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

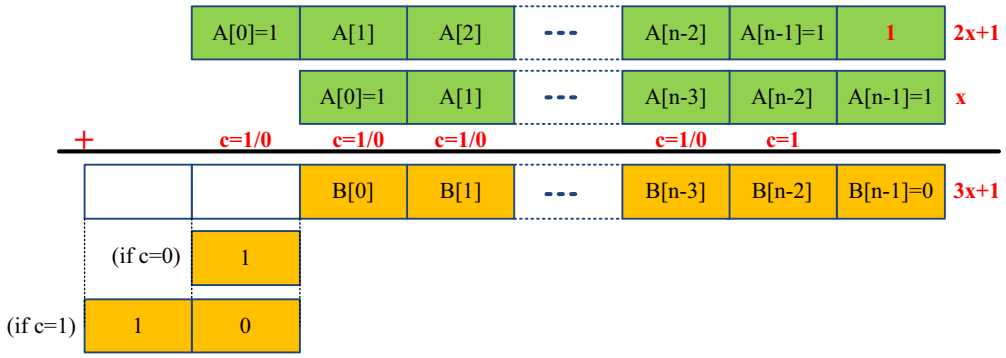


Figure 1: Computation for $3 * x + 1$ to support extremely large integers x via bit addition or even by logic condition judgement, instead of numerical computation such as multiplication. $3x + 1 = \{10/1\}\|B[0]\|B[1]\|...\|B[n-2]\|B[n-1]$. If $B[n-1]$ is removed, then the result is $(3x + 1)/2$.

($LSB(x)$ 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.

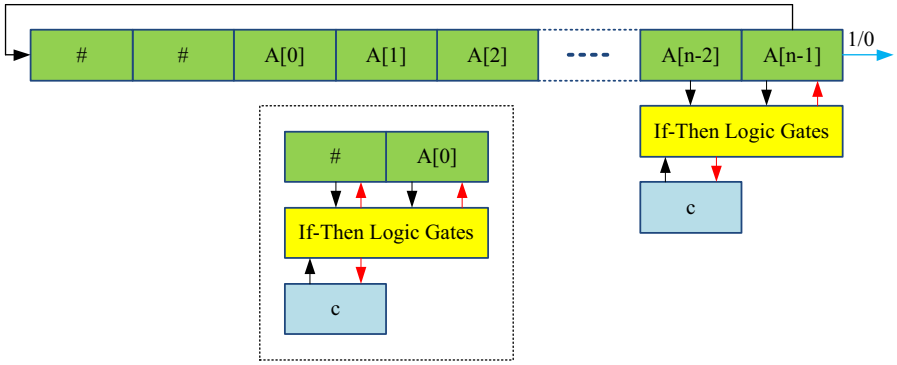


Figure 2: Possible hardware design of random bit sequence generator.

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).

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 S6

References *(12-20)*

Data S1 to S11

Supplementary Materials for

Collatz Computation Sequence for Sufficient Large Integers is Random

Wei Ren
School of Computer Science,
China University of Geosciences, Wuhan, China
Corresponding author: weirencs@cug.edu.cn

The PDF file includes:

Materials and Methods
Supplementary Text
Tables S1 to S6
References (12-20)

Other Supplementary Materials for this manuscript include the following:

Data S1 to S11

Materials and Methods

Methods

The steps for the evaluating the randomness of bit sequences are as follows:

(1) Setup length. Set the length of randomly generated starting integer by setting "UNIT" in C source code txpo46.c (see Data S1). E.g., UNIT=1000.

(2) Compile source code. Compile txpo46.c by any ANSI C compiler (i.e., MinGW or gcc).

(3) Run executable file. The computation sequence will be outputted to a file with the name "txpo46dynamics1000". Some other information such as ratio are also included the file. See Data S2.

(4) Data preparation for further randomness test (i.e., NIST 800-22 and GM/T 0005-2021). For NIST testing, it can be done by replacing "I/O" sequence in data files (e.g., txpo46dynamics1000) with "1/0" sequence, and removing the other information.

We can also generate bit sequence by RBSG source codes - txpo47.c (Data S7) directly, which is revised from txpo46.c.

For GM/T 0005-2021 testing, ASCII data file in which 8 bits are stored as 8 ASCII characters, needs to be converted into byte data file in which 8 bits are stored as a byte.

txpo48.c (see Data S3) provides this converting function. Indeed, ascii2bin.exe is compiled from Txpo48.c. E.g., inputting is the file named 10000, outputting is the file named 10000bin.

(5) Download NIST 800-22 test tool suite source code (*7,8*). Compile the codes in a Linux shell.

(6) Using NIST 800-22 test suite to test the randomness of the data file. See Data S4 and S5.

(7) Download GM/T 0005-2021 test tool from GitHub (*9,10*), either source codes or executable programs - Random_Test.exe compiled by source codes.

(8) Using Random_Test.exe to test the randomness of the data file.

Materials

Source codes and related data for the paper are provided (*11*). The details is listed in Table S1.

Table S1. Explanation on Source codes and Data files

Directory	File Name	Explanation
Dynamics	txpo46.c (see Data S1)	ANSI C code for generate computation sequence
	txpo46-1000.exe	Bit length of starting integer is 1000
	txpo46-10000.exe	Bit length of starting integer is 10000
	txpo46-100K.exe	Bit length of starting integer is 100000
	txpo46-500K.exe	Bit length of starting integer is 500000
	txpo46-1M.exe	Bit length of starting integer is 1000000
	txpo46-2M.exe	Bit length of starting integer is 2000000
	txpo46-3M.exe	Bit length of starting integer is 3000000
	txpo46-4M.exe	Bit length of starting integer is 4000000

	txpo46-5M.exe	Bit length of starting integer is 5000000
	txpo46-6M.exe	Bit length of starting integer is 6000000
	txpo46dynamics1000 (see Data S2)	Output of txpo46-1000.exe
	txpo46dynamics10000	Output of txpo46-10000.exe
	txpo46dynamics100000	Output of txpo46-100K.exe
	txpo46dynamics500000	Output of txpo46-500K.exe
	txpo46dynamics1000000	Output of txpo46-1M.exe
	txpo46dynamics2000000	Output of txpo46-2M.exe
	txpo46dynamics3000000	Output of txpo46-3M.exe
	txpo46dynamics4000000	Output of txpo46-4M.exe
	txpo46dynamics5000000	Output of txpo46-5M.exe
	txpo46dynamics6000000	Output of txpo46-6M.exe
Dynamics2Seq	txpo48.c (see Data S3)	Convert a data file of ASCII sequence (e.g., *) to a data file of byte sequence (e.g. *bin)
	ascii2bin.exe	Compiling from txpo48.c
	10000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from the dynamics sequence in the file txpo46dynamics10000. Note that, the length of sequence is larger than 10000 (i.e., stopping time of n is s(n), $s(n)>(n)_2$).
	10000bin	Data file, include a sequence of 1/0 in byte, converted from data file 10000
	100000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics100000
	100000bin	Data file, include a sequence of 1/0 in byte, converted from data file 100000
	500000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics500000
	500000bin	Data file, include a sequence of 1/0 in byte, converted from 500000
	1000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics1000000
	1000000bin	Data file, include a sequence of 1/0 in byte, converted from 1000000bin
	2000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics2000000
	2000000bin	Data file, include a sequence of 1/0 in byte, converted from 2000000bin
	3000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics3000000
	3000000bin	Data file, include a sequence of 1/0 in byte, converted from 3000000bin
	4000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics4000000

	4000000bin	Data file, include a sequence of 1/0 in byte, converted from 4000000bin
	5000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics5000000
	5000000bin	Data file, include a sequence of 1/0 in byte, converted from 5000000bin
	6000000	Data file, include a sequence of 1/0 in ASCII, by replacing I/O from txpo46dynamics6000000
	6000000bin	Data file, include a sequence of 1/0 in byte, converted from 6000000bin
NIST_results	Console.txt (see Data S4)	Console inputting and feedback by NIST test suite GUI
	10000-1633,30	Sequence generated by 10000 bit length starting integer. Sample length 1633. Sample number 30.
	100000-4852,100	100000 bit length starting integer. Sample length 4852. Sample number 100.
	500000-24000,100	500000 bit length starting integer. Sample length 24000. Sample number 100.
	500000-24200,100	500000 bit length starting integer. Sample length 24200. Sample number 100.
	500000-24208,100	500000 bit length starting integer. Sample length 24208. Sample number 100.
	500000-1210402,2	500000 bit length starting integer. Sample length 1210402. Sample number 2.
	1000000-24062,200	1000000 bit length starting integer. Sample length 24062. Sample number 200.
	1000000-30070,160	1000000 bit length starting integer. Sample length 30070. Sample number 160.
	1000000-48000,100	1000000 bit length starting integer. Sample length 48000. Sample number 100.
	1000000-1203103,4	1000000 bit length starting integer. Sample length 1203103. Sample number 4.
	2000000-24112,400	2000000 bit length starting integer. Sample length 24112. Sample number 400.
	2000000-32149,300	2000000 bit length starting integer. Sample length 32149. Sample number 300.
	2000000-96449,100 (See Data S5)	2000000 bit length starting integer. Sample length 96449. Sample number 100.
	2000000-160748,60	2000000 bit length starting integer. Sample length 160748. Sample number 60.
	2000000-1071656,9	2000000 bit length starting integer. Sample length 1071656. Sample number 9.
	3000000-20676,700	3000000 bit length starting integer. Sample length 20676. Sample number 700.
	3000000-36183,400	3000000 bit length starting integer. Sample length 36183. Sample number 400.

	3000000-144732,100	3000000 bit length starting integer. Sample length 144732. Sample number 100.
	3000000-160814,90	3000000 bit length starting integer. Sample length 160814. Sample number 90.
	3000000-1447328,10	3000000 bit length starting integer. Sample length 1447328. Sample number 10.
	4000000-19275,1000	4000000 bit length starting integer. Sample length 19275. Sample number 1000.
	4000000-38551,500	4000000 bit length starting integer. Sample length 38551. Sample number 500.
	4000000-192758,100	4000000 bit length starting integer. Sample length 192758. Sample number 100.
	4000000-1014516,19	4000000 bit length starting integer. Sample length 1014516. Sample number 19.
	5000000-24081,1000	5000000 bit length starting integer. Sample length 24081. Sample number 1000.
	5000000-34401,700	5000000 bit length starting integer. Sample length 34401. Sample number 700.
	5000000-54729,440	5000000 bit length starting integer. Sample length 54729. Sample number 440.
	5000000-240810,100	5000000 bit length starting integer. Sample length 240810. Sample number 100.
	5000000-1003376,24	5000000 bit length starting integer. Sample length 1003376. Sample number 24.
	6000000-28911,1000	6000000 bit length starting integer. Sample length 28911. Sample number 1000.
	6000000-41301,700	6000000 bit length starting integer. Sample length 41301. Sample number 700.
	6000000-65707,440	6000000 bit length starting integer. Sample length 65707. Sample number 440.
	6000000-289113,100	6000000 bit length starting integer. Sample length 289113. Sample number 100.
	6000000-1032549,28	6000000 bit length starting integer. Sample length 1032549. Sample number 28.
GM_results	10000bin	Sequence generated by starting integers with 10000 bit length. Sequence of 0/1 is in bytes. This file is converted from the file with file name 10000 by ascii2bin.exe.
	10000bin_result.txt	GM/T 0005-2021 testing results. Dump from Random_Test.exe.
	100000bin	Converted by the file 100000
	100000bin_result.txt	GM/T 0005-2021 testing results
	500000bin	Converted by the file 500000
	500000bin_result.txt	GM/T 0005-2021 testing results
	1000000bin	Converted by the file 1000000
	1000000bin_result.txt	GM/T 0005-2021 testing results

	2000000bin	Converted by the file 2000000
	2000000bin_result.txt	GM/T 0005-2021 testing results
	3000000bin	Converted by the file 3000000
	3000000bin_result.txt	GM/T 0005-2021 testing results
	4000000bin	Converted by the file 4000000
	4000000bin_result.txt	GM/T 0005-2021 testing results
	5000000bin	Converted by the file 5000000
	5000000bin_result.txt (see Data S6)	GM/T 0005-2021 testing results
	6000000bin	Converted by the file 6000000
	6000000bin_result.txt	GM/T 0005-2021 testing results
RBSG	txpo47.c (see Data S7)	Random Bit Sequence Generator, ANSI C codes. It is revised from txpo46.c.
	txpo47-1000.exe	The bit length of seed is 1000. Compiled from txpo47.c after setting UNIT=1000.
	txpo47RSBG1000 (see Data S8.)	The output file of txpo47-1000.exe, including seed and generated sequence. Indeed, the sequence is the dynamics with 1/0 instead of I/O.
Othercodes	txpo50.c (see Data S8)	ANSI C code. Use sub-function TXPODT (three x plus one divided two) instead of TXPO in txpo46.c.
	txpo51.c (see Data S9)	ANSI C code. Use one parameter in TXPODT, instead of one in txpo50.c.
sts-2_1_2.zip		NIST test suite for random number generator. Downloaded from NIST (*8*)
GM_nist_sts-2.0.8.zip Random_Test_Binary_2.0.8.zip		GM/T 0005-2021test tool for random bit sequence. Downloaded from GitHub (*10*)
Dynamics10000-75	txpo46dynamics10000, txpo46dynamics20000, ... txpo46dynamics740000, txpo46dynamics750000	txpo46.c with UNIT=10000, RUNTIMES=75. Thus, there are 75 files in this folder. The results is in Table S2.
Dynamics10-10	txpo46dynamics10, txpo46dynamics20, ... txpo46dynamics100 txpo46-10-10.exe	txpo46.c with UNIT=10, RUNTIMES=10. Thus, there are 10 files in this folder. The results is in Table S3.
Dynamics100-100	txpo46dynamics100, txpo46dynamics200, ... txpo46dynamics10000, txpo46-100-100.exe	txpo46.c with UNIT=100, RUNTIMES=100. Thus, there are 100 files in this folder. The results is in Table S4.
Dynamics1000-300	txpo46dynamics1000, txpo46dynamics2000,	txpo46.c with UNIT=1000, RUNTIMES=300. Thus, there are 1000 files in this folder. The

