard P. Brent

Australian National University
and University of Newcastle

1 October 2017

Joint work with Judy-anne Osborn

Introduction

This is part 2 of a talk describing joint work with Judy-anne Osborn. The *Kolakoski sequence* $K = (k_j)_{j\geq 1} \in \{1,2\}^{\mathbb{N}}$ was defined in her talk, and is OEIS sequence A000002.

The first 27 term of K are

$$1,2,2,1,1,2,1,2,2,1,2,2,1,1,2,1,1,2,2,1,2,1,1,2,1,2,2,\ldots$$

K has the property that its run-length encoding is itself (i.e. K):

$$\underbrace{1,}_{1}\underbrace{2,\,2,}_{2}\underbrace{1,\,1,}_{2}\underbrace{2,}_{1}\underbrace{1,}_{1}\underbrace{2,\,2,}_{2}\underbrace{1,}_{1}\underbrace{2,\,2,}_{2}\underbrace{1,\,1,}_{2}\underbrace{2,}_{1}\underbrace{1,\,1,}_{2}\underbrace{2,\,2,}_{2}\ldots,$$

and indeed this property defines K uniquely, except that we could omit the first term to obtain a sequence

$$2,2,1,1,2,1,2,2,1,2,2,1,1,2,1,1,2,2,1,2,1,1,2,1,2,2,\ldots$$

with the same property.

The alphabet and the discrepancy function

It may be convenient to change the alphabet $\Sigma = \{1, 2\}$. In a computer implementation, we can map $(1, 2) \mapsto (0, 1)$, i.e. $k \mapsto k - 1$, so that 64 consecutive terms such as $k_1 \ldots k_{64}$ can be represented in a single 64-bit computer word.

We are interested in the (open) question of whether K is equidistributed. Thus, it may be convenient to map $(1, 2) \mapsto (-1, 1)$, i.e. $k \mapsto (-1)^k = 2k - 3$.

We define the *Kolakoski discrepancy function* $\delta(n)$ by

$$\delta(n) := \sum_{j=1}^{n} (-1)^{k_j}.$$

Thus, $\delta(n)$ is the excess of twos over ones in the first n terms of K, and K is equidistributed if and only if

$$\delta(n) = o(n), \quad \text{i.e.} \quad \lim_{n\to\infty} \delta(n)/n = 0.$$

Algorithms for the discrepancy function

The obvious algorithm to compute $\delta(n)$ takes time and space linear in n. Nilsson (2012) gave an algorithm for computing $k_1 \ldots k_n$, and hence $\delta(n)$, in time $O(n)$ and space $O(\log n)$. We describe several algorithms that compute $\delta(n)$ faster, using a space-time tradeoff. The algorithms use ideas of Nilsson and Rao. It is conjectured that the fastest algorithm runs in time and space $O(n^\alpha)$, where $\alpha = \log(2)/\log(3) \approx 0.631$.

A linear time and space algorithm

It is straightforward to generate $k_1 k_2 \ldots k_n$ in linear fashion, using the fact that k is a fixed point of the "run-length decoding" function $T_1(\sigma_1 \sigma_2 \ldots) = 1^{\sigma_1} 2^{\sigma_2} 1^{\sigma_3} 2^{\sigma_4} \ldots$

Denote the complement of $x \in \Sigma$ by x'. Thus $1' = 2,\ 2' = 1$, but we can represent 2 by 0 if desired (to save space).

We use an array A of $n+1$ bits $A_1 \ldots A_{n+1}$, and indices i, j.

Algorithm 1:

$(A_1, A_2, i, j) \leftarrow (1, 1', 2, 2)$;
while $i \leq n$ do
  if $A_j = 1$ then $(A_i, i, j) \leftarrow (A'_{i-1}, i+1, j+1)$
    else $(A_i, A_{i+1}, i, j) \leftarrow (A'_{i-1}, A'_{i-1}, i+2, j+1)$.

Now $A_1 \ldots A_n = k_1 \ldots k_n$.

The algorithm uses time $O(n)$ and space $O(n)$.

