CDM
Iteration II

Klaus Sutner
Carnegie Mellon Universality

32-iteration 2017/12/15 23:20

Ducci Sequences

Here is a classical if slightly frivolous example of an apparently simple iteration that has lots of interesting properties.

> Place 4 integers on a circle. Compute the absolute values of all differences of adjacent pairs of numbers. Write these values between the corresponding numbers and erase the numbers themselves.
>
> Repeat. What happens?

This was introduced by Enrico Ducci (1864-1940) in the 1930s, and was mostly forgotten till Honsberger's "Ingenuity in Mathematics" appeared in 1970.

Less Informally

We can ignore integers, after one step we are dealing only with natural numbers.

So, we want to iterate the following function on $\mathbb{N}^4$:

$$D(x_1, x_2, x_3, x_4) = (|x_1 - x_2|, |x_2 - x_3|, |x_3 - x_4|, |x_4 - x_1|)$$

Is there anything one can say about repeated application of this function?

Sample Ducci Sequences

Here are two small examples, we iterate D on "random" initial conditions.

0	10	13	4	20
1	3	9	16	10
2	6	7	6	7
3	1	1	1	1
4	0	0	0	0

0	94	68	11	85
1	26	57	74	9
2	31	17	65	17
3	14	48	48	14
4	34	0	34	0
5	34	34	34	34
6	0	0	0	0

Somewhat surprisingly, after a few steps we reach the 0 vector which is, of course, a fixed point of the operation.

Exercise

Produce initial values so that it takes many steps to reach $\mathbf{0}$.

The Basics

The first observation is that all orbits end in fixed point $\mathbf{0}$.

Theorem (Ducci)

After finitely many steps the fixed point $(0, 0, 0, 0)$ *is reached.*

Proof?

Call the maximum element of a vector its weight.

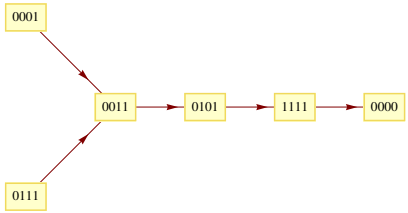
Clearly, weights are non-increasing and it suffices to show that they decrease every once in a while.

A little fumbling shows that weight can be preserved only if the vector looks like $(0, a, b, c)$ where $a \geq b, c$, up to rotation.

Now we can do a straightforward but exceedingly boring case analysis.

A Better Way

Consider only the parity of a vector, up to rotations:



Exercise

Verify the diagram and finish the argument.

Transients

It is rather surprisingly difficult to choose initial condition that lead to long runs before $\mathbf{0}$ is reached, but there is no bound on the number of steps.

Simply choosing large numbers won't work: they tend to cancel each other out.

Lemma (Webb 1982)

There is no bound on the number of steps needed to reach $\mathbf{0}$.

The idea is to consider a family X_n of special vectors with the hope that $D(X_n) = X_{n-1}$.

Alas, life is a bit more complicated than that.

Long Transients

To construct initial conditions leading to long transients, consider "tribonacci numbers"

$$t_n = t_{n-1} + t_{n-2} + t_{n-3} \qquad t_0 = 0 \qquad t_1 = t_2 = 1$$

The first few values are

$$0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, 927, 1705, 3136, \ldots$$

Now consider the configurations $X_n = (t_n, t_{n-1}, t_{n-2}, t_{n-3})$.

A little bit of work shows

$$D^3(X_n) = 2 \cdot R(X_{n-2})$$

where R denotes cyclic shift to the right.

Generalization I: How About the Reals?

We might try to force divergence by using irrational or even transcendental numbers. At least the following valiant attempt fails.

0	0	$\sqrt{2}$	$\sqrt{7}$	π
1	$\sqrt{2}$	$-\sqrt{2}+\sqrt{7}$	$-\sqrt{7}+\pi$	π
2	$2\sqrt{2}-\sqrt{7}$	$-\sqrt{2}+2\sqrt{7}-\pi$	$\sqrt{7}$	$-\sqrt{2}+\pi$
3	$-3\sqrt{2}+3\sqrt{7}-\pi$	$\sqrt{2}-\sqrt{7}+\pi$	$\sqrt{2}+\sqrt{7}-\pi$	$-3\sqrt{2}+\sqrt{7}+\pi$
4	$4\sqrt{2}-4\sqrt{7}+2\pi$	$-2\sqrt{7}+2\pi$	$-4\sqrt{2}+2\pi$	$-2\sqrt{7}+2\pi$
5	$4\sqrt{2}-2\sqrt{7}$	$4\sqrt{2}-2\sqrt{7}$	$4\sqrt{2}-2\sqrt{7}$	$4\sqrt{2}-2\sqrt{7}$
6	0	0	0	0

Exercise

Explain why this and similar attempts will fail to produce divergence.