	... txpo46dynamics300000 txpo46dynamics1000-300.exe	results is in Table S5.
Dynamics_Examples	txpo46dynamics2, txpo46dynamics3, ... txpo46dynamics9	Some examples - the dynamics of randomly selected starting integers with short length, e.g., 2 to 9.
Dynamics_txpo49	txpo49.c (see Data S10） txpo49-10-100.exe txpo49-1000-100.exe txpo49-1-10.exe txpo49-1-1000.exe txpo49-1-10000.exe txpo49-1-50000.exe	The first UNIT bits are randomly generated, the other 1 to RUNTIMES bits are generated by appending 1 at LSB. For example, txpo49-10-100.exe generates integers wit length from 11 to 110. The first 10 bits are randomly generated, and the other 1 to 100 bits are generated by appending 1 at LSB. txpo49-1000-100.exe generates integers wit length from 1001 to 1100. The first 1000 bits are randomly generated, and the other 1 to 100 bits are generated by appending 1 at LSB. If UNIT=1, then all bits are 1. For example, txpo49-1-1000.exe generates integers with length from 2 to 1001, and each integer are all bits of 1, e.g., 11, 111, 1111....
Dynamics_txpo53	txpo53.c (see Data S11) txpo53-10000-1000-100.exe txpo53-10000-1000-1000.exe	It generates RUNTIMES random integers with length UNIT+2 (LSB and MSB is 1), in which segment with SEGMENTLENGTH is randomly generated and UNIT/SEGEMENTLENGTH segments are repeatedly. UNIT=10000, RUNTIMES=1000, SEGMENTLENGTH=100. The outputting files are txpo53dynamics10002seglen100run1, ... , txpo53dynamics10002seglen100run1000

Supplementary Text

Related Work

Indeed, we study the problem independently with former methods, which somewhat relies on the observation and induction from the data (e.g., orbit to 1 or to the integer less than the

starting one, or stopping time); those data is from a large number of starting integers (possibly extremely large) that outputted by computer programs. We verified 2^{100000}-1 can return to 1 after 481603 times of 3*x+1 computation, and 863323 times of $x/2$ computation (*11*), which is the largest integer being verified in the world (in this paper, we will update this record). We also pointed out a new approach for the possible proof of Collatz conjecture (*12*). We proposed to use a tree-based graph to represent computation sequences (or reduced dynamics) (*13*) in terms of (3*x+1)/2 and x/2 to reveal two key inner properties in reduced Collatz dynamics: (1) the ratio that is the count of x/2 over the count of 3*x+1, is larger than ln3/ln2 (i.e., the count of x/2 over the count of (3*x+1)/2 is larger than \ln1.5/\ln2), and (2) all positive integers are partitioned regularly corresponding to ongoing dynamics (*14, 15*). Besides, we proves that reduced Collatz dynamics is periodical and the period equals 2 to the power of the count of x/2 (*16*). We also proposed an automata method for fast computing Collatz dynamics (*17*). All source codes and output data by computer programs in our old papers can be accessed in public repository (*18*).

Some Examples for n and d(n)

(1) Staring integer n=11; Collatz computation sequence d(n)=IIOOO.
n=111; d(n)=IIIOIOOIOOO.
n=1101; d(n)=IOOIOOO.
n=10101, d(n)=IOOOOO.
n=100001; d(n)=IOIOIIOOIIOIOOIOOO.
n=1111111; d(n)= IIIIIIIOIOOOIOOIOOIOOIIOIOOIOOO.
n=11101011;
d(n)=IIOIOIIIOIOIOOIIIIIOOOIIOIIIOIIIIOIOOIIIOIIOIIIIIIOOIIIIOOOIOIOIOOOIOOIIIOOOOIOOO.
n=100100101; d(n)=
IOOOIIIOOIIIIOIOIIOIIIOIIIIOIOOIIIOIIOIIIIIIOOIIIIOOOIOIOIOOOIOOIIIOOOOIOOO.

(2) n=1110010001;
d(n)=IOIOOIOIOIOIOOIOIIIIIOIOIIOIIIOIIIIOIOOIIIOIIOIIIIIIOOIIIIOOOIOIOIOOOIOOIIIOOOOIOOO. s(n)=83, r(n)=37/46=0.8043478131294251

(2) n=10011111110101001001;

d(n)=IOIIIOOIOIIIOOIIOOIIOIOOIOOOIOIOOIOIOOIIOIIIOOOOIOIIIOOIOIOOIIOIIOOOIIOOIIOIOOIOOO.

(3) n=11001101010111011010111010011;

d(n)=IIOOIIIOIOIOIOOOOIIOIIIOOOOIIIIIIOOOIIOIOIIIIOOOOIOOOOOIIIOIIOOOIIIIIIIIOIOOIOIIOOOIIOIOIIIIIOIIOIOIIOIOIOOIIOIOOIIIIOIIOOOOOIOIOIOIOOIOIIOIOIIIIOOIOIOOOIIIOOOOOOOOO.

(4)
n=11010001110011000101100001001100010010001010110010011101110100010111100000011111000010101010011100 11;

d(n)=IIOOIOIOIIIIOIIIOIOOIOOOOOIIIOIOIIOIOIIOOIIIIIOIOIIIOOOIIOOIOIOIIIOIOIOOI
OOIOOOIOIOOOIOOIOIOOOIIOOOOIOIOOOOOIIIOOOIOIOIOOOIIOOIIIIOOOOIIOIOOI
IOIOOIIIIOOOOIIIIIOIOIIIOOOIOOOIOIOIIOIIOIIOIIIIIOOOOOOIOIIOIIOIIIOIIOOIOOO
IOIIIIIOIOIIOIIIIIOIOOIIOOOIOOOIOOIOOOOIOOIIIIIIIOIOIIIOOOIOIOIIOIOIOOIIIIIOII
OOIIOOOOOOIIIOIIIIIIIIIIIIOIOIOOIOIOOOIIIOIIOIIIOIOIIOOIIOOOOIOOOIOIOOOOIOI
IOIIIIIOOIIOIIIOIOIIOOIIOOOOIOIIIOOIIIIIIIOOOOIOIIIOOIIOOIIOIOIIIOOIOIIIOOIIOII
IOOOOIIOIIOIOIIOOIIIOIOIOIIOIIOIOIOIOOIOOIIIIIOOIIIOIOOIOOOIOIOIOOIIOIOIOOI
OOOIIOIOOOOOOOOOOIOIOOIOOO.

Verified Computation for Extremely Large Integers

We randomly selected an odd starting integer with designated large bit length to check whether it can return to 1. The checking for extremely large starting integer can be used for verifying a witness or finding a counter-example. Indeed, intermediate integers in the orbit are also verified in the same run. E.g., if the length of starting integer is 6000000, then the count of computation time is 28911397, in which the count of (3*x+1)/2 computation is 14455482 and the count of x/2 computation is 14455915. Thus, 14455482*2+14455915 intermediate integers are checked too.

The proof for the rightness of the conjecture cannot solely rely on the computation, but some observations on the computation process may provide some heuristics for the approaches to the final proof. We pointed out the reduced Collatz conjecture is equivalent to the Collatz conjecture, and reduced Collatz dynamics is more primitive to original Collatz dynamics \cite{weijm}. We also prove some properties such as ratio of reduced Collatz dynamics \cite{weiratio}.

By observing Table 1, we can give two conjectures on empirical bounds as follows:

(1) For sufficient large n, $s(n)=K*||(n)_2||$, where $||(n)_2||$ is a bit length of n. $4.8<K<5.1$.

(2) $||(h(n))_2||=H+||(n)_2||$, e.g., $1 \leq H \leq 10$. If $||(n)_2|| \geqslant 1000$, then $H<<||(n)_2||$ for sufficient number of n (the percentage is proportional to $1-1/||(n)_2||$).

Table S2 lists results for dynamics of 10000 and 100000 scale (see Table S1, Dynamics10000-75).

Table S2. s(n) and r(n) for 10000 and 100000 scale starting integers

$\|(n)_2\|$	$\|h(n)_2\|$	s(n)	I	O	r(n)
10000	10003	49017	24617	24400	0.9911849498748779
20000	20005	94315	46888	47427	1.0114954710006714
30000	30001	144578	72291	72287	0.9999446868896484
40000	40002	189614	94396	95218	1.0087080001831055
50000	50002	240070	119921	120149	1.0019012689590454
60000	60002	288938	144444	144494	1.0003461837768555
70000	70004	337784	168953	168831	0.9992778897285461
80000	80002	383888	191732	192156	1.0022114515304565
90000	90002	435783	218165	217618	0.9974927306175232
100000	100004	480830	240277	240553	1.0011487007141113

150000	150002	723899	362090	361809	0.9992239475250244
200000	200002	967819	484440	483379	0.9978098273277283
250000	250004	1201442	600293	601149	1.0014259815216064
300000	300002	1446153	723142	723011	0.9998188614845276
350000	350002	1694103	848035	846068	0.9976805448532105
400000	400006	1927016	963440	963576	1.0001411437988281
450000	450004	2173208	1087223	1085985	0.9988613128662109
500000	500002	2407610	1203568	1204042	1.0003938674926758
550000	550002	2635879	1316043	1319836	1.0028821229934692
600000	600002	2895655	1448397	1447258	0.9992136359214783
650000	650001	3143125	1572987	1570138	0.9981887936592102
700000	700003	3361489	1679213	1682276	1.0018240213394165
750000	750003	3616528	1808578	1807950	0.9996527433395386

Table S3 lists results for dynamics of 10 scale (see Table S1 Dynamics10-10).

Table S3. s(n) and r(n) for 10 scale starting integers

$\|\|(n)_2\|\|$	$\|\|h(n)_2\|\|$	s(n)	I	O	r(n)
10	14	83	46	37	0.8043478131294251
20	22	83	40	43	1.0750000476837158
30	32	166	86	80	0.9302325844764710
40	44	224	116	108	0.9310345053672791
50	54	243	122	121	0.9918032884597778
60	63	244	116	128	1.1034482717514038
70	72	219	94	125	1.3297872543334961
80	85	438	226	212	0.9380530714988709
90	92	381	184	197	1.0706521272659302
100	103	550	284	266	0.9366196990013123

Table S4 lists results for dynamics of 100 scale (see Table S1 Dynamics100-100).

Table S4. s(n) and r(n) for 100 and 1000 scale starting integers

$\|\|(n)_2\|\|$	$\|\|h(n)_2\|\|$	s(n)	I	O	r(n)
100	106	509	258	251	0.9728682041168213
200	204	1037	528	509	0.9640151262283325
300	306	1400	694	706	1.0172910690307617
400	403	2080	1060	1020	0.9622641801834106
500	507	2197	1071	1126	1.0513539314270020
600	602	2476	1184	1292	1.0912162065505981
700	703	3439	1728	1711	0.9901620149612427
800	802	4071	2064	2007	0.9723837375640869
900	903	4609	2340	2269	0.9696581363677979
1000	1002	5016	2534	2482	0.9794790744781494
2000	2004	9623	4810	4813	1.0006237030029297
3000	3004	15028	7589	7439	0.9802345633506775
4000	4002	19314	9662	9652	0.9989650249481201
5000	5002	23984	11978	12006	1.0023375749588013

6000	6002	29075	14559	14516	0.9970465302467346
7000	7002	33919	16984	16935	0.9971149563789368
8000	8005	39786	20055	19731	0.9838443994522095
9000	9001	42794	21322	21472	1.0070350170135498
10000	10003	49171	24714	24457	0.9896010160446167

Table S5 lists results for dynamics of 100 scale (see Table S1 Dynamics1000-200).

Table S5. s(n) and r(n) for 1000 scale starting integers

$\|\|(n)_2\|\|$	$\|\|h(n)_2\|\|$	s(n)	I	O	r(n)
1000	1002	4636	2294	2342	1.0209240913391113
2000	2002	9625	4811	4814	1.0006235837936401
3000	3003	14147	7033	7114	1.0115171670913696
4000	4002	18757	9311	9446	1.0144989490509033
5000	5005	24403	12242	12161	0.9933834075927734
6000	6003	29132	14595	14537	0.9960260391235352
7000	7005	32781	16266	16515	1.0153080224990845
8000	8002	37401	18550	18851	1.0162264108657837
9000	9006	42316	21020	21296	1.0131303071975708
10000	10005	48508	24296	24212	0.9965426325798035
20000	20004	97148	48675	48473	0.9958500266075134
30000	30002	145439	72834	72605	0.9968558549880981
40000	40002	193632	96931	96701	0.9976271986961365
50000	50002	245116	123105	122011	0.9911133050918579
60000	60002	289997	145112	144885	0.9984356760978699
70000	70001	333907	166507	167400	1.0053631067276001
80000	80002	382858	191082	191776	1.0036319494247437
90000	90003	433117	216483	216634	1.0006974935531616
100000	100008	483030	241665	241365	0.9987586140632629
150000	150003	721043	360288	360755	1.0012961626052856
200000	200004	959794	479377	480417	1.0021694898605347
250000	250003	1196104	596925	599179	1.0037760734558105
300000	300002	1445457	722703	722754	1.0000705718994141

Ratio is Larger than the Bound of Ratio of Reduced Dynamics

Reduced stopping time of positive integer n is defined as the number of steps needed to iterate to the first integer less than n:

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

The Reduced Collatz computation sequence (i.e., reduced dynamics) of positive integer n is the sequence of Collatz computations that occurs from starting integer to the first integer less than n:

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

We define two functions as follows:

Function CntI: c $\rightarrow$ y. It takes as input c$\in${I,O}$^{\geqslant 1}$, and outputs y $\in$ Z that is the count of "I" in c.

Function CntO: c $\rightarrow$y. It takes as input c$\in${I,O}$^{\geqslant 1}$, and outputs y $\in$ Z that is the count of "O" in c.

E.g., CntI(IIOO)=2, CntO(IIOO)=2.

For positive integer n, if CntO(d_r(n))/CntI(d_r(n)) > λ, then r(n) >λ.

The proof is straightforward. Simply speaking, the original dynamics of n can be looked as the appending of multiple reduced dynamics \$n,n_1,...,n_k=2\$, where d_r(n)(n)<n, d_r(n)(n)/2^{p1}=n_1, ... , d_r(n_k)(n_k)=d_r(2)(2)=O(2)=1. Note that, d_r(n)(n) is the first integer less than n after computation sequence d_r(n). As in each reduced dynamics the count of "O" over the count of "I" is larger than λ, besides p_i (i=1,...,n) is only adding the ratio, thus r(n) is also larger than λ.

We proved that for reduced dynamics CntO(d_r(n))/CntI(d_r(n))>λ=$\log_2$1.5 \cite{weiratio}. Thus, the ratio asymptotically goes to 1 again confirms our conclusion.

Randomness Evaluation Items in NIST Test Suite

There exist 15 test metrics in the NIST Test Suite, which is listed in Table S6.

Table S6. Test Items in NIST Test Suite

Items	Contents
1	The Frequency (Monobit) Test
2	Frequency Test within a Block
3	The Runs Test
4	Tests for the Longest-Run-of-Ones in a Block
5	The Binary Matrix Rank Test
6	The Discrete Fourier Transform (Spectral) Test
7	The Non-overlapping Template Matching Test
8	The Overlapping Template Matching Test
9	Maurer's ``Universal Statistical'' Test
10	The Linear Complexity Test
11	The Serial Test
12	The Approximate Entropy Test
13	The Cumulative Sums (Cusums) Test
14	The Random Excursions Test
15	The Random Excursions Variant Test

Extension to General Collatz Conjecture and Computations

In this section, we will discuss two problems as follows:

(1) The proposed algorithms can be used for computing other related 3x+1 problem such as general 3x+1 problem or not.