Illustration of Algorithm 1

The algorithm can be illustrated by the following table. As the indices i and j increase, the third and fourth columns both represent initial segments of the Kolakoski sequence.

i increases by 2 when $A_j = 2$ (skipped values in parentheses).

i	j	A_i	A_j
1	1	1	1
2	2	2	2
(3)		2	
4	3	1	2
(5)		1	
6	4	2	1
7	5	1	1
8	6	2	2
(9)		2	
10	7	1	1

Saving space via recursion

Nilsson (2012) improved Algorithm 1 by reducing the space required to generate k_n (or $\delta(n)$) from $O(n)$ to $O(\log n)$.
The time required is still $O(n)$.

The idea is to generate the Kolakoski sequence by a recursive procedure, where the reference to A_j in Algorithm 1 is replaced by a recursive call to the same procedure, unless $j \leq 2$.

This can be illustrated by the table on the next slide.

Recursive generation of the Kolakoski sequence

Each column $A, B, C, \ldots$ gives the Kolakoski sequence. Column B generates column A, column C generates column B, etc. The depth of recursion increases at each blue row.

index	A	B	C	D	E	F	G	...
1	1	1	1	1	1	1	1	...
2	2	2	2	2	2	2	2	...
3	2							
4	1	2						
5	1							
6	2	1	2					
7	1	1						
8	2	2	1	2				
9	2							
10	1	1	1					
11	2	2	2	1	2			
12	2							
13	1	2						
14	1							
15	2	1	1	1				
16	1	2	2	2	1	2		
17	1							
18	2	2						
19	2							
20	1	1	2					
21	2	1						
22	1	2	1	1	1			
23	1							
24	2	1	2	2	2	1	2	
...								

A sublinear algorithm (Algorithm 2)

Nilsson's algorithm takes time of order n to generate k_n or $\delta(n)$. This is the best that we can do if all of $k_1, \ldots, k_n$ are required.

However, it is possible to generate a single k_n or $\delta(n)$ value (or a sparse sequence of such values) in time $\ll n$.

The first step is to convert Nilsson's recursive algorithm into an iterative algorithm that generates the table on the previous slide row by row. This gives a non-recursive algorithm with essentially the same time and space requirements as Nilsson's algorithm.

Next, we observe that a row in the table determines some of the following rows, as illustrated on the following slide.

One row determines several following rows

index	A	B	C	D	E	F
1–10	···					
11	2	2	2	1	2	
12	2					
13	1	2				
14	1					
15	2	1	1	1		
16	1	2	2	2	1	2
···						

Row 11 determines rows 12–15 as the number of columns (5) in row 11 is at least the number of columns in rows 12–15.

However, row 11 does not determine row 16, since row 16 has 6 columns, and the column labelled "F" could have an entry 1 or 2.

Another example

index	A	B	C	D
1–3	···			
4	1	2		
5	1			
6	2	1	2	
7-12	···			
13	1	2		
14	1			
15	2	1	1	1
···				

Row 4 determines row 5, but not row 6. Row 13 is identical to row 4, so determines row 14, but not row 15. Note that row 6 and row 15 differ in the entries marked in red.

Using table lookup

For each d, $1 \le d \le d_{max}$, where d_{max} is determined by the amount of random-access memory available, we can construct a table of 2^d entries indexed by d-bit binary integers (the *keys*). Each such integer m corresponds to a row (say row r) with d symbols from Σ, which we can map to $\{0, 1\}$.

The table tells us how far we can skip ahead, say ℓ rows, and gives sufficient information to construct row $r + \ell$ from row r. It also contains $\sum_{r<j\le r+\ell} (-1)^{k_j}$, so there is enough information to determine $\delta(n)$ when we reach row n.

In fact, it is sufficient to consider rows ending in 2, since rows ending in 1 have $\ell = 0$.

If a row has length $> d_{max}$, or if the table lookup would skip over the desired index n, we revert to the "slow but sure" iterative version of Nilsson's algorithm.

Space-time tradeoff

The mean value of ℓ over all 2^d d-bit keys is $(3/2)^d$. On the assumption that each possible key is equally likely to occur, we expect a speedup of order $(3/2)^{d_{max}}/d_{max}$, This is confirmed by numerical experiments.

Note: the keys are *not* substrings of the Kolakoski sequence. The latter can not include strings such as 111 or 222, so the number of length-d substrings that occur in the Kolakoski sequence is (much) less than 2^d. (It is polynomial in d.)

