Collatz Computation Sequence for Sufficient Large Integers is Random

Wei Ren[1*]

[1]School of Computer Science, China University of Geosciences,
Wuhan, 430074, China

*To whom correspondence should be addressed; E-mail: weirencs@cug.edu.cn.

Collatz conjecture is also known as 3x+1 conjecture, which states that each positive integer will return to 1 after Collatz computations that are either (3x+1)/2 when x is odd or x/2 when x is even. They can be denoted as 'I' computation and 'O' computation, respectively. Given a starting integer, the computation sequence from the integer to 1 consists of 'I' and 'O'. The main results in the paper are as follows: (1) We randomly select an extremely large integer and verify whether it can return to 1. The largest one has been verified has length of 6000000 bits, which is overwhelmingly much larger than currently known and verified, e.g., 128 bits, and its Collatz computation sequence consists of 28911397 'I' and 'O', only by an ordinary laptop. (2) We propose an dedicated algorithm that can compute 3x+1 for extremely large integers in million bit scale, by replacing multiplication with bit addition, and further only by logical condition judgement. (3) We discovery that the ratio - the count of 'O' over the count of 'I' in computation sequence goes to 1 asymptotically with the growth of starting integers. (4) We further discover that once the length of starting integer is sufficient large, e.g., 500000 bits, the correspond-

ing computation sequence (in which 'I' is replaced with 1 and 'O' is replaced with 0), presents sufficient randomness as a bit sequence. We firstly obtain the computation sequence of randomly selected integer with L bit length, where L is 500000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, by our proposed algorithm for extremely large integers. We evaluate the randomness of all computation sequences by both NIST SP 800-22 and GM/T 0005-2021. All sequences can pass the tests, and especially, the larger the better. (5) We thus propose an algorithm for random bit sequence generator by only using logical judgement (e.g., logic gates) and less than 100 lines in ANSI C. The throughput of the generator is about 625.693 bits/s over an ordinary laptop with Intel Core i7 CPU (1.8GHz).

1 Introduction

The Collatz conjecture is a mathematical conjecture that is first proposed by Lothar Collatz in 1937. It is also known as the 3x+1 conjecture, the Ulam conjecture, the Kakutani's problem, the Thwaites conjecture, or the Syracuse problem.

The Collatz conjecture is very simple to state: Take any positive integer x. If x is even, divide it by 2 to get $x/2$. If x is odd, multiply it by 3 and add 1 to get $3x+1$. Repeat the process again and again. The Collatz conjecture is that no matter what the integer (i.e., x) is taken, the process will always eventually reach 1.

The conjecture is so easy to understand with only required concept of addition, multiplication and division, but the study for the conjecture is quite a few due to its well-known hardness. M. Chamberland reviews the works on Collatz conjecture in 2006 (*1*), and some aspects in available analysis results are surveyed. J. C. Lagarias edits a book on $3x + 1$ problem and reviews the problem in 2010 (*2*). He also provides the historical review of the problem by annotated

bibliography in 1963-1999 (*3*) and 2000-2009 (*4*).

Currently, the maximal checked integer is about $593 * 2^{60}$ (*5*), which is no more than 70 bits. D. Barina proposed a new algorithmic approach for computational convergence verification of the Collatz problem (*6*), which can verify much more number of integers in 128 bits per second.

In this paper, we only discuss positive integers (denoted as Z^+). Any odd x will iterate to $3x+1$, which is always even. Collatz computation afterward is always $x/2$. If combing these two as $(3x+1)/2$, then the Collatz computation $T(x)$ can be defined as follows: $T(x) = (3x+1)/2$ if x is odd; Otherwise, $T(x) = x/2$. For the convenience in presentation, we denote $(3x+1)/2$ as '$I(x)$' (or just 'I') and $x/2$ as '$O(x)$' (or just 'O'). Indeed, 'I' is named from "Increase" due to $(3x+1)/2 > x$, and 'O' is named from "dOwn" due to $x/2 < x$.

$T^{(k+1)}(T^{(k)}(x))$ (k is a positive integer) means two successive Collatz computations, where $T^{(k+1)} = I$ if $T^{(k)}(x)\%2 = 1$, and $T^{(k+1)} = O$ if $T^{(k)}(x)\%2 = 0$. For simplicity by using less parentheses, we can rewrite it as $T^{(k)}T^{(k+1)}(x)$. Iteratively, $T^{(k)}(T^{(k-1)}(...(T^{(1)}(x))))$ $k \geq 2, k \in Z^+$ can be written as $T^{(1)}...T^{(k-1)}T^{(k)}(x)$, and $T^{(k)} = I$ if $T^{(1)}...T^{(k-1)}(x)\%2 = 1$ and $T^{(k)} = O$ if $T^{(1)}...T^{(k-1)}(x)\%2 = 0$.