(2) The dynamics in the other related 3x+1 problems maintain randomness or not.

We define a*x+b conjecture (a≠b, a,b are positive integers). Given any integer, if it is odd, than compute a*x+b; if it is even, then compute x/2. Compute literately, then x will return to 1.

Proposition 1. In general a*x+b conjecture, a and b must be both odd. Otherwise, conjecture is wrong.

Proof. The proof is to exclude the other three possible cases as follows:

(1) a is odd and b is even. In this case, a*x+b must be odd (recall x is odd). Thus, the computation for odd x will be infinite, instead of return to 1.

(2) a is even and b is odd. In this case, a*x+b must be odd. Thus, the computation will be infinite, instead of return to 1.

(3) a is even and b is even. In this case, a*x+b can be rewritten as $(a'*x+b')*2^c$ where c is a positive integer and $c=\max(\{c'|2^{c'}|a, 2^{c'}|b\})$. Either a'=a/c or b'=b/c will be odd. If a' and b' has distinct oddness, then the results will be degenerated to case (1) and (2). If a' and b' are both odd, then the problem is degenerated to the case to be proved - a and b are both odd.

Therefore, a and b are both odd. That is the end of the proof.

3x+q conjecture, where q is a positive odd integer, q⩾5, 3 \not|q. The Alg.\ref{alg:mainalg} can be revised accordingly just by adding q instead of 1. Nonetheless, when q=5,7,13,17,19, 3x+q conjecture is false by our simply computing.

q*x+1 conjecture, where q is a positive odd integer. The Alg.\ref{alg:mainalg} can be extended for (q*x+1)/2 computation, if $t = \log_2(q-1) \in \mathbf{Z}^+$. $(q*x+1)=(2^t+1)x+1=2^tx+x+1$. Appending t 0s after binary representation of x will obtain 2^tx. Afterwards, 2^tx+x+1 is similar to 2*x+x+1. Otherwise, q will be represented by binary and then q-1 can be represented by the addition of the power of 2. Nonetheless, when q=5,7,9,11,13,15,17,19, q*x+1 conjecture is false by our simply computing.

References

(12) Wei Ren, Simin Li, Ruiyang Xiao and Wei Bi, Collatz Conjecture for 2^{100000}-1 is True - Algorithms for Verifying Extremely Large Numbers, Proc. of 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)}, Oct. 2018,

Guangzhou, China, 411-416, 2018. doi:10.1109/SmartWorld.2018.00099, https://ieeexplore.ieee.org/document/8560077

(13) Wei Ren, A New Approach on Proving Collatz Conjecture, Journal of Mathematics, Hindawi, April 2019, ID 6129836, doi:10.1155/2019/6129836, https://www.hindawi.com/journals/jmath/2019/6129836/

(14) Wei Ren, Ratio and Partition are Revealed in Proposed Graph on Reduced Collatz Dynamics, Proc. of 2019 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), pp. 474-483, 16-28 Dec. 2019, Ximen, China. doi:10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00076, https://ieeexplore.ieee.org/document/9047265/

(15) Wei Ren, Ruiyang Xiao, How to Fast Verify Collatz Conjecture by Automata, Proc. of IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 2720-2729, 10-12 Aug. 2019, Zhangjiajie, China. doi:10.1109/HPCC/SmartCity/DSS.2019.00382, https://ieeexplore.ieee.org/document/8855475/

(16) Wei Ren, A Reduced Collatz Dynamics Maps to a Residue Class, and its Count of x/2 over Count of 3*x+1 is larger than ln3/ln2, International Journal of Mathematics and Mathematical Sciences, Hindawi, Volume2020, Article ID 5946759, doi: 10.1155/2020/5946759, https://www.hindawi.com/journals/ijmms/2020/5946759/

(17) Wei Ren, Collatz Dynamics is Partitioned by Residue Class Regularly, techrxiv, doi:10.36227/techrxiv.11742669.v1, https://www.techrxiv.org/ndownloader/files/21377280

(18) Wei Ren, Collatz Dynamics is Partitioned by Residue Class Regularly, Preprint, 2022, doi: 10.20944/preprints202209.0155.v1, https://www.preprints.org/manuscript/202209.0155/v1

(19) Wei Ren, Reduced Collatz Dynamics is Periodical and the Period Equals 2 to the Power of the Count of x/2, techrxiv, doi:10.36227/techrxiv.11664501.v1, https://www.techrxiv.org/ndownloader/files/21196512

(20) Wei Ren, Reduced Collatz Dynamics Data Reveals Properties for the Future Proof of Collatz Conjecture, Data, MDPI, 2019, 4, 89, doi:10.3390/data4020089, https://www.mdpi.com/2306-5729/4/2/89/pdf.

Data S1. (txpo46.c)

```
////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2023, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: The dynamics of an extremely large integer with binary length related to UNIT.
//Input:
//Output: Original Dynamics, namely, Collatz computation sequence
//Usage Method: Set UNIT to bit length what you want,
//            e.g., 1000, 10000, 100000, 1000000, ..., and compile (MinGW or GCC),
//            and execute the executable file in a shell (DOS or Linux)
//Usage Example: txpo46.exe
//Output Example: txpo46dynamics1000
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>   //if to generate different random integer in each run, enable it

static int MAXLEN=6000000; //it is used for setting the max length in malloc
static int UNIT=1000;     //it is used for setting the length of staring integer
static int RUNTIMES=1;     //it is used for setting the count of run

//Three X Plus One subfunction, result <= 3*a + 1
int TXPO(char* a, char* result);

int main(int argc, char *argv[]) {

    char* current_x;    //the current integer
    char* temp_x;       //temporary integer in computing dynamics, i.e., obit
```

```
char* dynamics;    //dynamics in terms of `I' and `O'

FILE *fp_dynamics;  //a file to store dynamics in terms of 'I' and 'O', 'I' means (3x+1)/2, and 'O' means x/2

int count_I, count_O;     //the count of 'I' and the count of 'O' in dynamics
int len;
int len_of_heightest;    //the length of the heightest, that is the largest length of the binary value in orbit

float ratio_OI=0;        //ratio = count_O/count_I = (the count of `O')/(the count of `I')

int i,j;                 //for loop

int start_x_len=0;       //the length of the starting integer
char filename[40];       //filename of the generated file recording original dynamics in terms of 'I' and 'O'
char extendname[20];     //filename extension to record start_x_len

        current_x = (char *)malloc(MAXLEN*5*sizeof(char));
        temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
        dynamics = (char *)malloc(MAXLEN*10*sizeof(char));

        if((current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
        {
                printf("debug: error of malloc.\n");
        }


        for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
        {
        start_x_len = UNIT*(i+1);  //e.g., if UNIT=1000, RUNTIMES=5, then start_x_len =1000,2000,3000,4000,5000

        strcpy(filename,"txpo46dynamics");  //prepare dynamics file name, the name is
        itoa(start_x_len,extendname,10);    //change maxlen to a string - extendname
```

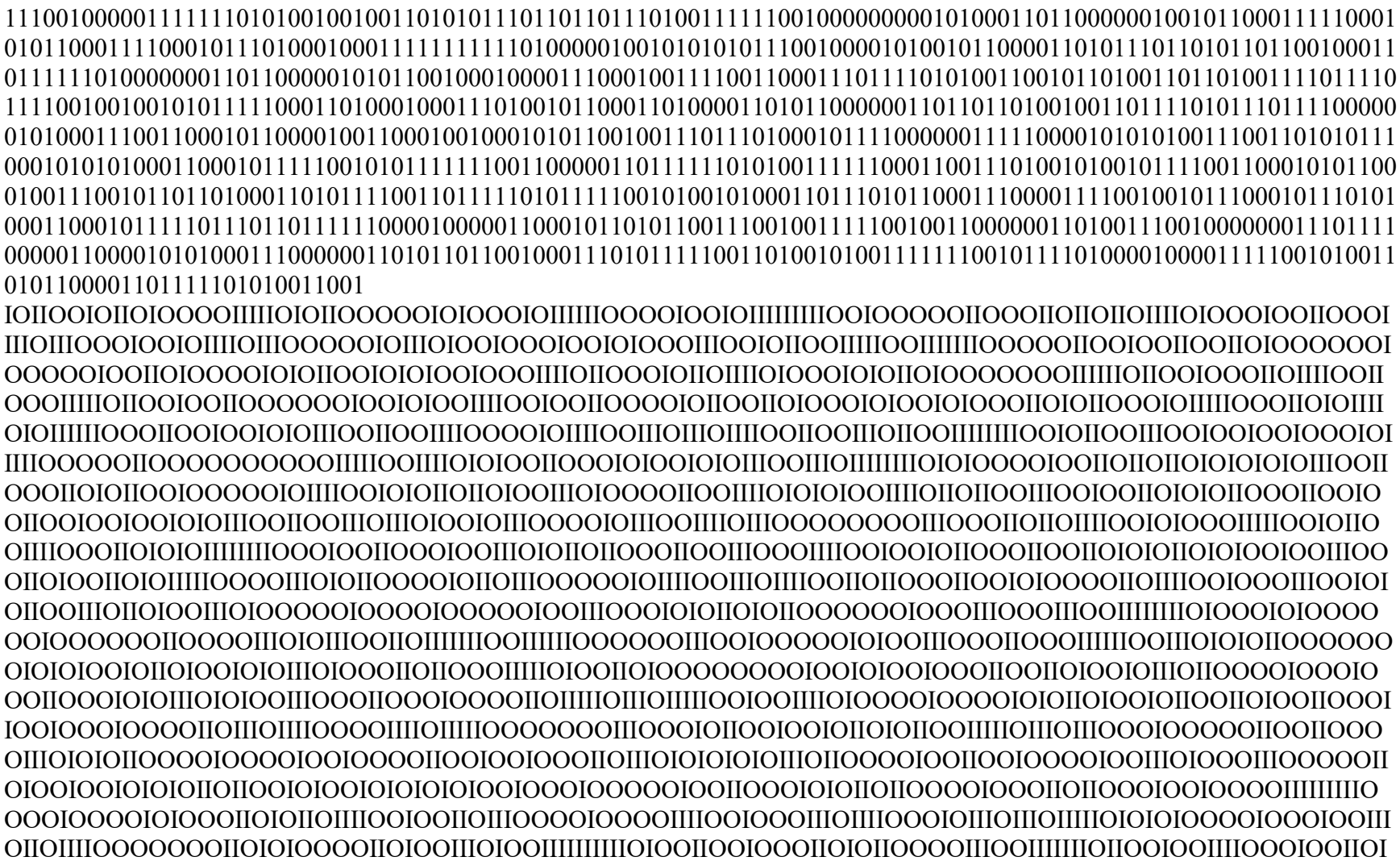
```
strcat(filename,extendname);        //append maxlen to the file name
//printf("debug: %s\n",filename);

fp_dynamics = fopen(filename,"w");

memset(current_x,0,sizeof(current_x));
memset(temp_x,0,sizeof(temp_x));
memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics

//Generate a random interger with the bit length of maxlen as a starting integer
//srand(time(NULL));        //for the random seed generation. if enabled, random integer in each run is different
strcat(current_x,"1");      //the MSB should be 1
for(j=0; j < start_x_len-2; j++)     //2 bits are fixed, MSB and LSB
{
        if(rand()%2==1)        //rand() returns an random integer
                strcat(current_x,"1");
        else
                strcat(current_x,"0");
}
strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
//strcpy(current_x, argv[1]);   //it could also be an inputting string

//printf("debug: start_x=%s sizeof(current_x)=%d strlen(current_x)=%d \n", start_x, sizeof(current_x), strlen(current_x));
fprintf(fp_dynamics, "%s\n", current_x);  //writing start_x into data file

count_I=0;
count_O=0;
len_of_heightest=0;

while(strlen(current_x)!=1)        //if outputting is original dynamics instead of reduced dynamics
{
len = strlen(current_x);
```

```
        if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
        {
        TXPO(current_x,temp_x);    //call subfunction TXPO(), i.e., temp_x <= current_x*3+1

        strcpy(current_x,temp_x);  //current_x <= temp_x;

        len = strlen(current_x);   //note that, current_x is changed, so hereby need to get new length
        current_x[len-1] = '\0';   //cut down the last bit that is always 0, it means current_x <= current_x/2

        if(len > len_of_heightest)    //store the largest len, which is only required in (3x+1)/2 computation
                len_of_heightest = len;

        strcat(dynamics,"I");     //debug - 'I' that means (3x+1)/2, is appended into dynamics

        count_I = count_I + 1;   //count the number of 'I'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
      else                      //current_x is even
      {
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

        strcat(dynamics,"O");     //debug - 'O' that means x/2, is appended into dynamics

        count_O = count_O + 1;     //count the number of 'O'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
    }       //end of while loop

        ratio_OI = (float)count_O/count_I;    //computing ratio

fprintf(fp_dynamics, "%s\n", dynamics);  //writing dynamics into data file
fprintf(fp_dynamics,"%d %d %d %d %d %18.16f\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O, ratio_OI);
fclose(fp_dynamics);
```

```
        //0.5849625007211561 < log(1.5)/log2 < 0.5849625007211562, which is the estimated bound
        if (ratio_OI <= 0.584962500721156)
        {
                printf("\n\n\n------Alert! Our Esitmated Bound is Wrong ------\n\n\n");
                exit(0);
        }

        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }       //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        free(temp_x);
        free(dynamics);

   return 1;
}


int TXPO(char* a, char* txpo_result) //this subfunction is compute 3x+1, that is, txpo_result <= 3*a + 1
{

   //recall that, `10' or `1' is appended by "a||0" is txpo_result

   //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
   int c = 1;       //only current carrier needs to be stored.

   int i;       //counter in for loop

   int n = strlen(a);       //the bit length of a, a is the inputting string, to compute 3*a+1
   //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);
```

```
    memset(txpo_result, 0, sizeof(txpo_result));
    //printf("debug: txpo_result=%s, strlen(txpo_result)=%d\n", txpo_result, strlen(txpo_result));

    //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
    for (i=n-2; i>=0; i--)
    {
                    if(a[i+1]=='0' && a[i]=='0' && c==0)
                            { c=0; a[i+1]='0'; continue;}
                    if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                            { c=0; a[i+1]='1'; continue;}
                    if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
                            { c=1; a[i+1]='0'; continue;}
                    if(a[i+1]=='1' && a[i]=='1' && c==1)
                            { c=1; a[i+1]='1'; continue;}

       //printf("debug: in TXPO's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
    }

    if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
       { a[0]='0'; strcat(txpo_result,"1"); }

    strcat(txpo_result, a);    //the other bits are appended to right
    strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even
    //printf("debug: in TXPO(): txpo_result = %s \n", txpo_result);

    return 1;

}
```