We have to initialise the lookup tables. This takes time $O(3^{d_{max}})$ if done in a lazy manner, but $O(2^{d_{max}})$ if done recursively, starting from small d.

Thus, the total time to compute $\delta(n)$ is [conjectured to be]

$$O((2/3)^{d_{max}} d_{max}\, n + 2^{d_{max}}).$$

Choosing d_{max}

Choosing $d_{max} = \lfloor \log(n)/\log(3) \rfloor$ gives time $O(n^\alpha \log n)$ and space $O(n^\alpha)$, where $\alpha = \log(2)/\log(3) \approx 0.631$.

For the values of n that we are interested in, say $n \approx 10^{19}$, this would give $d_{max} \approx 39$. Our program uses about $24 \times 2^{d_{max}}$ bytes, so on a machine with say 128 GB of memory available we are limited to $d_{max} \leq 32$. Thus, in practice, we are limited by the availability of memory. Choosing the largest possible $d_{max} \leq \log(n)/\log(3)$, the runtime is $O((2/3)^{d_{max}} d_{max}\, n)$.

One way to interpret this analysis is as follows. Each time that we double the amount of memory available for lookup tables, we obtain a speedup by a factor of about $3/2$. Since $(3/2)^{30} \approx 191,000$, the speedup is significant. $d_{max} = 30$ is optimal for $n \approx 3^{30} \approx 2 \times 10^{14}$, but sub-optimal for larger n.

Computing other quantities

It is possible to compute other functions related to $\delta(n)$ with only a constant factor slowdown by extending Algorithm 2.

For example, we can compute

$$\max_{a \le n \le b} \delta(n) \quad \text{and} \quad \min_{a \le n \le b} \delta(n), \text{ and hence } \max_{a \le n \le b} |\delta(n)|,$$

at the expense of a factor $5/3$ in memory for lookup tables, and a small constant factor increase in running time.

Our computer implementations compute $\max_{1 \le j \le n} \delta(j)$, $\min_{1 \le j \le n} \delta(j)$, and the least index j where each "significant" max/min occurs.

Another fast algorithm

We outline another algorithm, Algorithm 3, which (conjecturally) has better complexity than Algorithm 2, although only by a logarithmic factor. It is based on an idea of Michaël Rao (2012).

Before describing Algorithm 3, we need to consider finite state transducers. These are theoretical computing machines whose power lies between that of finite automata and Turing machines.

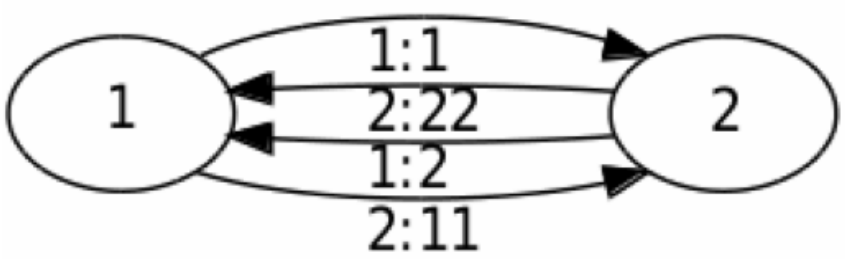
Finite-state transducers

A *finite-state transducer* (FST) is a finite-state machine with an input tape and an output tape.

For example, the Kolakoski sequence K is a fixed point of the FST $\mathcal{K}$ defined by the table