Stopping time of $n \in Z^+$ is defined as the minimal number of steps needed to iterate to 1:

$$s(n) = inf\{k : T^{(1)}...T^{(k-1)}T^{(k)}(n) = 1\}.$$

$T(x)$ is usually either $(3x+1)/2$ or $x/2$ (i.e., $T \in \{I, O\}$), the $s(n)$ is thus the count of $(3x+1)/2$ computation plus the count of $x/2$ computation. (If $T(x)$ is looked as either $3x+1$ or $x/2$, then $s(n)$ should be double the count of $(3x+1)/2$ computation plus the count of $x/2$ computation.)

The Collatz computation sequence (i.e., original dynamics) of $n \in Z^+$ is the sequence of Collatz computations that occurs from starting integer to 1:

$$d(n) = T^{(1)}...T^{(k-1)}T^{(k)},$$

where $T^{(1)}...T^{(k-1)}T^{(k)}(n) = 1$, $k = s(n)$, $T^{(i)} \in \{I, O\}, i = 1, ..., k$.

For example, Collatz computation sequence from starting integer 3 to 1 is $IIOOO$, because $3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. Thus, $s(3) = 5$. $d(3) = IIOOO$.

The ratio of $n \in Z^+$ is the count of $x/2$ over the count of $(3 * x + 1)/2$ in the Collatz computation sequence of n ($|\{...\}|$ returns the number of elements in a set):

$$r(n) = \frac{|\{i|T^{(i)} = O, i = 1, ..., s(n), T^{(1)}...T^{(k-1)}T^{(k)} = d(n)\}|}{|\{i|T^{(i)} = I, i = 1, ..., s(n), T^{(1)}...T^{(k-1)}T^{(k)} = d(n)\}|}.$$

E.g., $r(3) = 3/2 = 1.5$.

The hight of $n \in Z^+$ is the maximal integer (i.e., highest point) to which n iterates:

$$h(x) = sup\{T^{(1)}...T^{(k-1)}T^{(k)}(n) : k \in Z^+.$$

Note that, here $T^{(i)}$ ($i = 1, ..., k$) is either $3 * x + 1$ or $x/2$. That is, $h(x)$ is selected from integers that includes the even integers $3 * x + 1$ before $x/2$ (i.e., $(3 * x + 1)/2$ is separated into two integers $3 * x + 1$ and $x/2$).

E.g., $h(3) = 16$.

2 Results

The Largest Integer being Checked has 6000000 Bits - Only by a Laptop

Currently, the maximal checked integer is 128 bits. In contrast, we can verify much larger integers, e.g., any randomly selected odd integers with extremely large bit lengths, e.g., 6000000 bits, which is much larger than current scale, by only ordinary laptops. (Their objectives are

to verify all integers less than 128 bits; we only verify any randomly selected integers with 6000000 bits.)

We checked a randomly selected odd integer n that has length of 6000000 bits. The stopping time $s(n)$ is 28911397, in which the count of computation $(3x+1)/2$ is 14455482 and the count of $x/2$ is 14455915. The ratio that is the count of '$x/2$' over the count of '$(3x+1)/2$' is 1.0000299215316772.

Note that, above result is only computed by a laptop, instead of any high performance computers, i.e., Lenovo Thinkpad X1 Carbon, with following configurations: Intel(R) Core(TM) i7-10510U, CPU 1.80GHz 2.30GHz, 8.00GB RAM, X86 processor, 64 bit OS Window 10. The compiler is MinGW Developer Studio 2.05 that uses GNU GCC, and source code of our proposed algorithm is ANSI C with no more than 100 lines.

Table 1 shows stopping times of extremely large integers. It also shows the efficiency of the algorithm (proposed later).

Table 1: The Timing Cost for Computing Collatz Computation Sequence of Extremely Large Starting Integers (s: second, m: minute, h: hour, d: day, $\|(x)_2\|$ returns the bit length of x).