Data S2. (txpo46dynamics1000)



OIOIIIOIIIIIOOIOOOIOIIIIIIIOIOOOIIIOIOIOIOOIOOOIOOIOIIOOOOIIOOIIIOOOOIIIIOIIOOIOIOIOIIOOOOOIIOOOOOIOO
OOIIOIOIOOIOOIIOOOOOIOIIOIOOOOOIIIIIIIIOIIIIIIIOOIIOOIIIOIOOIOIOOOOOIOOIIIOOOIIOOOOOOIOIOIIIIOIOIIOII
IOIIIOIOIIIOOIIIIIOIOIOIOOIIOOOOOOOIOOIIIOOOIOIOIIOOIOIIOOIIIOIOOOOOOIOIIIIOIIOIOOIIIOOIOIIIIOIIIOIOIIOI
OIOIOOOOIIIOIIIIOIIOOOOIIIIIIOOIIIIIOOOIIIIOIIOOIIIOOOOIIIOIIOIIOOIOIIOIIOOIIOIOOIOIIIOIIIIIOOOOOIOOIIIIOIIO
OIIOOOIIIIIOOIOOIIIIIOOOOOIOOIIIIIIOOIOOOOOIOIOOOOOIIOOIIIOOOOIOOOIOOOOOIIIIOIOOOIIIIOIOOIOOIIIIIOOI
IOIOOOOOIIOOIIIIOOOOOIIIIOOOOIOIOIOIOOOOIIOIIOOOIIOOIIOIIOIIOOOIIOOIOOIIOOIIOIOIIOIOIOIIOIIIIIOOOIOIO
IOIIIIIIOOOIIIIIOOOIOOIOIOOIOOOIIOIIIOIIOIOIIOOOOIIIOIIOOOIIOOIOOOOOIOIOOOOOOOIOIOIIIIIOOOIIIIOOIIIIOOO
OIOIOOIOOOIIIIOOOIIOOOIIIIIOOOOOOOOOOOOOIIIIOOIOOIOIIOOIOOOOIIOOIOOIIOOOIOOOOIOOIIIIOOIOIIOIOIIOO
OIIOOIOIIIOIIIIIIOOOIOIIIIOOIIOIOOOOIIOOIOIIOIIIIIIOOOIIIOIIIOIIOIOIIIOOIOOOOIOIIIOOOIOOOOOIOIIIOIIIOOOIO
OOIOOIIIIOOOOOOIOIIOOOIOOOOIOIIIOOIOOIIOOIIIIIIIIIOOOOOIIIOIIIIIOIOOOIOIOOIOOOOIIIIIOOIIIIIOIIIOIOIOIOO
IOOOOIOOOIOIOIIIIOIOOOIIOOIIOOIOIOIIOOIOIIIIOOIOIOOOIOOOIOIIIOIIIIIIIIIOIOOIOIOOIOIOOOOOOIIOIIIIIOOOO
OOOIOIIOIIIIIOOOOIIOIOIIOIOOOIOOIIIIIIIOIIIIIIIOOIOOOOIIIOIIOOOOIOIOIIIIOOIOIOIIOOIIIOOIOOOOIIIOOIIOIIIO
OOIIOOIIOIOIIOOIIIIOOIOIOOOOIOOOOIOOOIIIIOIIOOIOIOOIIIIIIOIOIOIOOOOIOOIIIIOOOIOOOOIOOIIIOOOOOOIIOIII
OOOIIIOOOOOOOOOOIOIIIIOIIIIOOIOIIOOOIOOOOIOOIIOOIIOIIIOIOOOIOOOIIIOIIOIIIOOOOIOOIIOOOIIOIOOOOIOIIOI
OIOOOOIIOIIIOIIOIOOOOOOIIOOIOOIIIIOOOIOOIIIOOOIIOOOOOOIOIIIOIIOOOIOIIIIIOIOIIOOOOOIIOOOIOOOIOOOII
OIIIIIIOIIOIIIIIIIOIIOIOIOIIOOIIIOIIIIOIIOIOIIIOOIOIOIOOOOIOIOIIOIIOIIIIOIOIOOIOIOOOOOOIIOIOIOIOOIIIIIIIIOIOO
IOIIOOOIOOIOIIOOIOOIIIIIOOIOOOOOIOOOIOOIOIIIOOOIIIOIIOOOIOIOOOOIOIOIIIOOOIIIOOIIOOOIOIOOIOIOOIIIOO
OIOIIIOOOOIOOIIOIOIIOIIOOOOIIOIIIOOIOOIIIOOOIOIOOOOOOOIOIIIIIOOOIOIOOIOIIOIOOIIIOOOIOOIOOOIOOOIOOO
IOIOIOIOOOIOOOIOOIIOIIIIOIOOIIOOIOOIOIIIIOIOOOIOIOIOIIIIIIIOIIOOOIIIIOOIIOOOIOIIOIIOOOOIOIIOOOOIIOOIOI
OOIOIIIOIOOIIOOOOIIIIOIIOOOOIOIOIIIOIOIOOOIOIIOIIOOIIIIIIOOOIOIIIOOIIOOIOIOIIOIOOOIOOIOOOOOOIIOIOOIOI
IOOIOIIOIIOIIIIOIIOIIOOIOOOIOIIOOOIOIOOOOIIOOIOOIIIIOIIOOOIOOOIIOIIOIIOIOOIOOIOOOIOOIOOIOOOOOIIOOOI
OIIIOOOOIIOIOOOIIIIIIIOOIOIIOIIIIIIIOOOIOOIIIOOOOOOIOIOIIOOOIOIIIIOIIOOOOIIOIIIOIIIOOIOIIIIOIIOOIIIIIOIIIOO
OOIIIOOOOIOIIOIIOOIIIOIIOIIOIOOOOIOOOIOOIIIIIOIIOOIOOIIOIOOOOOOIIIIIIOOOIOOIIIIIIIOIOOIIIOIIOOOOIIIOIIIIIII
OIIIOIOIOOIIOOIOOIIOOIIIOOIOOOOOOIOOIIOOOOIOIOOOOIOOOOOOOIIIOOIIIOIOIIOIIOOOIOOOOOIIIIOOOIOIOOO
IIOOOIIOOIIOIOIIIOOOOIOOOOOIIIIOOIOIIOIIIIIIIOOOOIIIIIIIOOOIOIOIOIOOOIOOIIIOOOOOOOOIIOIIIIIIOIOIIIIOIOO
OOOOOIOIOIOIOOOOOIIOOOIIOOOOOOOIOOIOIOIIOOIIIOOOOOOOOIOIIIIOIOIIIOOIOIIIOOOIIIIIIIIIIOOOOOIIOOIIO
OOOOIOOOOIIOOIOOIOOOOOIOOOIOOOOOOOOOOIIOOOOOIIIIOOIOOOIIOOIIOIOOIOOO
1000 1002 4636 2294 2342 1.0209240913391113

The format of the data is as follows:
(1) The bit length of randomly generated extremely large integer,
(2) The bit length of integer in the obit
(3) The total count of ‘I’ and ‘O’ in Collatz computation sequence (dynamics)

(4) The count of ‘I’ in computation sequence
(5) The count of ‘O’ in computation sequence
(6) The ratio of the count of ‘O’ over the count of ‘I’

More data files are named and provided as follows:

txpo46dynamics10000
txpo46dynamics100000,
txpo46dynamics500000,
txpo46dynamics1000000,
txpo46dynamics2000000,
txpo46dynamics3000000,
txpo46dynamics4000000,
txpo46dynamics5000000,
txpo46dynamics6000000.

The bit length of randomly generated extremely large integer is 10000, 100000, 500000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, respectively.

Data S3. (txpo48.c)

```
 ////////////////////////////////////////////////////////////////////////////
//Copyright 2022, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: This program is change ASCII data into binary data
//         Read ASCII data file, and then each 8 characters will change into one byte
//Input: data file with ASCII as 0 and 1
//Output: data file with binary in which each byte is 8 characters
//Usage Example: txpo48.exe ASCII_file the_number_of_byte
//              txpo48.exe 6000000 3613924
//Output Example:
////////////////////////////////////////////////////////////////
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int main(int argc, char *argv[])
{

	FILE *fp_read;
	FILE *fp_write;

	char c1,c2,c3,c4,c5,c6,c7,c8;
	unsigned int d1,d2,d3,d4,d5,d6,d7,d8;
	unsigned int binary;
	int cnt,length_of_byte;

	char filename_r[40];
	char filename_w[40];

	strcpy(filename_r,argv[1]);
	strcpy(filename_w,filename_r);
	strcat(filename_w,"bin");

	//printf("debug: reading file is %s, writting file is %s.\n", filename_r, filename_w);


	if( !(fp_read = fopen(filename_r, "r")))
	{
		printf("Cannot open the file!\n");
		exit(0);
	}
```

```
if( !(fp_write = fopen(filename_w, "wb+")))
{
        printf("Cannot open the file!\n");
        exit(0);
}

cnt=1;
length_of_byte = atoi(argv[2]);
//printf("debug:length of byte is %d.\n", length_of_byte);

while(cnt<=length_of_byte)
{
        //printf("debug:cnt=%d.\n",cnt);

        c1=fgetc(fp_read);  //read 8 characters
        c2=fgetc(fp_read);
        c3=fgetc(fp_read);
        c4=fgetc(fp_read);
        c5=fgetc(fp_read);
        c6=fgetc(fp_read);
        c7=fgetc(fp_read);
        c8=fgetc(fp_read);
        //printf("debug: reading 8 characters are: %c%c%c%c%c%c%c%c.\n", c1,c2,c3,c4,c5,c6,c7,c8);

        if(c1=='0')
                d1=0;
        else
                d1=1;

        if(c2=='0')
                d2=0;
        else
                d2=1;
```

```
if(c3=='0')
        d3=0;
else
        d3=1;

if(c4=='0')
        d4=0;
else
        d4=1;

if(c5=='0')
        d5=0;
else
        d5=1;

if(c6=='0')
        d6=0;
else
        d6=1;

if(c7=='0')
        d7=0;
else
        d7=1;

if(c8=='0')
        d8=0;
else
        d8=1;

//printf("debug: 8 bits to be written are: %d%d%d%d%d%d%d%d.\n", d1,d2,d3,d4,d5,d6,d7,d8);
```

```
                binary = (int)(d1*pow(2,7)+d2*pow(2,6)+d3*pow(2,5)+d4*pow(2,4)+d5*pow(2,3)+d6*pow(2,2)+d7*2+d8);
                //printf("debug:computed binary=%x.\n",binary);

                fputc(binary,fp_write);
                //printf("debug:writing binary=%x, cnt=%d.\n",binary,cnt);

                cnt=cnt+1;

        }

        fclose(fp_read);
        fclose(fp_write);

        return 1;

}
```

Data S4. (console.txt, Abbreviate)

```
Lenovo@LAPTOP-I3CADE4P ~/sts-2.1.2
$ ./assess.exe 289113
        G E N E R A T O R   S E L E C T I O N
        ---------------------------------------

   [0] Input File              [1] Linear Congruential
   [2] Quadratic Congruential I   [3] Quadratic Congruential II
   [4] Cubic Congruential         [5] XOR
   [6] Modular Exponentiation     [7] Blum-Blum-Shub
   [8] Micali-Schnorr             [9] G Using SHA-1

  Enter Choice: 0
```

```
        User Prescribed Input File: 6000000

        S T A T I S T I C A L   T E S T S
        _________________________________

 [01] Frequency                    [02] Block Frequency
 [03] Cumulative Sums              [04] Runs
 [05] Longest Run of Ones          [06] Rank
 [07] Discrete Fourier Transform   [08] Nonperiodic Template Matchings
 [09] Overlapping Template Matchings  [10] Universal Statistical
 [11] Approximate Entropy          [12] Random Excursions
 [13] Random Excursions Variant    [14] Serial
 [15] Linear Complexity

     INSTRUCTIONS
       Enter 0 if you DO NOT want to apply all of the
       statistical tests to each sequence and 1 if you DO.

Enter Choice: 1

    P a r a m e t e r   A d j u s t m e n t s
    ----------------------------------------
 [1] Block Frequency Test - block length(M):         128
 [2] NonOverlapping Template Test - block length(m): 9
 [3] Overlapping Template Test - block length(m):    9
 [4] Approximate Entropy Test - block length(m):     10
 [5] Serial Test - block length(m):                  16
 [6] Linear Complexity Test - block length(M):       500

Select Test (0 to continue): 0
```

```
   How many bitstreams? 100

   Input File Format:
    [0] ASCII - A sequence of ASCII 0's and 1's
    [1] Binary - Each byte in data file contains 8 bits of data

   Select input mode:  0

     Statistical Testing In Progress.....

     Statistical Testing Complete!!!!!!!!!!!


Lenovo@LAPTOP-I3CADE4P ~/sts-2.1.2
$ ./assess.exe 1032549
          G E N E R A T O R   S E L E C T I O N
          ____________________________________

    [0] Input File                [1] Linear Congruential
    [2] Quadratic Congruential I  [3] Quadratic Congruential II
    [4] Cubic Congruential        [5] XOR
    [6] Modular Exponentiation    [7] Blum-Blum-Shub
    [8] Micali-Schnorr            [9] G Using SHA-1

   Enter Choice: 0


            User Prescribed Input File: 6000000

            S T A T I S T I C A L   T E S T S
            _________________________________

    [01] Frequency                [02] Block Frequency
```

```
 [03] Cumulative Sums                [04] Runs
 [05] Longest Run of Ones            [06] Rank
 [07] Discrete Fourier Transform     [08] Nonperiodic Template Matchings
 [09] Overlapping Template Matchings [10] Universal Statistical
 [11] Approximate Entropy            [12] Random Excursions
 [13] Random Excursions Variant      [14] Serial
 [15] Linear Complexity

      INSTRUCTIONS
        Enter 0 if you DO NOT want to apply all of the
        statistical tests to each sequence and 1 if you DO.

Enter Choice: 1

     P a r a m e t e r   A d j u s t m e n t s
     ----------------------------------------
 [1] Block Frequency Test - block length(M):         128
 [2] NonOverlapping Template Test - block length(m): 9
 [3] Overlapping Template Test - block length(m):    9
 [4] Approximate Entropy Test - block length(m):     10
 [5] Serial Test - block length(m):                  16
 [6] Linear Complexity Test - block length(M):       500

Select Test (0 to continue): 0

How many bitstreams? 28

Input File Format:
 [0] ASCII - A sequence of ASCII 0's and 1's
 [1] Binary - Each byte in data file contains 8 bits of data

Select input mode:  0
```

```
    Statistical Testing In Progress.....

    Statistical Testing Complete!!!!!!!!!!!!
```

Data S5. (finalAnalysisReport.txt in directory "2000000-96449,100")