initial state	input	output	final state
1	1	1	2
1	2	11	2
2	1	2	1
2	2	22	1

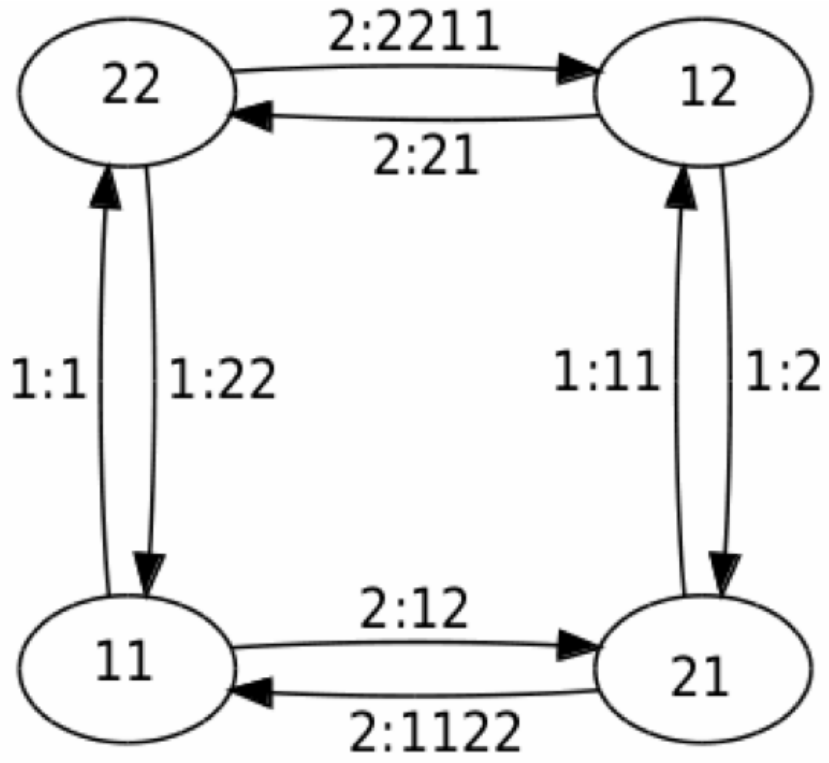
or equivalently by the labelled, directed graph



Composition of FSTs

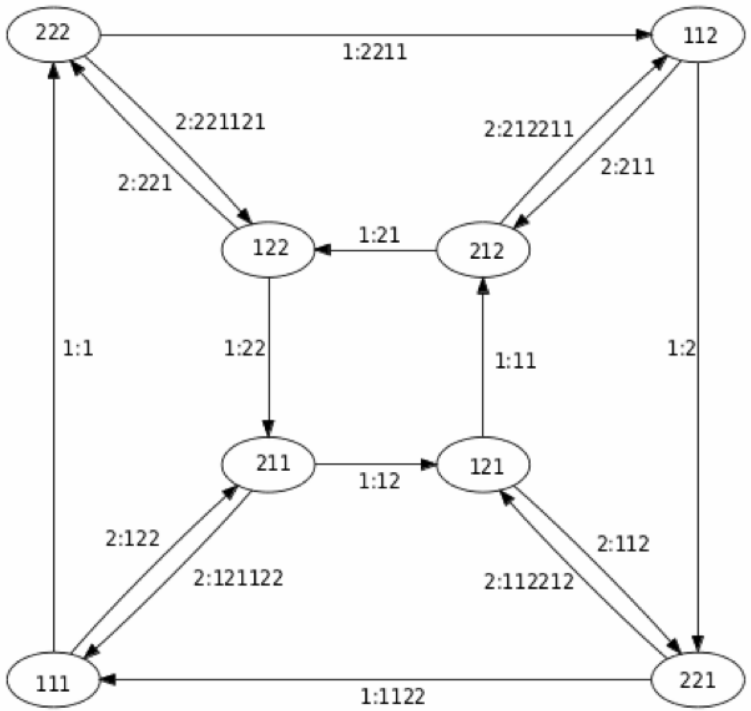
By connecting the output of an FST $\mathcal{K}$ to the input of an FST $\mathcal{L}$ we get an FST that is denoted by $\mathcal{K} \circ \mathcal{L}$. (Note the order!) The set of states of $\mathcal{K} \circ \mathcal{L}$ is the cartesian product of the sets of states of $\mathcal{K}$ and $\mathcal{L}$.

For example, $\mathcal{K}^2 := \mathcal{K} \circ \mathcal{K}$ can be represented by the graph



Composition of FSTs cont.