$\|(n)_2\|$	$\|(h(n))_2\|$	$s(n)$	Timing Cost
1000	1002	5016	$<$1s
10000	10003	49017	2s
100000	100002	485260	2m34s
500000	500004	2420805	1h4m29s
1000000	1000004	4812415	4h25m51s
2000000	2000003	9644913	23h12m31s
3000000	3000001	14473280	1d22h11m7s
4000000	4000004	19275810	3d5h13m32s
5000000	5000007	24081026	5d7h55m35s
6000000	6000004	28911397	8d1h40m44s

How to Compute (3x+1)/2 for Extremely Large Integers in 6000000 Bit Scale - an Ultra-lightweight Algorithm

Simply speaking, the main heuristics in the algorithm is that we change numerical multiplication into bit addition. That is, we change $(3x+1)/2$ computation into a simple bit addition over the binary representation of x (note that, hereby x is odd). More specifically, x is represented as a bit string (in computer programs it could be an array of bits). E.g., suppose the bit length of x is n. $3x+1$ can be computed by $(2x+1)+x$. $2x$ can be computed simply by left shifting 1 bit of x. Indeed, it can be computed simply by append 0 at LSB (Least Significant Bit) of x. $2x+1$ can be computed by change the LSB of $2x$ from 0 to 1. $(2x+1)+x$ can be computed by adding a bit string with length of $n+1$ (i.e., $(2x+1)$) to a bit string with length of n (i.e., x), note that, bit by bit. The LSB of the summation (i.e., $(2x+1)+x$) must be 0, because x is odd. Then, simply removing this 0 can obtain a bit string directly, which is the division of the summation by 2 (i.e, $(3x+1)/2$).

In our computer programs a bit string is represented by a character array so that it becomes possible to represent and compute extremely large integers. Represent x as an array $A[i]$, $i = 0, ..., n-1$, $A[i] \in \{0,1\}$, where $A[n-1]$ is LSB and $A[0]$ is MSB (Most Significant Bit) of x. $3x+1$ thus can be looked as $A[i]+A[i+1]+c$ for each i ($i = n-2, ..., 0$) where c is current carrier (partially). $(3x+1)/2$ can be computed just by removing the LSB of $3x+1$ that is always 0. Therefore, numerical computation of $(3x+1)/2$ is simplified as bit addition. That is, adding $A[i]$, $A[i+1]$, and current carrier c obtains a summation in $[0,3]$. The LSB of the summation is assigned to $A[i]$; the MSB of the summation is assigned to the next carrier.

Suppose $3x+1$ is represented by $B[0]\|B[1]\|...\|B[n-1]$ (partially). Fig.1 depicts the design rationale as follows:

Eq.1 summarizes bit computation procedures in the computation of $3x+1$ as follows

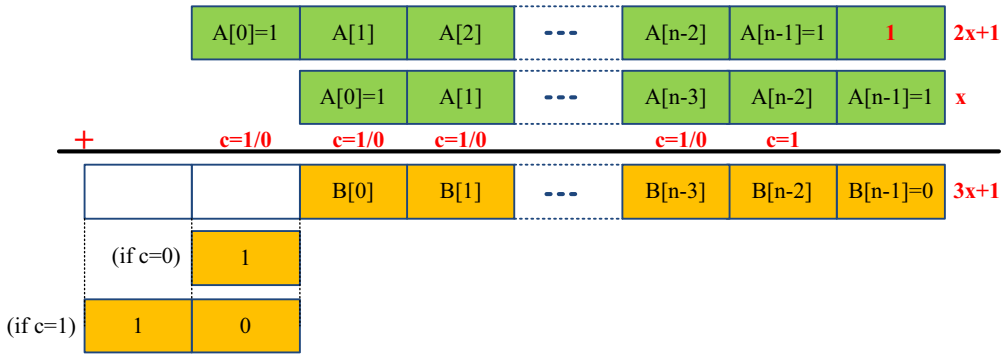

Figure 1: Computation for $3 * x + 1$ to support extremely large integers x via bit addition or even by logic condition judgement, instead of numerical computation such as multiplication. $3x + 1 = \{10/1\}\|B[0]\|B[1]\|...\|B[n-2]\|B[n-1]$. If $B[n-1]$ is removed, then the result is $(3x + 1)/2$.