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
  generator is <2000000>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 11  16  14   7  10   8   7   7   9  11  0.474986     98/100    Frequency
 13  10  10  12  12  10   8   7  12   6  0.834308     98/100     BlockFrequency
 15   5  16  12  12   9   5   4   6  16  0.013569     98/100    CumulativeSums
 11  14  14  15   7   6  12   4   7  10  0.153763     99/100     CumulativeSums
  9  14  15   8  10   6  15   7  10   6  0.262249    100/100     Runs
 11   9  12   9  12   6  13   6   8  14  0.616305     98/100    LongestRun
 13  12  14  25   7   6   5   6   7   5  0.000051 *  100/100     Rank
 10  16   9  11   6   6  16   6  11   9  0.191687    100/100     FFT
  5  13   8   6   8  13   8  17  11  11  0.202268    100/100     NonOverlappingTemplate
 11  15  11   9  10   8  10  10  12   4  0.616305     97/100     NonOverlappingTemplate
 10  14   6  13   7   8  15  13   8   6  0.289667     98/100    NonOverlappingTemplate
  6  12  14  12   9  10   7   8  17   5  0.171867     99/100    NonOverlappingTemplate
  7   7   9  14  13   9  15   8   9   9  0.574903    100/100    NonOverlappingTemplate
 14  11  17   6   4  12   9   6   9  12  0.108791     98/100    NonOverlappingTemplate
  9  13  11  11   8  10  13   7  10   8  0.924076    100/100     NonOverlappingTemplate
  5   9  14  10  10   8   6  13  11  14  0.455937     99/100     NonOverlappingTemplate
  8   7  11  11  10   6  14  16  12   5  0.262249     99/100     NonOverlappingTemplate
 12  11  10  11   6  10  10   7  10  13  0.911413     99/100     NonOverlappingTemplate
 11   8  22   9  12  11   3  12   7   5  0.003996    100/100     NonOverlappingTemplate
```

10	9	9	9	14	7	13	7	11	11	0.851383	98/100	NonOverlappingTemplate
11	9	9	10	3	16	7	15	10	10	0.202268	98/100	NonOverlappingTemplate
13	10	17	9	9	5	7	9	9	12	0.350485	99/100	NonOverlappingTemplate
8	10	10	13	7	11	9	6	12	14	0.739918	98/100	NonOverlappingTemplate
11	10	11	13	6	10	9	11	10	9	0.964295	99/100	NonOverlappingTemplate
9	9	8	9	13	8	7	13	15	9	0.699313	99/100	NonOverlappingTemplate
9	7	14	13	10	9	10	11	7	10	0.867692	99/100	NonOverlappingTemplate
8	15	9	10	7	11	10	10	9	11	0.897763	99/100	NonOverlappingTemplate
14	8	8	14	11	8	11	9	9	8	0.816537	98/100	NonOverlappingTemplate
12	8	9	8	12	7	16	5	10	13	0.383827	100/100	NonOverlappingTemplate
12	11	5	7	16	5	12	11	13	8	0.224821	99/100	NonOverlappingTemplate
3	15	10	10	7	13	14	7	8	13	0.162606	99/100	NonOverlappingTemplate
9	3	8	13	11	12	10	11	13	10	0.554420	100/100	NonOverlappingTemplate
10	8	11	8	12	8	6	9	19	9	0.236810	99/100	NonOverlappingTemplate
8	12	9	7	10	8	8	14	17	7	0.350485	99/100	NonOverlappingTemplate
9	8	7	12	12	11	6	14	15	6	0.383827	99/100	NonOverlappingTemplate
10	11	9	6	8	13	17	9	7	10	0.437274	99/100	NonOverlappingTemplate
11	8	13	8	12	9	7	8	7	17	0.401199	100/100	NonOverlappingTemplate
6	7	13	6	8	13	17	13	8	9	0.181557	97/100	NonOverlappingTemplate
9	15	6	7	7	9	17	10	14	6	0.115387	98/100	NonOverlappingTemplate
7	12	13	11	6	9	12	9	14	7	0.637119	100/100	NonOverlappingTemplate
7	7	9	9	14	12	7	18	11	6	0.162606	100/100	NonOverlappingTemplate
10	13	15	5	11	12	13	5	5	11	0.191687	100/100	NonOverlappingTemplate
7	13	16	9	10	10	6	11	6	12	0.419021	98/100	NonOverlappingTemplate
8	10	6	7	16	6	10	14	11	12	0.334538	100/100	NonOverlappingTemplate
15	9	8	14	8	6	12	4	13	11	0.236810	99/100	NonOverlappingTemplate
14	7	10	8	10	13	9	8	17	4	0.171867	98/100	NonOverlappingTemplate
13	11	6	10	13	4	8	11	9	15	0.334538	99/100	NonOverlappingTemplate
14	6	8	10	7	10	10	11	16	8	0.474986	99/100	NonOverlappingTemplate
13	9	5	15	15	7	8	6	11	11	0.236810	100/100	NonOverlappingTemplate
6	6	11	9	13	10	11	12	11	11	0.834308	99/100	NonOverlappingTemplate
16	9	10	9	5	7	15	9	10	10	0.366918	99/100	NonOverlappingTemplate
8	13	14	8	8	12	7	10	8	12	0.759756	98/100	NonOverlappingTemplate

12	5	6	12	16	4	11	13	15	6	0.045675	98/100	NonOverlappingTemplate
13	7	5	14	8	14	9	14	10	6	0.262249	98/100	NonOverlappingTemplate
9	13	6	10	12	6	6	11	17	10	0.262249	99/100	NonOverlappingTemplate
10	11	11	11	5	9	11	9	15	8	0.739918	100/100	NonOverlappingTemplate
11	9	12	14	14	8	8	4	7	13	0.350485	98/100	NonOverlappingTemplate
12	5	13	9	12	5	5	17	11	11	0.108791	100/100	NonOverlappingTemplate
15	7	6	8	9	12	5	10	13	15	0.224821	100/100	NonOverlappingTemplate
16	9	10	8	8	11	7	12	8	11	0.699313	96/100	NonOverlappingTemplate
5	8	10	14	9	14	7	9	13	11	0.514124	100/100	NonOverlappingTemplate
13	8	9	17	8	4	13	7	11	10	0.202268	99/100	NonOverlappingTemplate
17	10	3	8	7	15	9	9	8	14	0.071177	99/100	NonOverlappingTemplate
13	5	11	9	12	9	10	9	10	12	0.867692	100/100	NonOverlappingTemplate
10	12	12	10	12	11	6	7	7	13	0.779188	100/100	NonOverlappingTemplate
10	5	11	8	8	13	13	9	11	12	0.759756	100/100	NonOverlappingTemplate
13	16	5	10	12	10	9	6	12	7	0.319084	100/100	NonOverlappingTemplate
8	9	11	18	11	11	9	5	9	9	0.350485	100/100	NonOverlappingTemplate
12	9	7	12	11	14	11	8	12	4	0.534146	100/100	NonOverlappingTemplate
8	5	8	10	8	10	16	11	9	15	0.350485	98/100	NonOverlappingTemplate
13	6	9	12	7	6	12	10	10	15	0.494392	97/100	NonOverlappingTemplate
13	7	9	7	11	19	9	7	9	9	0.202268	98/100	NonOverlappingTemplate
15	10	9	13	14	7	8	8	5	11	0.401199	97/100	NonOverlappingTemplate
7	12	9	12	13	11	9	11	6	10	0.867692	99/100	NonOverlappingTemplate
7	11	10	12	6	12	11	11	11	9	0.924076	100/100	NonOverlappingTemplate
12	7	9	7	10	15	13	10	8	9	0.719747	95/100	* NonOverlappingTemplate
9	8	9	8	10	11	10	7	13	15	0.798139	99/100	NonOverlappingTemplate
9	9	6	16	10	18	6	8	9	9	0.122325	99/100	NonOverlappingTemplate
7	6	13	12	6	12	13	16	6	9	0.213309	100/100	NonOverlappingTemplate
15	4	13	11	9	10	8	8	12	10	0.494392	99/100	NonOverlappingTemplate
11	12	6	11	14	11	10	7	9	9	0.834308	100/100	NonOverlappingTemplate
9	15	3	11	17	15	7	7	6	10	0.030806	100/100	NonOverlappingTemplate
5	12	9	6	7	14	8	18	11	10	0.122325	100/100	NonOverlappingTemplate
6	6	9	10	17	10	9	13	10	10	0.419021	100/100	NonOverlappingTemplate
9	9	9	8	14	11	7	12	8	13	0.834308	98/100	NonOverlappingTemplate

8	10	15	5	9	12	10	7	18	6	0.096578	99/100	NonOverlappingTemplate
8	11	11	5	9	9	12	13	15	7	0.534146	99/100	NonOverlappingTemplate
11	8	5	11	11	10	7	12	14	11	0.719747	99/100	NonOverlappingTemplate
12	12	6	12	7	6	20	9	13	3	0.011791	99/100	NonOverlappingTemplate
7	9	4	7	13	14	9	12	10	15	0.275709	100/100	NonOverlappingTemplate
8	4	8	13	11	9	9	8	17	13	0.224821	100/100	NonOverlappingTemplate
18	13	9	11	8	5	15	6	6	9	0.062821	98/100	NonOverlappingTemplate
9	11	7	11	11	6	8	16	8	13	0.514124	99/100	NonOverlappingTemplate
9	15	8	5	16	8	11	10	10	8	0.350485	100/100	NonOverlappingTemplate
12	10	10	8	8	12	8	11	12	9	0.978072	97/100	NonOverlappingTemplate
6	9	13	9	12	10	15	7	11	8	0.637119	100/100	NonOverlappingTemplate
11	8	9	11	17	15	12	6	8	3	0.080519	99/100	NonOverlappingTemplate
13	9	8	12	8	11	14	10	7	8	0.816537	98/100	NonOverlappingTemplate
11	9	4	14	10	7	12	8	14	11	0.455937	99/100	NonOverlappingTemplate
11	8	10	11	11	10	12	10	8	9	0.996335	98/100	NonOverlappingTemplate
13	6	4	8	5	10	17	12	11	14	0.066882	99/100	NonOverlappingTemplate
11	12	8	12	11	10	6	12	11	7	0.883171	98/100	NonOverlappingTemplate
13	8	8	15	10	9	10	9	8	10	0.851383	100/100	NonOverlappingTemplate
10	11	14	7	11	12	8	9	9	9	0.924076	99/100	NonOverlappingTemplate
10	9	13	4	13	8	6	11	9	17	0.181557	97/100	NonOverlappingTemplate
10	4	13	10	9	11	18	10	9	6	0.171867	99/100	NonOverlappingTemplate
6	10	9	11	11	10	8	10	14	11	0.911413	100/100	NonOverlappingTemplate
8	10	11	7	5	11	15	9	10	14	0.514124	98/100	NonOverlappingTemplate
6	11	13	10	8	9	10	13	12	8	0.851383	100/100	NonOverlappingTemplate
12	8	5	9	16	8	14	11	8	9	0.383827	98/100	NonOverlappingTemplate
9	10	10	11	12	10	12	9	10	7	0.991468	99/100	NonOverlappingTemplate
9	6	14	12	10	10	13	9	10	7	0.779188	98/100	NonOverlappingTemplate
9	5	8	11	8	9	13	16	9	12	0.474986	99/100	NonOverlappingTemplate
6	5	9	10	13	12	8	14	9	14	0.419021	100/100	NonOverlappingTemplate
10	11	10	18	12	9	8	6	9	7	0.350485	99/100	NonOverlappingTemplate
7	15	12	3	7	11	16	11	7	11	0.108791	99/100	NonOverlappingTemplate
7	13	14	9	11	7	15	11	2	11	0.137282	99/100	NonOverlappingTemplate
9	5	12	11	13	13	10	9	7	11	0.739918	100/100	NonOverlappingTemplate

12	14	10	6	6	9	9	5	11	18	0.108791	99/100		NonOverlappingTemplate
8	9	11	6	17	5	11	13	15	5	0.075719	98/100		NonOverlappingTemplate
13	7	7	10	13	10	7	17	7	9	0.319084	96/100		NonOverlappingTemplate
10	10	9	11	12	7	16	9	6	10	0.657933	99/100		NonOverlappingTemplate
7	6	9	17	8	9	11	14	6	13	0.202268	99/100		NonOverlappingTemplate
12	6	11	7	14	7	6	9	18	10	0.137282	99/100		NonOverlappingTemplate
9	17	7	7	10	12	7	12	6	13	0.275709	100/100		NonOverlappingTemplate
13	13	13	8	8	8	9	9	9	10	0.897763	97/100		NonOverlappingTemplate
7	9	14	13	6	8	11	10	8	14	0.574903	99/100		NonOverlappingTemplate
9	12	10	5	9	8	12	4	21	10	0.020548	99/100		NonOverlappingTemplate
11	12	10	14	9	14	11	7	9	3	0.366918	100/100		NonOverlappingTemplate
12	6	9	10	9	10	9	14	9	12	0.883171	96/100		NonOverlappingTemplate
10	5	11	17	15	7	13	8	6	8	0.115387	99/100		NonOverlappingTemplate
3	13	9	14	8	9	10	14	9	11	0.366918	100/100		NonOverlappingTemplate
13	9	8	10	10	8	12	12	11	7	0.935716	99/100		NonOverlappingTemplate
11	9	4	11	8	10	10	14	8	15	0.455937	95/100	*	NonOverlappingTemplate
8	13	5	12	11	10	10	14	10	7	0.657933	99/100		NonOverlappingTemplate
6	9	12	11	12	7	6	9	8	20	0.075719	99/100		NonOverlappingTemplate
10	12	11	5	12	13	7	10	12	8	0.739918	99/100		NonOverlappingTemplate
4	13	11	7	13	12	7	14	6	13	0.224821	99/100		NonOverlappingTemplate
6	9	5	17	6	9	10	11	11	16	0.102526	98/100		NonOverlappingTemplate
9	5	10	15	12	8	13	8	10	10	0.616305	99/100		NonOverlappingTemplate
8	9	8	6	10	11	17	11	11	9	0.554420	98/100		NonOverlappingTemplate
10	12	4	15	9	14	7	7	10	12	0.319084	100/100		NonOverlappingTemplate
11	5	6	12	14	6	14	10	12	10	0.366918	98/100		NonOverlappingTemplate
11	10	10	17	9	13	8	8	8	6	0.455937	100/100		NonOverlappingTemplate
13	8	6	5	10	12	7	21	11	7	0.019188	100/100		NonOverlappingTemplate
6	13	14	9	13	9	7	11	9	9	0.699313	100/100		NonOverlappingTemplate
11	6	15	15	10	9	11	8	11	4	0.275709	99/100		NonOverlappingTemplate
11	8	13	7	8	14	9	11	10	9	0.867692	98/100		NonOverlappingTemplate
10	11	6	13	9	10	10	9	9	13	0.924076	98/100		NonOverlappingTemplate
7	7	13	10	6	12	8	12	17	8	0.289667	99/100		NonOverlappingTemplate
14	9	9	12	10	8	6	10	11	11	0.883171	100/100		NonOverlappingTemplate

11	12	6	11	8	8	8	13	8	15	0.616305	99/100	NonOverlappingTemplate
10	8	4	10	12	8	10	13	12	13	0.637119	99/100	NonOverlappingTemplate
5	12	8	10	11	10	13	10	12	9	0.851383	100/100	NonOverlappingTemplate
8	13	8	7	11	17	9	10	10	7	0.474986	100/100	NonOverlappingTemplate
9	15	4	10	17	15	7	7	6	10	0.048716	100/100	NonOverlappingTemplate
9	10	8	13	9	14	8	9	7	13	0.798139	98/100	OverlappingTemplate
0	0	0	0	0	0	100	0	0	0	0.000000 *	100/100	Universal
16	12	13	7	9	8	12	9	8	6	0.455937	100/100	ApproximateEntropy
0	2	1	1	2	0	1	0	0	1	----	8/8	RandomExcursions
0	1	0	3	1	2	1	0	0	0	----	8/8	RandomExcursions
1	1	0	1	1	2	1	0	1	0	----	8/8	RandomExcursions
1	0	0	1	2	1	1	2	0	0	----	8/8	RandomExcursions
2	1	0	1	3	0	0	0	0	1	----	8/8	RandomExcursions
1	1	0	0	2	1	1	0	0	2	----	8/8	RandomExcursions
1	0	1	0	3	0	0	0	1	2	----	8/8	RandomExcursions
0	1	2	2	1	0	0	0	0	2	----	8/8	RandomExcursions
0	1	1	1	1	0	1	0	1	2	----	8/8	RandomExcursionsVariant
0	2	0	0	2	1	0	1	1	1	----	8/8	RandomExcursionsVariant
0	1	2	1	0	0	1	1	0	2	----	8/8	RandomExcursionsVariant
1	1	1	1	0	1	1	0	2	0	----	8/8	RandomExcursionsVariant
2	1	2	0	1	0	1	0	1	0	----	8/8	RandomExcursionsVariant
2	2	2	0	0	0	1	0	1	0	----	8/8	RandomExcursionsVariant
2	1	1	1	0	2	0	0	0	1	----	8/8	RandomExcursionsVariant
2	1	2	0	0	1	1	0	1	0	----	8/8	RandomExcursionsVariant
1	2	1	0	1	2	0	0	1	0	----	8/8	RandomExcursionsVariant
2	1	0	1	1	1	0	0	1	1	----	8/8	RandomExcursionsVariant
1	1	1	0	0	1	2	1	0	1	----	8/8	RandomExcursionsVariant
1	1	0	0	1	0	1	0	3	1	----	8/8	RandomExcursionsVariant
1	1	0	0	1	2	0	2	1	0	----	8/8	RandomExcursionsVariant
1	1	0	1	0	2	1	0	0	2	----	8/8	RandomExcursionsVariant
1	1	1	3	0	1	0	0	1	0	----	8/8	RandomExcursionsVariant
2	1	3	0	0	1	0	0	1	0	----	8/8	RandomExcursionsVariant
1	3	0	2	0	0	0	0	1	1	----	8/8	RandomExcursionsVariant