$\mathcal{K}^3 := \mathcal{K}^2 \circ \mathcal{K}$ can be represented by



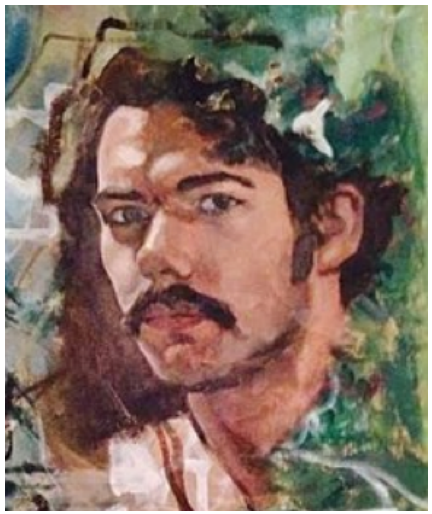
The directed graph representing $\mathcal{K}^d$ has 2^d vertices. Each vertex has two out-edges and two in-edges, making a total of 2^{d+1} directed edges. The edges are labelled by inputs $\in \{1, 2\}$, and outputs of variable length (strings over $\{1, 2\}$). The total length of the outputs is $2 \cdot 3^d$, so the average length of an output is $(3/2)^d$.

Remark

Rao's graphs are related to, but not the same as, the graphs of Chvátal, described in Part 1. In fact, a Chvátal graph of size 2^{d+1} can be mapped to a Rao graph of size 2^d.

Acknowledgement: the graphs for $\mathcal{K}$, $\mathcal{K}^2$, $\mathcal{K}^3$ are from Michaël Rao's website `http://www.arthy.org/kola/kola.php`.

Rufus Oldenburger



William Kolakoski

Michaël Rao

The "expander" interpretation of $\mathcal{K}$

We can think of the transducer $\mathcal{K}$ as a **black box** with 1-bit state (memory) that takes an initial segment of the Kolakoski sequence K as input and outputs a (generally longer) initial segment of K.

For example, if we assume that $\mathcal{K}$ has initial state 1 and we input the first 5 symbols 12211 of K, we obtain the first 7 symbols 1221121. This can be seen from the picture

$$\underbrace{1}_{1}\ \underbrace{2}_{22}\ \underbrace{2}_{11}\ \underbrace{1}_{2}\ \underbrace{1}_{1}\ \cdots$$

I find it easier to write K from *right to left*, since I store bits in computer words least-significant bit first, and these bits are traditionally regarded as at the right end of the word. Thus the action of $\mathcal{K}$ can be pictured as

$$\cdots 11221 \to \mathcal{K} \to \cdots 1211221$$

This convention is natural if inputs are on the left and outputs on the right. If you don't like it, put the inputs on the right!

Interpretation of $\mathcal{K}^d$

Remembering the right-to-left convention for inputs and outputs, we have

$$\cdots 11221 \to \mathcal{K} \to \cdots 1211221$$

and

$$\cdots 1211221 \to \mathcal{K} \to \cdots 1221211221$$

which may be abbreviated as

$$\cdots 11221 \to \mathcal{K}^2 \to \cdots 1221211221$$

Similarly,

$$\cdots 11221 \to \mathcal{K}^3 \to \cdots 211221221211221$$

We can think of $\mathcal{K}^d$ as a black box with d-bit state. The "amplification factor" is (conjecturally) $(3/2)^d$.

The idea of Algorithm 3

The idea of Algorithm 3 is to choose some exponent d (say $d = 32$) and precompute a representation of $\mathcal{K}^d$. Then we can input the Kolakoski sequence (generated for example by Nilsson's algorithm) and use $\mathcal{K}^d$ to "accelerate" the input generator by a factor of (hopefully) about $(3/2)^d$.

The problem with this idea is that we still have to generate the entire output sequence. We can not hope to generate $k_1, \ldots, k_n$ in time of order less than n.

An important observation is that, in order to compute the Kolakoski discrepancy function δ at a sufficiently sparse set of points, we do not need to generate *all* of an initial segment $k_1, \ldots, k_n$.

In the computed representation of $\mathcal{K}^d$, we can abbreviate the output string $s_1 \cdots s_\ell$ by (ℓ, η), where ℓ is the length and $\eta := \sum_{1 \le j \le \ell} (-1)^{s_j}$ is the contribution to the discrepancy function arising from $s_1 \cdots s_\ell$.

Algorithm 3 (outline)

To represent $\mathcal{K}^d$, we could use lookup tables of size 2^{d+1}, indexed by the state (2^d possibilities) and input symbol (2 possibilities). The table entries contain ℓ, η, and the resulting state (d bits). If $d \leq 32$ it is sufficient to allow 20 bits for ℓ and 11 bits for η, so the entries fit into a 64-bit word. This makes a total of 16×2^d bytes.