($LSB(x)$ and $MSB(x)$ returns the LSB and MSB of x, respectively):

$$
\begin{cases}
B[n-1] \Leftarrow LSB(A[n-1] + 0 + 1) = LSB(1 + 0 + 1) = 0 \\
\quad c \Leftarrow MSB(A[n-1] + 0 + 1) = MSB(1 + 0 + 1) = 1, \\
B[n-2] \Leftarrow LSB(A[n-1] + A[n-2] + c) \\
\quad c \Leftarrow MSB(A[n-1] + A[n-2] + c), \\
\quad ... \\
B[n-k] \Leftarrow LSB(A[n-k+1] + A[n-k] + c) \\
\quad c \Leftarrow MSB(A[n-k+1] + A[n-k] + c), \\
\quad ... \\
B[1] \Leftarrow LSB(A[1] + A[2] + c) \\
\quad c \Leftarrow MSB(A[1] + A[2] + c), \\
B[0] \Leftarrow LSB(A[0] + A[1] + c) = LSB(1 + A[1] + c) \\
\quad c \Leftarrow MSB(A[0] + A[1] + c) = MSB(1 + A[1] + c).
\end{cases}
\tag{1}
$$

The MSB (or leftmost two bits) in the binary representation of $3x + 1$ (recall Fig. 1),

depends on whether last $c = 0$ or $c = 1$. If $c = 0$, then $MSB(3x + 1) = 1$ because $A[0] + c = 1 + 0 = 1$. If $c = 1$, then the leftmost two bits are "10" because $A[0] + c = 1 + 1 = 2$. $(2)_2 = 10$. The final computation result of $3 * x + 1$ can be represented as a binary string like $\{10/1\}\|B[0]\|B[1]\|...\|B[n-2]\|B[n-1]$, where $\|$ is concatenation. Obviously, if last $c = 0$, then $(3x + 1)/2$ has the same bit length of x (i.e., n); if last $c = 1$, then the bit length of $(3x + 1)/2$ is 1 more than x (i.e., $n + 1$).

After above preparations, we propose algorithm Alg.1 as follows:

Alg.1 can be revised for computing $(3x + 1)/2$ by simply setting the LSB of $(3x + 1)$ to the terminal symbol instead of '$0'$ (e.g., '$\backslash 0'$ in C language).

Alg.1 is more easier to understood than following enhancement. Especially, it can be easily extended for computing other related $3x + 1$ conjectures (or general cases) such as $qx + 1$ or $3x + q$ ($q \in [1]_2$).

Enhancement Method 1.

Indeed, we can further improve Alg.1 by using logical condition judgement to replace bit addition, in Alg.2 as follows (i.e., the distinction of two algorithms are only operations in the loop):

Enhancement Method 2.

Indeed, $B[i]$ can be omitted and corresponding value can be stored in $A[i + 1]$ where $i = n-2, ..., 0$, thus only one array rather than two is required in the computation (see Alg.3). (This enhancement will be helpful for further hardware design for random bit stream generator.)

As static memory allocated for an array is much less than heap space (virtual memory) dynamically allocated. E.g., by using "malloc()" function in C language, we can store and compute an integer whose bit length is about $2^{32} = 4 * 1024 * 1024 * 1024 \approx 4 * 10^9$ in 32 bit operating systems or 2^{64} in 64 bit operating systems theoretically.

Data: x
Result: $3 * x + 1$.
$B[n-1] \Leftarrow$ '0';
$c \Leftarrow 1$;
for $(i = n-2; i >= 0; i--)$ **do**
  $sum \Leftarrow A[i+1] + A[i] + c$;
  if $sum == 2 || sum == 3$ **then**
    $c \Leftarrow 1$;
  end
  if $sum == 0 || sum == 1$ **then**
    $c \Leftarrow 0$;
  end
  if $sum == 0 || sum == 2$ **then**
    $B[i] \Leftarrow$ '0';
  end
  if $sum == 1 || sum == 3$ **then**
    $B[i] \Leftarrow$ '1';
  end
end
if $c == 1$ **then**
  $result \Leftarrow$ "10"$\|B$;
end
else
  $result \Leftarrow$ '1'$\|B$;
end
return $result$;

Algorithm 1: Input an extremely large integer x that is represented in binary like $A[0]\|...\|A[n-1]$. Output $result = 3 * x + 1$. In code "||" means "or". "$\|$" is concatenation.