```
  1   1   3   0   0   0   0   2   0   1    ----       8/8      RandomExcursionsVariant
 16   6   9  12   6  13   7  14   6  11  0.191687    97/100     Serial
 15  13  15   6   4  12   8  13   6   8  0.096578    98/100     Serial
 11   9  12   7   5  12  14  15   3  12  0.129620    98/100     LinearComplexity


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
The minimum pass rate for each statistical test with the exception of the
random excursion (variant) test is approximately = 96 for a
sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test
is approximately = 7 for a sample size = 8 binary sequences.

For further guidelines construct a probability table using the MAPLE program
provided in the addendum section of the documentation.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Data S6. (500000bin_result.txt)

```
Count Of Bits: 2420800, 0_num: 1208908, 1_num: 1211892
PASS  p_value = 0.055127  统计值 = 1.917871  耗时:  3.000ms --Frequency Test
PASS  p_value = 0.277998  统计值 = 24337.120000  耗时:  0.000ms --BlockFrequency100 Test
PASS  p_value = 0.315837  统计值 = 17.050787  耗时:  12.000ms --Poker4 Test
PASS  p_value = 0.911713  统计值 = 225.066596  耗时:  17.000ms --Poker8 Test
PASS  p_value1 = 0.099875  统计值 = 4.607673  耗时:  62.000ms --Serial2_1 Test
PASS  p_value2 = 0.335007  统计值 = 0.929445  耗时:  62.000ms --Serial2_2 Test
PASS  p_value1 = 0.675223  统计值 = 12.966266  耗时:  139.000ms --Serial5_1 Test
PASS  p_value2 = 0.608453  统计值 = 6.346742  耗时:  139.000ms --Serial5_2 Test
PASS  p_value = 0.333823  统计值 = 0.683378  耗时:  10.000ms --Runs Test
PASS  p_value = 0.342841  统计值 = 32.540162  耗时:  196.000ms --RunsDistribution Test
```

```
PASS  p_value = 0.434410  统计值 = 5.900663  耗时:  13.000ms --LongestRunOfOnes Test
PASS  p_value = 0.399450  统计值 = 0.842604  耗时:  2.000ms --BinaryDerivate3 Test
PASS  p_value = 0.704059  统计值 = 0.379847  耗时:  2.000ms --BinaryDerivate7 Test
PASS  p_value = 0.335329  统计值 = 0.963435  耗时:  0.000ms --SelfCorrelation1 Test
PASS  p_value = 0.452836  统计值 = -0.750695  耗时:  1.000ms --SelfCorrelation2 Test
PASS  p_value = 0.517907  统计值 = -0.646575  耗时:  1.000ms --SelfCorrelation8 Test
PASS  p_value = 0.050413  统计值 = 1.956440  耗时:  1.000ms --SelfCorrelation16 Test
PASS  p_value = 0.795107  统计值 = 0.458558  耗时:  50.000ms --Rank Test
PASS  p_value = 0.064877  耗时:  7.000ms --CumulativeSums（forward） Test
PASS  p_value = 0.099026  耗时:  7.000ms --CumulativeSums（backward） Test
PASS  p_value = 0.269000  统计值 = 5.183320  耗时:  38.000ms --ApproximateEntropy2 Test
PASS  p_value = 0.601655  统计值 = 29.343855  耗时:  75.000ms --ApproximateEntropy5 Test
PASS  p_value = 0.897105  统计值 = 2.232668  耗时:  2025.000ms --LinearComplexity Test
PASS  p_value = 0.329170  统计值 = 0.975786  耗时:  91.000ms --Universal Test
PASS  p_value = 0.055258  统计值 = 1.916845  耗时:  367.000ms --DiscreteFourierTransform Test
```

Data S7. (txpo47.c for RBSG)

```
////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2023, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: RBSG via the dynamics of an extremely large integer with binary length related to UNIT.
//Input:
//Output: Random Bit Sequence (Original Dynamics/Collatz computation sequence in 1/0)
//Usage Example: txpo47.exe
//Output Example:
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
//#include <time.h>   //if to generate different random integer in each run, enable it

static int MAXLEN=6000000; //it is used for setting the max length in malloc
static int UNIT=1000;     //it is used for setting the length of staring integer
static int RUNTIMES=1;     //it is used for setting the count of run

//Three X Plus One subfunction, result <= 3*a + 1
int TXPO(char* a, char* result);

int main(int argc, char *argv[]) {

    char* current_x;    //the current integer
    char* temp_x;       //temporary integer in computing dynamics, i.e., obit

    char* dynamics;    //dynamics in terms of `1' and `0'

    FILE *fp_dynamics;  //a file to store dynamics

    int len;
    int i,j;                //for loop

    int start_x_len=0;       //the length of the starting integer
    char filename[40];       //filename of the generated file recording rbs
    char extendname[20];     //filename extension to record start_x_len

        current_x = (char *)malloc(MAXLEN*5*sizeof(char));
        temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
        dynamics = (char *)malloc(MAXLEN*10*sizeof(char));

        if((current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
        {
                printf("debug: error of malloc.\n");
```

```
}

for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
{
start_x_len = UNIT*(i+1);  //e.g., if UNIT=1000, RUNTIMES=5, then start_x_len =1000,2000,3000,4000,5000

strcpy(filename,"txpo47RBSG");  //prepare dynamics file name, the name is
itoa(start_x_len,extendname,10);    //change maxlen to a string - extendname
strcat(filename,extendname);        //append maxlen to the file name

fp_dynamics = fopen(filename,"w");

memset(current_x,0,sizeof(current_x));
memset(temp_x,0,sizeof(temp_x));
memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics

//Generate a random interger with the bit length of maxlen as a starting integer
//srand(time(NULL));        //for the random seed generation. if enabled, random integer in each run is different
strcat(current_x,"1");      //the MSB should be 1
for(j=0; j < start_x_len-2; j++)     //2 bits are fixed, MSB and LSB
{
        if(rand()%2==1)        //rand() returns an random integer
                strcat(current_x,"1");
        else
                strcat(current_x,"0");
}
strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
//strcpy(current_x, argv[1]);   //seed could also be an inputting string

fprintf(fp_dynamics, "seed=%s\n", current_x);  //writing start_x into data file

while(strlen(current_x)!=1)        //if outputting is original dynamics instead of reduced dynamics
{
```

```
        len = strlen(current_x);
        if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
        {
        TXPO(current_x,temp_x);    //call subfunction TXPO(), i.e., temp_x <= current_x*3+1
        strcpy(current_x,temp_x);  //current_x <= temp_x;

        len = strlen(current_x);   //note that, current_x is changed, so hereby need to get new length
        current_x[len-1] = '\0';   //cut down the last bit that is always 0, it means current_x <= current_x/2

        strcat(dynamics,"1");     //'1' that means (3x+1)/2, is appended into dynamics

    }
    else                    //current_x is even
    {
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

        strcat(dynamics,"0");     //'0' that means x/2, is appended into dynamics

    }
}       //end of while loop


        fprintf(fp_dynamics, "RBSG=%s\n", dynamics);  //writing dynamics into data file

        fclose(fp_dynamics);

        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }         //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        free(temp_x);
        free(dynamics);
```

```
    return 1;
}


int TXPO(char* a, char* txpo_result) //this subfunction is compute 3x+1, that is, txpo_result <= 3*a + 1
{

    //recall that, `10' or `1' is appended by "a||0" is txpo_result

    //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
    int c = 1;        //only current carrier needs to be stored.

    int i;        //counter in for loop

    int n = strlen(a);          //the bit length of a, a is the inputting string, to compute 3*a+1
    //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);

    memset(txpo_result, 0, sizeof(txpo_result));
    //printf("debug: txpo_result=%s, strlen(txpo_result)=%d\n", txpo_result, strlen(txpo_result));

    //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
    for (i=n-2; i>=0; i--)
    {
                    if(a[i+1]=='0' && a[i]=='0' && c==0)
                            { c=0; a[i+1]='0'; continue;}
                    if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                            { c=0; a[i+1]='1'; continue;}
                    if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
                            { c=1; a[i+1]='0'; continue;}
                    if(a[i+1]=='1' && a[i]=='1' && c==1)
                            { c=1; a[i+1]='1'; continue;}
```

```
      //printf("debug: in TXPO's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
   }

   if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
      { a[0]='0'; strcat(txpo_result,"1"); }

   strcat(txpo_result, a);    //the other bits are appended to right
   strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even

   return 1;

}
```

Data S8. (txpo47RBSG1000)

seed=1110010000011111110101001001001101010111011011011101001111110010000000001010001101100000010010110001111
1000101011000111100010111010001000111111111110100000100101010101110010000101001011000011010111011010110110011
0001101111110100000001101100000101011001000100001110001001111001100011101111010100110010110100110110100111110
1111011110010010010101111100011010001000111010010110001101000011010110000001101101101001001101111010111011111
0000001010001110011000101100001001100010010000101011001001110111010001011110000001111100001010101001110011010
1011100010101010001100010111110010101111111001100000110111111010100111111000110011101001010010111100110001010
0110001001110010110110100011010111100110111111010111110010100101000110111010110001110000111100100101110001011
1010100011000101111101110110111111000010000011000101101011001110010011111001001100000011010011100100000001111
0111100000110000101010001110000001101011011001000111010111110011010010100111111100101111010000100001111100101
1001101011000011011111010100110011
RBSG=1011001011010000111110101100000101000101111110000100101111111110010000011000110110110111110100010011000
0111101110001001011110111000001011101001000100101000111001011001111100111111100000110010011001101000000100000
0100110100001010110010101001000111101100010110111101000101011010000000111111011001000110111100110001111101100
0100110000001001010011110010011000010110011010000101001010001101011000010111110001101011110101111110001100100
1
0101110011001111000010111100111011101111001100111011001111111100101100111001001001000101111100000110000000000
```
```

011111001111010100110001010010101110011101111111101010000100110110110101010101110011000110101100100000101111
001010110110100111010000110011110101010011110110110011100100110101011000110010011001001001010111001100111011
101001011100001011100111101110000000011100011011011110010100011111001011001111000110101011111111000100110001
001110101101100011001110001111001001011000110011010101101010010011100011010011010111110000111010110000101101
110000010111100111011110011011000110010100001101111001000111001010110011101101001110100000100001000001001110
001010110101100000010001110001110011111111010001010000000100000011000011101011100110111111100111111000001110
010000010100111000110001111110011101010110000000101010010110100101011101000110110001111010011010000000001001
010010001100110100101110110000100010001100010101110101001110001100010000110111110111011111001001111010000100
001010110100101100110100110001100100010000110111011110000111101111100000001110001011001001011010110011111011
101110001000001100110000111010101100001000010010000110010010001101110101010101110110000100110010000100111010
001110000011010010010101011011001010010101010100100010000001001100010101101100001000110110001001000011111111
000010000101000110101101111001001101110000100001111001000111011110001011101110111101010100001000100111101101
111000000011010100001101001110100111111111101001100100011010110000111001111111011001001111000100110101011101
111100100010111111101000111010101001000100101100001100111000011110110010101011000001100000100001101010010011
000001011010000011111111101111111100110011101001010000010011100011000000101011110101101110111010111001111101
010100110000000010011100010101100101100111010000000101111011010011100101111011101011010101000011101111101100001
111110011111000111101100111000011101101100101101100110100101110111110000010011110110011000111110010011111000
001001111110010000010100000110011100001000100000111101000111101001001111100110100000110011110000011110000101
010100001101100011001101101100011001001100110101101010110111110001010101111110001111100010010100100011011101
101011000011101100011001000000101000000101011111000111100111100001010010001111000110001111100000000000111100
100101100100001100100110001000010011110010110101100011001011101111111000101111001101000011001011011111110001111
011101101011100100000101110001000001011101110001000100111100000001011000100001011100100110011111111100000011101
111101000101001000011111001111101110101010010000100010101111010001100110010101100101111001010001000101110111
111111010010100101000000110111111000000001011011111000011010110100010011111111011111111001000011101100001010111
110010101100111001000011100110111000110011010110011110010100001000010001111011001010011111101010100001001111
000100000100111000000110111000111000000000101111011110010110001000010011001101110100010001110110111000010011O
001101000010110101000011011101101000000110010011110001001110001100000010111011000101111101011000000110001000111
000110111111101101111111101101010110011101111011010111001010100001010110110111101010010100000011010101001111111
111010010110001001011001001111100100000100010010111000111011000101000010101110001110011000101001010011100010
111000010011010110110000110111001001110001010000000101111100010100101101001110001001000100010101010001000
100110111101001100100101111010001010101111111011000111100110001011011000010110000110010100101110100110000111
101100001010111010100010110110011111100010111001100101011010001001000000110100101100101101101111011011001000
1011000101000011001001111011000100011011011010010010001001001000000110001011100001101000111111100101101111111

```
000100111000000101011000101111011000011011101110010111101100111111011100001110000101101100111011011010000100010011111011001001101000000111111000100111111101001110110000111011111101110101001100100110011100100000010011000001010000100000001110011101011011000100000111100010100011000110011010111000010000011110010110111111100001111110001010101000100111000000000110111111010111101000000010101010000011000110000000100101011001110000000010111101011100101110001111111111000001100110000010000110010010000010001000000000110000011110010001100110100100
```