In fact, we can save a factor of two in storage by taking advantage of a symmetry which is evident by inspection of the graph of $\mathcal{K}^3$. If the vertex label (i.e. state) is $\sigma_1 \cdots \sigma_d$, we can assume that $\sigma_d = 1$, since the vertices with $\sigma_d = 2$ and their corresponding edges are easily reconstructed (the outputs are flipped $1 \leftrightarrow 2$). Thus, we need only 8×2^d bytes for tables. For example, if $d = 32$, this is 32GB (a factor of 4 less than Algorithm 2 for $d_{max} = 32$).

If $32 < d \leq 36$ we can use 23 bits for ℓ and 12 bits for η, so 9×2^d bytes for tables. For example, if $d = 34$, this is 144GB, which will fit on a machine with 256GB (the largest available to us).

Algorithm 3 (some details)

As described so far, Algorithm 3 would output $\delta(n)$ at irregularly spaced values of the index n, separated on average by about $(3/2)^d$ (≈ 431440 if $d = 32$). Since we aim to compute $\delta(n)$ for n of order 10^{20}, we would prefer to output $\delta(n)$ at equally spaced values of n separated by say 10^{16}.

To solve this problem, if the current interval covered by the table lookup process is $[n_0, \ldots, n_0 + \eta]$ and a value for which we want output falls in this interval, then we revert to the "slow but sure" process of generating $k_{n_0}, k_{n_0+1}, \ldots, k_{n_0+\eta}$. Provided this exceptional case occurs sufficiently rarely, the amortised complexity (or average speed) of the algorithm is hardly affected. Since $10^{16}/(3/2)^{32} \approx 2.3 \times 10^{10}$ is large, this is the case in practice.

Computing the max/min in Algorithm 3

As for Algorithm 2, we would like to compute $\max_{1 \le j \le n} \delta(j)$ and $\min_{1 \le j \le n} \delta(j)$ as well as $\delta(n)$ at a set of regularly spaced points. (Who knows what extreme behaviour could be hiding in an interval of length 10^{16} ?)

The lazy approach is to compute the max and min values once per table lookup. We also compute how much $\delta(n)$ can vary in the intervals given by a table lookup (577 for $d = 32$, and 905 for $d = 34$). Thus, our estimates of the max and min values will be in error by less than 1000, which is sufficient to estimate the asymptotic behaviour.

If we want the exact max and min values, we can simply use the "slow but sure" process whenever the current $\delta(n)$ is sufficiently close to the current max/min that a new max/min might lie in the current interval. This potentially reduces the average speed of the algorithm, but in practice we have found that it causes a slowdown of less than 10 percent on average (although it causes a large slowdown in rare cases). There is no significant storage penalty.

Complexity of various algorithms

The complexity analysis of Algorithm 3 is similar to that of Algorithm 2, except that the $\log n$ factor can be avoided.

To summarise:

Algorithm 1: $T = O(n)$, $S = O(n)$.

Nilsson: $T = O(n)$, $S = O(\log n)$.

Algorithm 2: $T = O(n^\alpha \log n)$, $S = O(n^\alpha)$,
or, for $S \ll n^\alpha$, $T = O(n \log S / S^\beta)$.

Algorithm 3: $T = O(n^\alpha)$, $S = O(n^\alpha)$,
or, for $S \ll n^\alpha$, $T = O(n / S^\beta)$.

Here $\alpha = \log 2 / \log 3 \approx 0.63$, $\beta = \log_2(3/2) = 1/\alpha - 1 \approx 0.58$.

The time bounds for Algorithms 2–3 are conjectured because they depend on unproved assumptions such as equidistribution.

Rao's algorithm

Michaël Rao (2012) uses composition of FSTs to compute a function equivalent to the Kolokoski discrepancy function $\delta(n)$, for various $n \leq 10^{18}$. In fact he computes $(n - \delta(n))/2$, which is the number of ones in the sequence $k_1, \ldots, k_n$. So far as we know, he does not compute the max/min values of $\delta(n)$.

Rao does not describe his approach in detail, but presumably it is similar to our Algorithm 3. Since he does not say how long his computation to 10^{18} took, we can not compare the efficiency of our algorithms/implementations.