Ratio Goes to 1 Asymptotically - with the Growth of Starting Integers

We observe and conjecture that the ratio goes to 1 asymptotically with the growth of starting integers, by empirical analysis. That is,

$$\lim_{n \to \infty} r(n) = 1.$$

Data: x
Result: $3 * x + 1$.
$B[n-1] \Leftarrow$ '0';
$c \Leftarrow 1$;
for $(i = n-2; i >= 0; i--)$ **do**
  if $(A[i+1], A[i], c) == (\text{'0'}, \text{'0'}, 0)$ **then**
    $c \Leftarrow 0$, $B[i] \Leftarrow$ '0', $continue$;
  end
  if $(A[i+1], A[i], c) == (\text{'0'}, \text{'0'}, 1) || (\text{'0'}, \text{'1'}, 0) || (\text{'1'}, \text{'0'}, 0)$ **then**
    $c \Leftarrow 0$, $B[i] \Leftarrow$ '1', $continue$;
  end
  if $(A[i+1], A[i], c) == (\text{'0'}, \text{'1'}, 1) || (\text{'1'}, \text{'0'}, 1) || (\text{'1'}, \text{'1'}, 0)$ **then**
    $c \Leftarrow 1$, $B[i] \Leftarrow$ '0', $continue$;
  end
  if $(A[i+1], A[i], c) == (\text{'1'}, \text{'1'}, 1)$ **then**
    $c \Leftarrow 1$, $B[i] \Leftarrow$ '1', $continue$;
  end
end
if $c == 1$ **then**
  $result \Leftarrow$ "10"$\|B$;
end
else
  $result \Leftarrow$ '1'$\|B$;
end
return $result$;

Algorithm 2: Input an extremely large integer x. Output $result = 3 * x + 1$. In this enhancement, bit addition is replaced by logical condition judgement.

Table 2 shows the trend of ratio.

Randomness Evaluation of Computation Sequence - by NIST Test Suite and GM/T

We discover that $d(n)$ is random for sufficient large n. Of course, 'I' (or 'O') in the sequence $d(n)$ should be replaced by '1' (or '0'), respectively. That is, each computation in the sequence is deterministic, but the computation sequence overall presents randomness.

Data: x
Result: $3 * x + 1$.
$c \Leftarrow 1$;
for $(i = n - 2; i >= 0; i - -)$ **do**
  if $(A[i+1], A[i], c) == (`0', `0', 0)$ **then**
    $c \Leftarrow 0$, $A[i+1] \Leftarrow `0'$, $continue$;
  end
  if $(A[i+1], A[i], c) == (`0', `0', 1)||(`0', `1', 0)||(`1', `0', 0)$ **then**
    $c \Leftarrow 0$, $A[i+1] \Leftarrow `1'$, $continue$;
  end
  if $(A[i+1], A[i], c) == (`0', `1', 1)||(`1', `0', 1)||(`1', `1', 0)$ **then**
    $c \Leftarrow 1$, $A[i+1] \Leftarrow `0'$, $continue$;
  end
  if $(A[i+1], A[i], c) == (`1', `1', 1)$ **then**
    $c \Leftarrow 1$, $A[i+1] \Leftarrow `1'$, $continue$;
  end
end
if $c == 1$ **then**
  $A[0] = `0'$, $result \Leftarrow `1'||A$;
end
else
  $result \Leftarrow A$;
end
$result \Leftarrow A||`0'$;
return $result$;

Algorithm 3: Input an extremely large integer x. Output $result = 3 * x + 1$. In this enhancement, $B[i]$ is omitted by using $A[i+1]$.

The observation that the $r(n)$ goes to 1 when n grows in above section provides a witness on $d(n)$ is a random sequence when n is sufficient large in this section. The evaluation on randomness of $d(n)$ in this section confirm again the empirical analysis on $r(n)$ in above section.

The NIST Test Suite (*7,8*) is applied to verify the randomness of an inputting bit sequence. Here inputting bit sequence is Collatz computation sequence for a randomly selected large integer, after 'I/O' in the sequence is replaced by '1/0', respectively. The evaluation metrics by NIST Test Suite have two folders as follows: (1) The proportion of inputting sequences that pass

Table 2: $r(n)$ is the count of 'O' over the count of 'I'. $abs(x)$ returns the absolute value of x.

$\|(n)_2\|$	$s(n)$	'I'	'O'	$r(n)$	$abs(1-r(n))$
10	83	46	37	0.8043478131294251	0.1956521868705749
20	83	40	43	1.0750000476837158	0.0750000476837158
30	166	86	80	0.9302325844764710	0.0697674155235290
100	550	284	266	0.9366196990013123	0.0633803009986877
500	2197	1071	1126	1.0513539314270020	0.0513539314270020
1000	5016	2534	2482	0.9794790744781494	0.0205209255218506
10000	49017	24617	24400	0.9911849498748779	0.0088150501251221
100000	485260	243072	242188	0.9963632225990295	0.0036367774009705
500000	2420805	1211893	1208912	0.9975402355194092	0.0024597644805908
1000000	4812415	2405366	2407049	1.0006996393203735	0.0006996393203735
2000000	9644913	4823403	4821510	0.9996075630187988	0.0003924369812012
3000000	14473280	7238834	7234446	0.9993938207626343	0.0006061792373657
4000000	19275810	9637963	9637847	0.9999879598617554	0.0000120401382446
5000000	24081026	12038787	12042239	1.0002866983413696	0.0002866983413696
6000000	28911397	14455482	14455915	1.0000299215316772	0.0000299215316772

a statistical test. (2) The distribution of P-values that checks whether the being tested sequences are uniformly distributed.