Data S8. (txpo50.c)

```
//////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2022, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: The dynamics of an extremely large integer with binary length UNIT.
//Input:
//Output: Original Dynamics, namely, Collatz computation sequence
//Usage Example: txpo50.exe
//Output Example:
//////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>

static int MAXLEN=6000000; //for malloc
static int UNIT=100000;   //lenth of randomly generated integer
static int RUNTIMES=1;

//Three X Plus One Divde Two sub-function. that is, result <= (3*a + 1)/2
int TXPODT(char* a, char* result);

int main(int argc, char *argv[]) {
```

```
char* current_x;    //the current integer
char* temp_x;       //temporary integer in obit

char* dynamics;    //dynamics in terms of `I' and `O'

FILE *fp_dynamics;  //a file to store dynamics in terms of 'I' and 'O', 'I' means (3x+1)/2, and 'O' means x/2

int count_I, count_O;  //the count of 'I' and the count of 'O' in dynamics
int len;               //bit length of integer
int len_of_heightest;  //the length of the heightest, that is the largest length of the binary value in orbit

float ratio_OI=0;      //ratio = count_O/count_I = (the count of `O')/(the count of `I')

int i,j;               //for loop

int start_x_len=0;       //the length of the starting integer
char filename[40];       //Filename for generated file recording all dynamics in terms of 'I' and 'O'
char extendname[20];     //Filename extension on start_x_len

        current_x = (char *)malloc(MAXLEN*5*sizeof(char));
        temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
        dynamics = (char *)malloc(MAXLEN*10*sizeof(char));

        if((current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
        {
                printf("debug: error of malloc.\n");
        }


        for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
        {
        start_x_len = UNIT*(i+1);  //e.g., if UNIT=1000, RUNTIMES=5, then start_x_len =1000,2000,3000,4000,5000
```

```
	strcpy(filename,"txpo50dynamics");  //prepare dynamics file name, the name is
	itoa(start_x_len,extendname,10);    //change maxlen to a string - extendname
	strcat(filename,extendname);        //append maxlen to the file name
	//printf("debug: %s\n",filename);

	fp_dynamics = fopen(filename,"w");

	memset(current_x,0,sizeof(current_x));
	memset(temp_x,0,sizeof(temp_x));
	memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics

	//Generate a random interger with the bit length of start_x_len as a starting integer
	//srand(time(NULL));        //for the random seed generation in C. If enabled, starting integer in each run is different
	strcat(current_x,"1");      //the MSB should be 1
	for(j=0; j < start_x_len-2; j++)     //2 bits are fixed, MSB and LSB
	{
		if(rand()%2==1)        //rand() returns an random integer
			strcat(current_x,"1");
		else
			strcat(current_x,"0");
	}
	strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
	//strcpy(current_x, argv[1]);   //it could also be an input string

	fprintf(fp_dynamics, "%s\n", current_x);  //writing start_x into data file

	count_I=0;
	count_O=0;
	len_of_heightest=0;

	while(strlen(current_x)!=1)        //if outputting is original dynamics instead of reduced dynamics
{
```

```
        len = strlen(current_x);
                if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
    {
        TXPODT(current_x,temp_x);    //call subfunction TXPODT(), i.e., temp_x <= (current_x*3+1)/2

        strcpy(current_x,temp_x);  //current_x <= temp_x;

                        len = strlen(current_x);

                        if(len > len_of_heightest)    //store the largest len, which is only required in (3x+1)/2 computation
                                len_of_heightest = len;            //this length is after (3x+1)/2, length of 3x+1 is len+1

        strcat(dynamics,"I");     //debug - 'I' that means (3x+1)/2, is appended into dynamics

        count_I = count_I + 1;   //count the number of 'I'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
    }
    else                        //current_x is even
    {
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

                        strcat(dynamics,"O");     //debug - 'O' that means x/2, is appended into dynamics

        count_O = count_O + 1;     //count the number of 'O'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
    }
  }       //end of while loop

        ratio_OI = (float)count_O/count_I;    //computing ratio

        //printf("%d %d %d %d %d %18.16f\n%s\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O, ratio_OI,
dynamics);
        fprintf(fp_dynamics, "%s\n", dynamics);  //writing dynamics into data file
```

```
        fprintf(fp_dynamics,"%d %d %d %d %d %18.16f\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O,
ratio_OI);
        fclose(fp_dynamics);

        //0.5849625007211561 < log(1.5)/log2 < 0.5849625007211562, which is the estimated bound
        if (ratio_OI <= 0.584962500721156)
        {
                printf("\n\n\n------Alert! Our Esitmated Bound is Wrong ------\n\n\n");
                exit(0);
        }

        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }       //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        free(temp_x);
        free(dynamics);

   return 1;
}


int TXPODT(char* a, char* txpo_result) //this subfunction computes (3x+1)/2, i.e., txpo_result <= (3*a+1)/2
{

   //`10' or `1' is appended by a||`0' is txpo_result

   //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
   int c = 1;       //only current carrier needs to be stored.

   int i;       //counter in for loop
```

```
    int n = strlen(a);          //the bit length of a, a is the inputting string, to compute 3*a+1
    //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);

    memset(txpo_result, 0, sizeof(txpo_result));
    //printf("debug: txpo_result=%s, strlen(txpo_result)=%d\n", txpo_result, strlen(txpo_result));

    //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
    for (i=n-2; i>=0; i--)
    {
                    if(a[i+1]=='0' && a[i]=='0' && c==0)
                            { c=0; a[i+1]='0'; continue;}
                    if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                            { c=0; a[i+1]='1'; continue;}
                    if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
                            { c=1; a[i+1]='0'; continue;}
                    if(a[i+1]=='1' && a[i]=='1' && c==1)
                            { c=1; a[i+1]='1'; continue;}

       //printf("debug: in TXPODT's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
    }

          if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
       { a[0]='0'; strcat(txpo_result,"1"); }

    strcat(txpo_result, a);    //the other bits are appended to right
    //strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even
    //printf("debug: in TXPO(): txpo_result = %s \n", txpo_result);

    return 1;

}
```

Data S9. (txpo51.c)

```
////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2022, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------
//Function: The dynamics of an extremely large integer with binary length UNIT. Use one parameter in TXPODT().
//Input:
//Output: Original Dynamics, namely, Collatz computation sequence
//Usage Example: txpo51.exe
//Output Example:
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>

static int MAXLEN=6000000;  //for malloc
static int UNIT=100000;   //lenth of randomly generated integer
static int RUNTIMES=1;

//Three X Plus One subfunction, result <= (3*a + 1)/2
int TXPODT(char* a);

int main(int argc, char *argv[]) {

    char* current_x;    //the current integer
    //char* temp_x;     //temporary integer in obit

        char* dynamics;    //dynamics in terms of `I' and `O'

        FILE *fp_dynamics;  //a file to store dynamics in terms of 'I' and 'O', 'I' means (3x+1)/2, and 'O' means x/2
```

```
int count_I, count_O;     //the count of 'I' and the count of 'O' in dynamics
int len;                  //bit length of integer
int len_of_heightest;    //the length of the heightest, that is the largest length of the binary value in orbit

float ratio_OI=0;          //ratio = count_O/count_I = (the count of `O')/(the count of `I')

int i,j;                //i for loop

int start_x_len=0;       //the length of the starting integer
char filename[40];       //Filename for generated file recording all dynamics in terms of 'I' and 'O'
char extendname[20];     //Filename extension on start_x_len

current_x = (char *)malloc(MAXLEN*5*sizeof(char));
//temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
dynamics = (char *)malloc(MAXLEN*10*sizeof(char));

//if((current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
if((current_x == NULL) || (dynamics == NULL))
{
        printf("debug: error of malloc.\n");
}

for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
{
start_x_len = UNIT*(i+1);  //e.g., if UNIT=1000, RUNTIMES=5, then maxlen =1000,2000,3000,4000,5000

strcpy(filename,"txpo51dynamics");  //prepare dynamics file name, the name is
itoa(start_x_len,extendname,10);    //change maxlen to a string - extendname
strcat(filename,extendname);        //append maxlen to the file name
//printf("debug: %s\n",filename);

fp_dynamics = fopen(filename,"w");
```

```
memset(current_x,0,sizeof(current_x));
//memset(temp_x,0,sizeof(temp_x));
memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics

//Generate a random interger with the bit length of maxlen as a start integer
//srand(time(NULL));        //for the random seed generation. if enabled, each run is different
strcat(current_x,"1");      //the MSB should be 1
for(j=0; j < start_x_len-2; j++)     //2 bits are fixed, MSB and LSB
{
        if(rand()%2==1)        //rand() returns an random integer
                strcat(current_x,"1");
        else
                strcat(current_x,"0");
}
strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
//strcpy(current_x, argv[1]);   //it could also be an input string

fprintf(fp_dynamics, "%s\n", current_x);  //writing start_x into data file

count_I=0;
count_O=0;
len_of_heightest=0;

while(strlen(current_x)!=1)        //if outputting is original dynamics instead of reduced dynamics
{
len = strlen(current_x);
        if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
{
TXPODT(current_x);
//printf("debug: After TXPODT(): temp_x=%s\n", temp_x);

                len = strlen(current_x);
                if(len > len_of_heightest)    //store the largest len, which is only required in (3x+1)/2 computation
```

```
                                len_of_heightest = len;          //this length is after (3x+1)/2, length of 3x+1 is len+1

        strcat(dynamics,"I");     //debug - 'I' that means (3x+1)/2, is appended into dynamics

        count_I = count_I + 1;   //count the number of 'I'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
      else                      //current_x is even
      {
        //printf("debug: current_x=%s strlen(current_x)=%d \n", current_x, len);
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

                          strcat(dynamics,"O");     //debug - 'O' that means x/2, is appended into dynamics

        count_O = count_O + 1;     //count the number of 'O'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
   }       //end of while loop

        ratio_OI = (float)count_O/count_I;    //computing ratio

        fprintf(fp_dynamics, "%s\n", dynamics);  //writing dynamics into data file
        fprintf(fp_dynamics,"%d %d %d %d %d %18.16f\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O,
ratio_OI);
        fclose(fp_dynamics);

        //0.5849625007211561 < log(1.5)/log2 < 0.5849625007211562, which is the estimated bound
        if (ratio_OI <= 0.584962500721156)
        {
                printf("\n\n\n------Alert! Our Esitmated Bound is Wrong ------\n\n\n");
                exit(0);
        }
```

```
        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }       //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        //free(temp_x);
        free(dynamics);

    return 1;
}

//this subfunction is compute (3x+1)/2, that is, a <= (3*a + 1)/2
int TXPODT(char* a)
{

    //`10' or `1' is appended by a||`0' is txpo_result

    //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
    int c = 1;      //only current carrier needs to be stored.

    int i;      //counter in for loop

    int n = strlen(a);      //the bit length of a, a is the inputting string, to compute 3*a+1
    //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);

    //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
    for (i=n-2; i>=0; i--)
    {
            if(a[i+1]=='0' && a[i]=='0' && c==0)
                    { c=0; a[i+1]='0'; continue;}
            if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                    { c=0; a[i+1]='1'; continue;}
            if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
```

```
                        { c=1; a[i+1]='0'; continue;}
                if(a[i+1]=='1' && a[i]=='1' && c==1)
                        { c=1; a[i+1]='1'; continue;}

      //printf("debug: in TXPO's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
   }

      if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
      {
                        a[0]='0';
                        //strcat(txpo_result,"1");

                         //put `1' in front of array a, so that the length of array a is changed from n to n+1
                         for(i=n-1;i>=0;i--)
                         {
                                 a[i+1]=a[i];
                         }
                         a[0]='1';
                         a[n+1]='\0';
                }

   //strcat(txpo_result, a);    //the other bits are appended to right

   //strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even
   //printf("debug: in TXPO(): txpo_result = %s \n", txpo_result);

   return 1;

}
```