Our implementation of Algorithm 3, running on a 3.47Ghz machine with $d = 34$ and using 144GB of memory, takes about 38 hours to compute $\delta(10^{18}) = 111,069,790$ (this is 2.6×10^{16} per hour).

We are now (as at 29 Sept 2017) up to 7.6×10^{19}, and hope to reach 10^{20} in about 7 weeks.

Computation of $\delta(n)$

The table gives some values of $\delta(n)$ and scaled values of $\delta(n)$ and $\Delta(n) := \max_{1 \le j \le n} |\delta(j)|$.

n	$\delta(n)$	$\delta(n)/n^{1/2}$	$\Delta(n)/n^{1/2}$
10^3	-4	-0.1265	0.1897
10^6	+28	+0.0280	0.0660
10^9	-2,446	-0.0773	0.1560
10^{12}	-101,402	-0.1014	0.1515
10^{15}	-1,954,842	-0.0618	0.1390
10^{18}	+111,069,790	+0.1111	0.1415
10^{19}	+548,593,100	+0.1735	0.1782
7×10^{19}	+1,177,987,702	+0.1408	0.1581

The $\delta(n)$ values are in agreement with those computed by Michaël Rao, up to his limit of 10^{18}. All results have been computed by at least two runs, and usually by two different programs (e.g. using Algorithms 2 and 3, or Algorithm 3 with different values of d).

Some extreme values of $\delta(n)$

Our computations show that

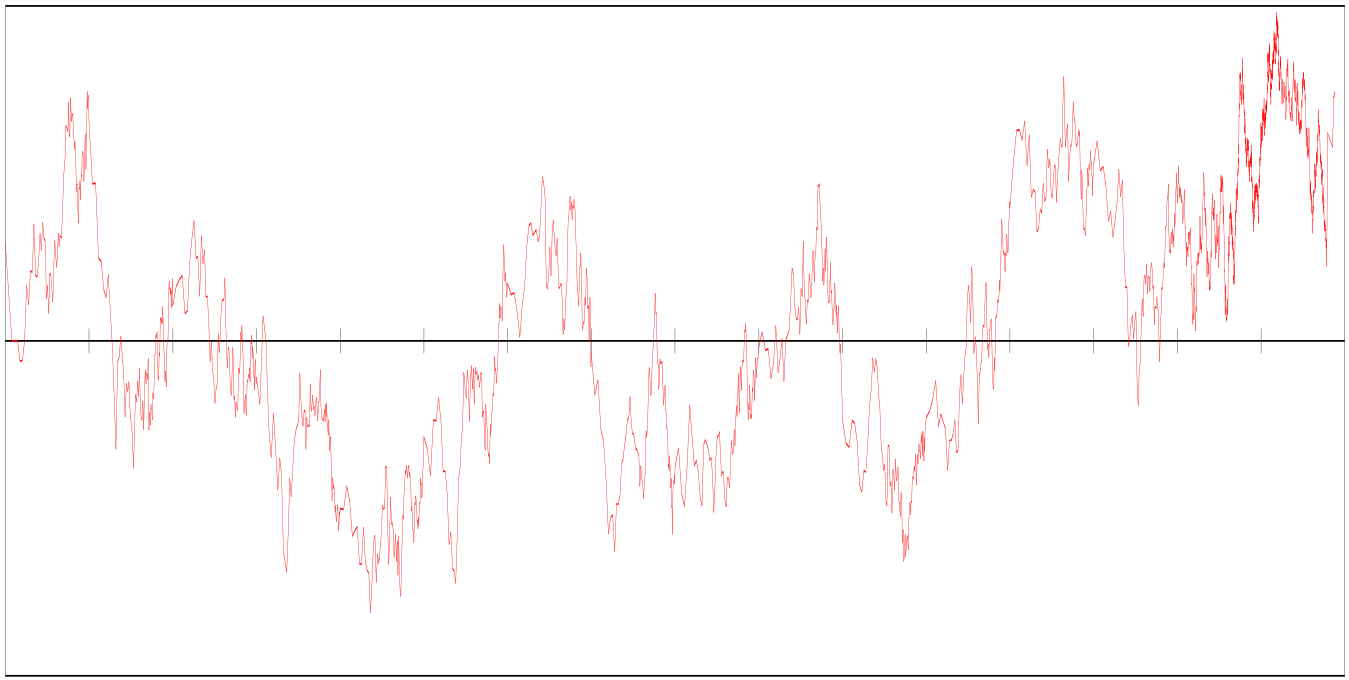
$$\forall n \in [1572, 7.6 \times 10^{19}] \;\; |\delta(n)| < 0.2663\, n^{1/2}.$$