The significance level is 0.01. The length of testing samples is suggested to 1000000 bits. The other parameters are by default. The test results on existing 15 test metrics by NIST Test Suite are listed in Table 3. The distribution of P-values can be evaluated by a P-value of the P-values ($P - value_T$), which is larger than 0.0001 (if applicable), thus the sequences can be considered to be uniformly distributed. (The details on the test files are provided in supplementary materials such as many files named finalAnalysisReport.txt.)

We also use GM/T 0005-2021 (*9*) for evaluating the randomness of Collatz computation bit sequences. Some of test items, namely, 7, 8, 14, 15 in NIST SP800-22 are not included in the GM/T 0005-2021 specification, but 4 other test items are included - the Poker test, Runs Distribution Test, Binary Derivative Test, the Autocorrelation Test. The significance level is 0.01. The length of testing samples should be 1000000 bits. All testing sequences pass the

Table 3: Test Results. Pass rate 1: The minimum pass rate for each statistical test with the exception of the random excursion (variant) test. Pass rate 2: The minimum pass rate for the random excursion (variant) test.

Length of Staring Integer	Length of Collatz Computation Sequence	Number of Samples	Length of a Sample	The Minimum Pass Rate 1	The Minimum Pass Rate 2
500000	2420805	100	24208	96%	NA
500000	2420805	100	24200	96%	NA
500000	2420805	100	24000	96%	NA
1000000	4812415	200	24062	193/200=96.5%	NA
1000000	4812415	160	30070	154/160=96.25%	NA
1000000	4812415	100	48000	96/100=96%	NA
2000000	9644913	400	24112	390/400=97.5%	NA
2000000	9644913	300	32149	291/300=97%	NA
2000000	9644913	100	96449	96/100=96%	7/8=87.5%
2000000	9644913	60	160748	57/60=95%	8/9=88.89%
3000000	14473280	700	20676	685/700=97.86%	NA
3000000	14473280	400	36183	390/400=97.5%	NA
3000000	14473280	100	144732	96/100=96%	15/17=88.24%
3000000	14473280	90	160814	86/90=95.56%	9/11=81.82%
3000000	14473280	10	1447328	8/10=80%	5/6=83.33%
4000000	19275810	1000	19275	980/1000=98%	NA
4000000	19275810	500	38551	488/500=97.6%	8/9=88.89%
4000000	19275810	100	192758	96/100=96%	28/30=93.33%
4000000	19275810	19	1014516	17/19=89.47%	11/13=84.62%
5000000	24081026	1000	24081	980/1000=98%	NA
5000000	24081026	700	34401	685/700=97.86%	NA
5000000	24081026	440	54729	429/440=97.5%	10/12=83.33%
5000000	24081026	100	240810	96/100=96%	29/31=93.55%
5000000	24081026	24	1003376	22/24=91.67%	15/17=88.24%
6000000	28911397	1000	28911	980/1000=98%	NA
6000000	28911397	700	41301	685/700=97.86%	12/14=85.71%
6000000	28911397	440	65707	429/440=97.5%	13/15=86.67%
6000000	28911397	100	289113	96/100=96%	31/34=91.18%
6000000	28911397	28	1032549	26/28=92.86%	16/18=88.89%

evaluation of GM/T 0005-2021 (Indeed, for starting integer with 100000 bits, all tests pass. For 10000 bits, only one test, i.e., Universal Test, fails). The source code for GM/T 0005-2021 test

suite can be downloaded from GitHub (*10*).

Random Bit Sequence Generator - by only Logic Gates

Due to the evaluations in above section, we thus can propose a method for random bit sequence generator. The rationale is quite simple - randomly select $x \in Z^+$ whose $(x)_2$ is sufficient large. Do following steps iteratively: if $x\%2 = 1$, then output 1 and $x \Leftarrow (3x + 1)/2$; if $x\%2 = 0$, then output 0 and $x \Leftarrow x/2$.