S10 (txpo49.c)

```
////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2023, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: The dynamics of an extremely large integer with binary length related to UNIT.
//Input:
//Output: Original Dynamics, namely, Collatz computation sequence
//Usage Method: Set UNIT to bit length what you want,
//            e.g., 1000, 10000, 100000, 1000000, ..., and compile (MinGW or GCC),
//            and execute the executable file in a shell (DOS or Linux)
//Usage Example: txpo49.exe
//Output Example: txpo49dynamics1000
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>   //if to generate different random integer in each run, enable it

static int MAXLEN=6000000; //it is used for setting the max length in malloc
static int UNIT=1;     //it is used for setting the length of staring integer
static int RUNTIMES=100000;     //it is used for setting the count of run

//Three X Plus One subfunction, result <= 3*a + 1
int TXPO(char* a, char* result);

int main(int argc, char *argv[]) {

    char* starting_x;   //the starting integer
        char* current_x;    //the current integer
    char* temp_x;       //temporary integer in computing dynamics, i.e., obit
```

```
        char* dynamics;    //dynamics in terms of `I' and `O'

        FILE *fp_dynamics;  //a file to store dynamics in terms of 'I' and 'O', 'I' means (3x+1)/2, and 'O' means x/2

int count_I, count_O;     //the count of 'I' and the count of 'O' in dynamics
int len;
        int len_of_heightest;    //the length of the heightest, that is the largest length of the binary value in orbit

        float ratio_OI=0;        //ratio = count_O/count_I = (the count of `O')/(the count of `I')

int i,j;                //for loop

        int start_x_len=0;       //the length of the starting integer
char filename[40];       //filename of the generated file recording original dynamics in terms of 'I' and 'O'
        char extendname[20];     //filename extension to record start_x_len

        starting_x = (char *)malloc(MAXLEN*5*sizeof(char));
        current_x = (char *)malloc(MAXLEN*5*sizeof(char));
        temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
        dynamics = (char *)malloc(MAXLEN*10*sizeof(char));

        if((starting_x==NULL)||(current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
        {
                printf("debug: error of malloc.\n");
        }

        memset(starting_x,0,sizeof(starting_x));

        for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
        {
        start_x_len = UNIT+(i+1);  //e.g., if UNIT=1000, RUNTIMES=5, then start_x_len =1001,1002,1003,1004,1005

        strcpy(filename,"txpo49dynamics");  //prepare dynamics file name, the name is
```

```
        itoa(start_x_len,extendname,10);    //change maxlen to a string - extendname
        strcat(filename,extendname);        //append maxlen to the file name
        //printf("debug: %s\n",filename);

        fp_dynamics = fopen(filename,"w");

    memset(current_x,0,sizeof(current_x));
        memset(temp_x,0,sizeof(temp_x));
        memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics

        //Generate a random interger with the bit length of maxlen as a starting integer
        //srand(time(NULL));        //for the random seed generation. if enabled, random integer in each run is different
        if(i==0)
        {
        strcat(current_x,"1");      //the MSB should be 1
        for(j=0; j < start_x_len-2; j++)     //2 bits are fixed, MSB and LSB
        {
                if(rand()%2==1)        //rand() returns an random integer
                        strcat(current_x,"1");
                else
                        strcat(current_x,"0");
        }
        strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
        //strcpy(current_x, argv[1]);   //it could also be an inputting string

        strcpy(starting_x,current_x);
        }
        else
        {
        strcat(starting_x,"1");
        strcpy(current_x,starting_x);
        }
    //printf("debug: start_x=%s sizeof(current_x)=%d strlen(current_x)=%d \n", start_x, sizeof(current_x), strlen(current_x));
```

```
        fprintf(fp_dynamics, "%s\n", current_x);  //writing start_x into data file

        count_I=0;
        count_O=0;
        len_of_heightest=0;

        while(strlen(current_x)!=1)        //if outputting is original dynamics instead of reduced dynamics
{
        len = strlen(current_x);
                if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
    {
        TXPO(current_x,temp_x);    //call subfunction TXPO(), i.e., temp_x <= current_x*3+1

        strcpy(current_x,temp_x);  //current_x <= temp_x;

                        len = strlen(current_x);   //note that, current_x is changed, so hereby need to get new length
                        current_x[len-1] = '\0';   //cut down the last bit that is always 0, it means current_x <= current_x/2

                        if(len > len_of_heightest)    //store the largest len, which is only required in (3x+1)/2 computation
                                len_of_heightest = len;

        strcat(dynamics,"I");     //debug - 'I' that means (3x+1)/2, is appended into dynamics

        count_I = count_I + 1;   //count the number of 'I'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
    }
    else                    //current_x is even
    {
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

                        strcat(dynamics,"O");     //debug - 'O' that means x/2, is appended into dynamics

        count_O = count_O + 1;     //count the number of 'O'
```

```
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
   }       //end of while loop

        ratio_OI = (float)count_O/count_I;    //computing ratio

        //printf("%d %d %d %d %d %18.16f\n%s\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O, ratio_OI,
dynamics);
        fprintf(fp_dynamics, "%s\n", dynamics);  //writing dynamics into data file

        fprintf(fp_dynamics,"%d %d %d %d %d %18.16f\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O,
ratio_OI);
        fclose(fp_dynamics);

        //0.5849625007211561 < log(1.5)/log2 < 0.5849625007211562, which is the estimated bound
        if (ratio_OI <= 0.584962500721156)
        {
                printf("\n\n\n------Alert! Our Esitmated Bound is Wrong ------\n\n\n");
                exit(0);
        }

        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }       //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        free(temp_x);
        free(dynamics);

   return 1;
}
```

```
int TXPO(char* a, char* txpo_result) //this subfunction is compute 3x+1, that is, txpo_result <= 3*a + 1
{

        //recall that, `10' or `1' is appended by "a||0" is txpo_result

   //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
   int c = 1;       //only current carrier needs to be stored.

        int i;       //counter in for loop

   int n = strlen(a);        //the bit length of a, a is the inputting string, to compute 3*a+1
   //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);

        memset(txpo_result, 0, sizeof(txpo_result));
        //printf("debug: txpo_result=%s, strlen(txpo_result)=%d\n", txpo_result, strlen(txpo_result));

        //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
   for (i=n-2; i>=0; i--)
   {
                if(a[i+1]=='0' && a[i]=='0' && c==0)
                        { c=0; a[i+1]='0'; continue;}
                if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                        { c=0; a[i+1]='1'; continue;}
                if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
                        { c=1; a[i+1]='0'; continue;}
                if(a[i+1]=='1' && a[i]=='1' && c==1)
                        { c=1; a[i+1]='1'; continue;}

      //printf("debug: in TXPO's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
   }

        if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
      { a[0]='0'; strcat(txpo_result,"1"); }
```

```
    strcat(txpo_result, a);    //the other bits are appended to right
        strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even
    //printf("debug: in TXPO(): txpo_result = %s \n", txpo_result);

    return 1;

}
```

S11 (txpo53.c)

```
////////////////////////////////////////////////////////////////////////////
//Copyright 2016-2023, Dr. Wei Ren, China University of Geosciences, Wuhan, China
//              Email: weirencs@cug.edu.cn
//--------------------------------------------------------------------------
//Function: The dynamics of an extremely large integer with binary length related to UNIT.
//Input:
//Output: Original Dynamics, namely, Collatz computation sequence
//Usage Method: Set UNIT to bit length what you want,
//            e.g., 1000, 10000, 100000, 1000000, ..., and compile (MinGW or GCC),
//            and execute the executable file in a shell (DOS or Linux)
//Usage Example: txpo53.exe
//Output Example: txpo53dynamics100002seglen1000run1, ..., txpo53dynamics102run1000
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>   //if to generate different random integer in each run, enable it

static int MAXLEN=7000000; //it is used for setting the max length in malloc
```

```
static int UNIT=100000;     //it is used for setting the length of staring integer
static int RUNTIMES=100;     //it is used for setting the count of running times
static int SEGMENTLENGTH=10000; //it is the length of segment

//Three X Plus One subfunction, result <= 3*a + 1
int TXPO(char* a, char* result);

int main(int argc, char *argv[]) {

   char* segment;      //the repeating of a segment as a starting x
        char* current_x;    //the current integer
   char* temp_x;       //temporary integer in computing dynamics, i.e., obit

        char* dynamics;    //dynamics in terms of `I' and `O'

        FILE *fp_dynamics;  //a file to store dynamics in terms of 'I' and 'O', 'I' means (3x+1)/2, and 'O' means x/2

   int count_I, count_O;     //the count of 'I' and the count of 'O' in dynamics
   int len;
        int len_of_heightest;    //the length of the heightest, that is the largest length of the binary value in orbit

        float ratio_OI=0;        //ratio = count_O/count_I = (the count of `O')/(the count of `I')

   int i,j;                //for loop

        int start_x_len=0;       //the length of the starting integer
   char filename[40];       //filename of the generated file recording original dynamics in terms of 'I' and 'O'
        char extendname[20];     //filename extension to record start_x_len

        current_x = (char *)malloc(MAXLEN*5*sizeof(char));
        temp_x = (char *)malloc(MAXLEN*5*sizeof(char));
        dynamics = (char *)malloc(MAXLEN*10*sizeof(char));
        segment = (char *)malloc(MAXLEN*sizeof(char));
```

```
        if((current_x == NULL) || (temp_x == NULL) || (dynamics == NULL))
        {
                printf("debug: error of malloc.\n");
        }


        for(i=0;i<RUNTIMES;i++)   //RUNTIMES is the counts of the loop
        {
        start_x_len = UNIT;  //e.g., if UNIT=1000, RUNTIMES=5, then try 5 times of random integers with length 1000

        strcpy(filename,"txpo53dynamics");  //prepare dynamics file name, the name is
        itoa(start_x_len+2,extendname,10);    //change maxlen to a string - extendname
        strcat(filename,extendname);        //append maxlen to the file name
        itoa(SEGMENTLENGTH,extendname,10);
        strcat(filename,"seglen");
        strcat(filename,extendname);        //append segment length to the file name
        itoa(i+1,extendname,10);            //the (i+1)-th running time, from 1 to RUNTIMES
        strcat(filename,"run");
        strcat(filename,extendname);        //append the (i+1)-th running time to the file name
        //printf("debug: %s\n",filename);

        fp_dynamics = fopen(filename,"w");

memset(current_x,0,sizeof(current_x));
        memset(temp_x,0,sizeof(temp_x));
        memset(dynamics,0,sizeof(dynamics)); //initialization of dynamics
        memset(segment,0,sizeof(segment));

        //Generate a random interger with the bit length of maxlen as a starting integer
        //srand(time(NULL));        //for the random seed generation. if enabled, random integer in each run is different
        strcat(current_x,"1");      //the MSB should be 1
        for(j=0; j < SEGMENTLENGTH; j++)     //2 bits are fixed, MSB and LSB
```

```
        {
                if(rand()%2==1)       //rand() returns an random integer
                        strcat(segment,"1");      //segment is randomly generated
                else
                        strcat(segment,"0");
        }

        for(j=0;j<UNIT/SEGMENTLENGTH;j++)
                strcat(current_x,segment);   //concatinate the segments repeatly

        strcat(current_x,"1");      //the LSB should be 1, as the randomly generated integer should be odd.
        //strcpy(current_x, argv[1]);   //it could also be an inputting string

//printf("debug: start_x=%s sizeof(current_x)=%d strlen(current_x)=%d \n", start_x, sizeof(current_x), strlen(current_x));
        fprintf(fp_dynamics, "%s\n", current_x);  //writing start_x into data file

        count_I=0;
        count_O=0;
        len_of_heightest=0;

        while(strlen(current_x)!=1)       //if outputting is original dynamics instead of reduced dynamics
{
        len = strlen(current_x);
                if(current_x[len-1]=='1')      // if current integer x is odd, means if the LSB is 1
    {
        TXPO(current_x,temp_x);    //call subfunction TXPO(), i.e., temp_x <= current_x*3+1

        strcpy(current_x,temp_x);  //current_x <= temp_x;

                        len = strlen(current_x);   //note that, current_x is changed, so hereby need to get new length
                        current_x[len-1] = '\0';   //cut down the last bit that is always 0, it means current_x <= current_x/2

                        if(len > len_of_heightest)    //store the largest len, which is only required in (3x+1)/2 computation
```

```
                                len_of_heightest = len;

        strcat(dynamics,"I");     //debug - 'I' that means (3x+1)/2, is appended into dynamics

        count_I = count_I + 1;   //count the number of 'I'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
      else                      //current_x is even
      {
        current_x[len-1] = '\0';    //cut off the last bit that is 0, it means current_x <= current_x/2

                          strcat(dynamics,"O");     //debug - 'O' that means x/2, is appended into dynamics

        count_O = count_O + 1;     //count the number of 'O'
        //printf("debug: %s\t%d\t%d\t%s \n\n", current_x, count_I, count_O, dynamics);
      }
   }       //end of while loop

        ratio_OI = (float)count_O/count_I;    //computing ratio

        //printf("%d %d %d %d %d %18.16f\n%s\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O, ratio_OI,
dynamics);
        fprintf(fp_dynamics, "%s\n", dynamics);  //writing dynamics into data file

        fprintf(fp_dynamics,"%d %d %d %d %d %18.16f\n", start_x_len, len_of_heightest, count_I+count_O, count_I, count_O,
ratio_OI);
        fclose(fp_dynamics);

        //0.5849625007211561 < log(1.5)/log2 < 0.5849625007211562, which is the estimated bound
        if (ratio_OI <= 0.584962500721156)
        {
                printf("\n\n\n------Alert! Our Esitmated Bound is Wrong ------\n\n\n");
                exit(0);
```

```
        }

        printf("debug:i=%d is done in for(i=0;i<RUNTIMES;i++) loop.\n",i);

        }       //end of for(i=0;i<RUNTIMES;i++) loop

        free(current_x);
        free(temp_x);
        free(dynamics);

   return 1;
}


int TXPO(char* a, char* txpo_result) //this subfunction is compute 3x+1, that is, txpo_result <= 3*a + 1
{

        //recall that, `10' or `1' is appended by "a||0" is txpo_result

   //because rightest bit (LSB) of a is 1, and +1 (in 3x+1), so the carrier is 1
   int c = 1;       //only current carrier needs to be stored.

        int i;       //counter in for loop

   int n = strlen(a);       //the bit length of a, a is the inputting string, to compute 3*a+1
   //printf("debug: in TXPO(): n=%d, a=%s a[%d]=%c \n", n, a, n-1, a[n-1]);

        memset(txpo_result, 0, sizeof(txpo_result));
        //printf("debug: txpo_result=%s, strlen(txpo_result)=%d\n", txpo_result, strlen(txpo_result));

        //bit addition from rightest bit to leftest bit, or, from LSB to MSB.
   for (i=n-2; i>=0; i--)
   {
```

```
                if(a[i+1]=='0' && a[i]=='0' && c==0)
                        { c=0; a[i+1]='0'; continue;}
                if((a[i+1]=='0' && a[i]=='0' && c==1)||(a[i+1]=='0' && a[i]=='1' && c==0)||(a[i+1]=='1' && a[i]=='0' && c==0))
                        { c=0; a[i+1]='1'; continue;}
                if((a[i+1]=='0' && a[i]=='1' && c==1)||(a[i+1]=='1' && a[i]=='0' && c==1)||(a[i+1]=='1' && a[i]=='1' && c==0))
                        { c=1; a[i+1]='0'; continue;}
                if(a[i+1]=='1' && a[i]=='1' && c==1)
                        { c=1; a[i+1]='1'; continue;}

      //printf("debug: in TXPO's loop: a[%d]=%c c=%d \n", i+1, a[i+1], c);
   }

        if (c==1)    //carrier is 1, leftest bit (MSB) of a is 1, so 1+1=10, the head is ``10"
      { a[0]='0'; strcat(txpo_result,"1"); }

   strcat(txpo_result, a);    //the other bits are appended to right
        strcat(txpo_result,"0");   //the LSB of txpo_result is 0, as 3*x+1 must be even
   //printf("debug: in TXPO(): txpo_result = %s \n", txpo_result);

   return 1;

}
```