The constant here is essentially the best possible because, for $n = 15,337,359,163,568,838,683 \approx 1.5 \times 10^{19}$, we have $\delta(n) = 1,042,818,353$, and $\delta(n)/n^{1/2} = 0.2662\ldots$

For $n \le 7 \times 10^{19}$, the maximum of $|\delta(n)|$ is $1,322,572,912$ at $n = 48,613,959,221,650,774,546 \approx 4.9 \times 10^{19}$, and $\delta(n)/n^{1/2} \approx 0.1897$.

A graph of the computational results to 7.6×10^{19}

The plot shows $\delta(n)/n^{1/2} \in [-0.27, 0.27]$ versus $\log_{10}(n) \in [4, 20]$.
It looks like a stationary process.



$\delta(n)/\sqrt{n}$ vs $\log_{10}(n)$, $10^4 \le n \le 7.617 \times 10^{19}$, in $[4, 20] \times [-0.27, +0.27]$

Conjecture

From the numerical results up to 7.6×10^{19}, it would be plausible to conjecture that $\delta(n) = O(n^{1/2})$.

For a random walk with i.i.d. random variables, we would get $\delta(n) \ll \sqrt{n \log\log n}$ almost surely (Khinchin's *law of the iterated logarithm*). More precisely,

$$\limsup \frac{\delta(n)}{\sqrt{2n \log\log n}} = 1 \quad \text{a.s.}$$

The function $\sqrt{\log\log n}$ grows too slowly to make a significant difference to the numerical results.

Thus, to be on the safe side, I will only conjecture that

$$\delta(n) = O(\sqrt{n \log\log n}).$$

A dubious conjecture

John Smith and Ariel Scolnikov (2013)[1] conjectured that

$$\delta(n) = O(\log n).$$

Our numerical results make this conjecture extremely unlikely. For example,

$$\frac{\delta(n)}{\log n} > 2.5 \times 10^7 \quad \text{for} \quad n = 5 \times 10^{19}.$$

[1] `http://planetmath.org/kolakoskisequence`. The conjecture is repeated at `http://codegolf.stackexchange.com/questions/8369/`.

References

A. Carpi, On repeated factors in C^∞-words, *Information Processing Letters* **52** (1994), 289–294.

V. Chvátal, *Notes on the Kolakoski sequence*, DIMACS Tech. Report 93-84, Dec. 1993. `http://dimacs.rutgers.edu/TechnicalReports/abstracts/1993/93-84.html`

F. M. Dekking, *What is the long range order in the Kolakoski sequence?*, Report 95–100, TU-Delft, 1995. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.6839`

W. Kolakoski, Self generating runs, Problem 5304, *Amer. Math. Monthly* **72** (1965), 674. Partial solution: Ü. Necdet, *ibid* **73** (1966), 681–682.

J. Nilsson, A space-efficient algorithm for calculating the digit distribution in the Kolakoski sequence, *J. of Integer Sequences* **15**, article 12.6.7 (2012).

J. Nilsson, Letter frequencies in the Kolakoski sequence, *Acta Physica Polonica A* **126** (2014), 549–552.

References cont.

R. Oldenburger, Exponent trajectories in symbolic dynamics, *Trans. Amer. Math. Soc.* **46** (1939), 453–466.

OEIS, Sequence A000002, Kolakoski sequence, `http://oeis.org/A000002`.

OEIS, Sequence A088568, partial sums of Oldenburger-Kolakoski sequence A000002, `http://oeis.org/A088568`.

M. Rao, *Trucs et bidules sur la séquence de Kolakoski*, Oct. 1, 2012. `http://www.arthy.org/kola/kola.php`.

Wikipedia, *Kolakoski sequence*, `https://en.wikipedia.org/wiki/Kolakoski_sequence`.

Wikipedia, *L-system*, `https://en.wikipedia.org/wiki/L-system`.