Random bit sequence generator algorithm Alg.4 is proposed as follows:

The proposed algorithm relies on only logical condition judgement, which is much more lightweight than Chaos-based algorithms for random bit sequence generator, e.g., Logistics, Tent, Chebyshev. The other algorithms relying on number theory computation such as modular exponentiation are also computation-intensive.

From the viewpoint of Chaos, the simplest mapping for Chaos is discovered in this paper (only by logical computation).

The proposed algorithm does not rely on any dedicated hardware such as LSFR (Linear Shift Feedback Register). It is suitable for software implementation for random bit sequence generator. Note that, the C language for implementing the algorithm is less than 100 lines (see Data S7.).

The staring integer x that is imported into the algorithm can be looked as a seed for the random bit sequence generator. The bit length of x and T are both security thresholds.

In ordinary laptop, the throughput (i.e., generated bits per second) of random bit sequence generator is $2420805/(64 * 60 + 29) = 2420805/3869 = 625.693bits/s = 78.2bytes/s$ (recall Table 1, $\|(x)_2 = 500000\|$, $s(n) = 2420805$, timing cost is 1h4m29s).

Note that, the processing can be bit-wise parallelization. Check the LSB of the array. If it is 0, then output random bit 0, remove it, and check next LSB of array. If it is 1, then output

Data: $x \in Z^+$, bit length of x is n, $n \geq 500000$. T is a threshold for ending, e.g., 100.
Result: Random Bit Sequence RBG.
while $len(x) > T$ **do**
  $n \Leftarrow len(x)$;
  if $A[n-1] == 0$ **then**
    $RBG \Leftarrow RBG \| `0', A[n-1] \Leftarrow `\backslash 0', x \Leftarrow A$;
  end
  else
    $RBG \Leftarrow RBG \| `1', c \Leftarrow 1$;
    for $(i = n-2; i >= 0; i--)$ **do**
      if $(A[i+1], A[i], c) == (`0', `0', 0)$ **then**
        $c \Leftarrow 0, A[i+1] \Leftarrow 0, continue$;
      end
      if $(A[i+1], A[i], c) == (`0', `0', 1) || (`0', `1', 0) || (`1', `0', 0)$ **then**
        $c \Leftarrow 0, A[i+1] \Leftarrow 1, continue$;
      end
      if $(A[i+1], A[i], c) == (`0', `1', 1) || (`1', `0', 1) || (`1', `1', 0)$ **then**
        $c \Leftarrow 1, A[i+1] \Leftarrow 0, continue$;
      end
      if $(A[i+1], A[i], c) == (`1', `1', 1)$ **then**
        $c \Leftarrow 1, A[i+1] \Leftarrow 1, continue$;
      end
    end
    if $c == 1$ **then**
      $A[0] \Leftarrow `0', x \Leftarrow `1' \| A$;
    end
    else
      $x \Leftarrow A$;
    end
  end
end
return RBG;

Algorithm 4: Random bit sequence generator algorithm. Suppose binary representation of x is $A[0] \| ... \| A[n-1]$.

random bit 1. The other bits in the array start to update. Once the LSB of the other bits are updated, the next random bit can be out and the others can start to update. That is, each bits can be computed in parallel once bit information is available, by full pipelines for all bits.

Certainly, special hardware can also be designed and constructed for the algorithm for further improving the throughput, Fig.2 shows the rationale of possible hardware design. "If-Then" module can be implemented by dedicated hardware such as only logic gates (e.g., and/or gates).

For example, the register has sufficient redundant units (denoted as "#") for storing 1 more bit when last $c = 1$ in a round of $(3x + 1)/2$ computation. If $LSB(x) = 1$ (i.e., $A[n-1] = 1$), then output random bit 1, update and shift right with feedback the register. Otherwise, output random bit 0 and shift right 1 bit of the register. Besides, in "If-Then" module, if and only if one input is "#", there exists one more update line to "#" unit.

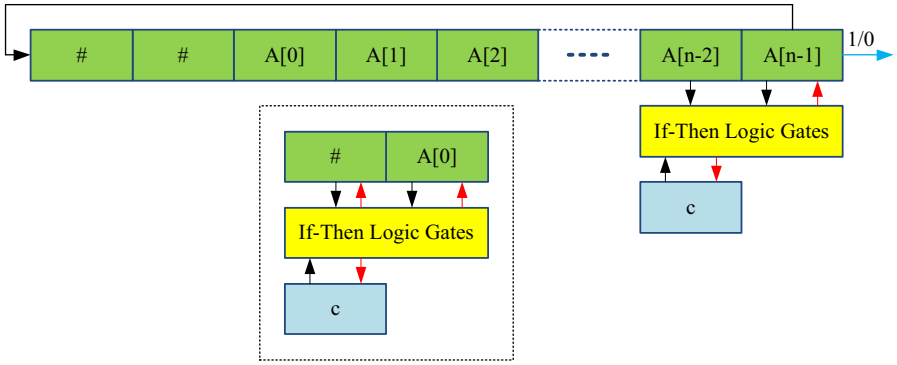

Figure 2: Possible hardware design of random bit sequence generator.

Conclusion

In this paper, we verify the largest integer with 6000000 bits for Collatz conjecture. It can return to 1 and much larger than current known integers that is 128 bits. We also propose algorithms that can verify extremely large integers for Collatz conjecture, by changing multiplication into bit addition, and further into logical condition judgement. We discover that the ratio (i.e., the count of $x/2$ over the count of $(3x + 1)/2$ in $d(n)$) goes to 1 asymptotically with the growth of starting integer n. We discover that the Collatz computation sequence of sufficient large integers is random (pseudorandom). We randomly select some sufficient large integers and obtain

their Collatz computation sequence, and all the sequences can pass the evaluation of NIST randomness evaluations and GM/T 0005-2021. We thus propose a random bit sequence generator algorithm by using the discovery. All source codes to compute Collatz computation sequences and the data (namely, computation sequences consisting of 'I' and 'O' which represents '1' and '0' respectively) are available in open accessible venue (and provided as supplementary materials).

Reporting summary

Further information on research design is available in the Nature Portfolio Reporting Summary linked to this article.

Data availability

Data outputted by our codes and the analysis of the data can be downloaded (*11*). Some examples for the data are included as Supplementary Data S1-S9.

Code availability

All codes required for the paper is ANSI C, and can be downloaded (*11*).

References

1. M. Chamberland, *An Update on the 3x+1 Problem* **Butlleti de la Societat Catalana de Matematiques, 18, pp.19-45**,
(https://chamberland.math.grinnell.edu/papers/3x_survey_eng.pdf)

2. J. C. Lagarias, *The 3x+1 Problem: an Overview* (The Ultimate Challenge: The 3x + 1 Problem, Edited by Jeffrey C. Lagarias. American Mathematical Society, Providence, RI, 2010, pp. 3-29, https://doi.org/10.48550/arXiv.2111.02635)

3. J. C. Lagarias, *The 3x + 1 Problem: An Annotated Bibliography (1963-1999)*, (January 1, 2011 version, https://doi.org/10.48550/arXiv.math/0309224)

4. J. C. Lagarias, *The 3x+1 Problem: An Annotated Bibliography, II (2000-2009)*, (January 10, 2012 version, https://doi.org/10.48550/arXiv.math/0608208)

5. E. Roosendaal, *On the 3x + 1 problem*, (http://www.ericr.nl/wondrous/index.html)

6. D. Barina, *Convergence verification of the Collatz problem*, **The Journal of Supercomputing, 77, 2681-2688**, 2021. https://doi.org/10.1007/s11227-020-03368-x

7. SP 800-22 Rev.1a, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*,
https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final

8. NIST, *NIST SP 800-22: Download Documentation and Software*,
(https://csrc.nist.gov/Projects/Random-Bit-Generation/Documentation-and-Software)

9. State Cryptography Administration, *GM/T 0005-2021: Randomness Test Specifications, 2021*,
(http://www.sca.gov.cn/sca/xxgk/2021-10/19/content_1060880.shtml)

10. GM_nist_sts, https://github.com/dds2333/GM_nist_sts/releases, 2022

11. Wei Ren, *Programs, Data, and Testing Results on Collatz Dynamics of Extremely Large Integers*, **Science Data Bank**, doi:10.57760/sciencedb.07210

Acknowledgement

The research was financially supported by the Provincial Key Research and Development Program of Hubei (No. 2020BAB105), Knowledge Innovation Program of Wuhan-Basic Research

(No. 2022010801010197), the Foundation of State Key Laboratory of Public Big Data (No. PBD2022-13), the Opening Project of Nanchang Innovation Institute, Peking University (No. NCII2022A02), and the Foundation of National Natural Science Foundation of China (No. 61972366). All source codes and data for this paper can be accessed by Science Data Bank (*?*).

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.57760/sciencedb.07210.

Supplementary materials

Materials and Methods

Supplementary Text

Tables S1 to S2

References *(12-20)*

Data S1 to S11