An Amazing Exception

After a few more experiments one might conclude that convergence to $\mathbf{0}$ always occurs, but that is false. Let q be the real root of $x^3 - x^2 - x - 1 = 0$, so

$$q = \frac{1}{3}\left(1 + \sqrt[3]{19 - 3\sqrt{33}} + \sqrt[3]{19 + 3\sqrt{33}}\right) \approx 1.83929$$

Then $(1, q, q^2, q^3)$ does not converge. E.g. after 5 steps we get a term

$$\frac{1}{9}\left(24 - 4\sqrt[3]{19+3\sqrt{33}} - 6\left(19+3\sqrt{33}\right)^{2/3} + \left(19-3\sqrt{33}\right)^{2/3} \left(-6 + 4\sqrt[3]{19+3\sqrt{33}}\right) + 4\sqrt[3]{19-3\sqrt{33}}\left(-1 + \left(19+3\sqrt{33}\right)^{2/3}\right)\right)$$

This was shown by M. Lotan in 1949.

Generalization II: How About $n \neq 4$?

Over $\mathbb{N}^n$ evolution does no longer lead to a fixed point in general, but eventually becomes periodic.

Theorem (Ciamberlini, Marengoni 1937)

Every Ducci sequence of width n *ends in fixed point* $\mathbf{0}$ *if, and only if,* n *is a power of 2.*

Exercise

Why must any Ducci sequence over $\mathbb{N}^n$ *be ultimately periodic?*

Example: $n = 5$

0	1	2	3	4	5	6	7	8	9	10	11
4	2	2	3	1	3	1	2	1	2	1	1
6	4	5	2	4	2	3	1	3	1	0	1
10	9	7	6	6	5	4	4	2	1	1	1
1	2	1	0	1	1	0	2	1	0	0	2
3	1	1	1	2	1	2	1	1	0	2	1

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	0	1	0	0	0	1	1	1	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0	0	1	1	1	1	0	1	0
1	0	1	0	1	1	0	0	1	0	0	0	1	1	1	1
1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1
0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0

It is a bit surprising that after 12 steps the system is in a binary state. From then on, it repeats every 15 steps.

The Limit Cycles

Needless to say, the binary behavior in the example is no coincidence.

Theorem (Burmester, Forcade, Jacobs 1978)

All Ducci sequences over the integers are ultimately periodic. The vectors on the limit cycle are binary in the sense that $x_i \in \{0, c\}$ for some c.

Once the vector is binary, the Ducci operator degenerates into exclusive or:

$$D(\boldsymbol{x}) = \boldsymbol{x} \text{ xor } L(\boldsymbol{x})$$

where L denotes the cyclic left-shift operation. We'll come back to this later in our discussion of cellular automata.

Pseudo-Random Numbers

Iteration is a nice and orderly way to generate sequences, so it somewhat surprising that it also provides a computationally efficient way produce pseudo-random numbers.

One usually wants a numbers modulo some fixed N. The basic idea is simple: pick a function

$$f : \mathbb{Z}_N \to \mathbb{Z}_N$$

at random and use the sequence $x_n = f^n(x_0)$ where x_0 is chosen somehow–preferably at random ;-).

Then the sequence x_n has nice properties:

Lemma

The expected value of transient and period of (x_n) is $\Theta(\sqrt{N})$.

Minor Obstruction

"Pick f and random" is not a helpful instruction. What we need instead is an explicit method to compute $f(x)$. Preferably the computation should be cheap, say, quadratic in $\log N$.

> Anyone attempting to produce random numbers by purely arithmetic means is, of course, in a state of sin.
>
> John von Neumann

But there is a big surprise: there are good choices for f that are easy to compute and still behave sufficiently random for many applications.

Warning: Cryptographic applications are very tricky.

Using Arithmetic

A popular choice going back to Derrick Lehmer (1948) are linear congruential generators:

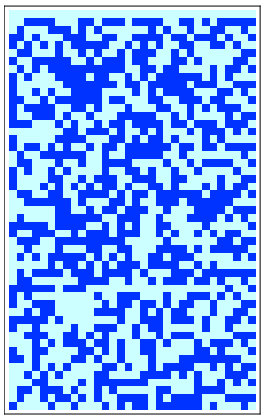
$$f(x) = a \cdot x + b \bmod N$$

for suitable choices of a and b.

Getting the constants right is critical, bad choices ruin the LCG: A reasonably good choice is for example

$$a = 1664525, b = 1013904223, N = 2^{32}$$

A Run



Multiplicative Congruential Generator

Omit the additive offset and use multiplicative constants only. If need be, use a higher order recurrence.

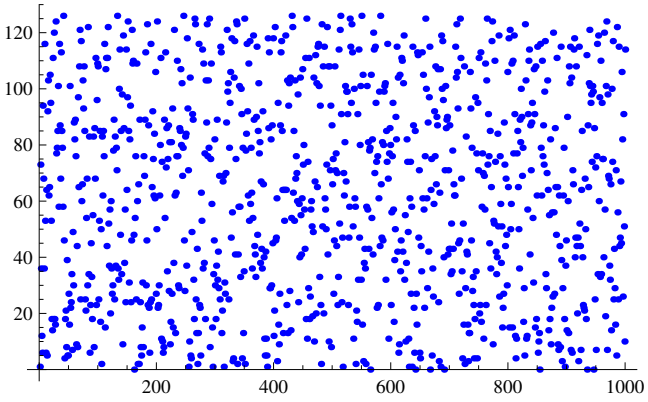
$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \ldots a_k x_{n-k} \bmod N$$

For prime moduli one can achieve period length $N^k - 1$.

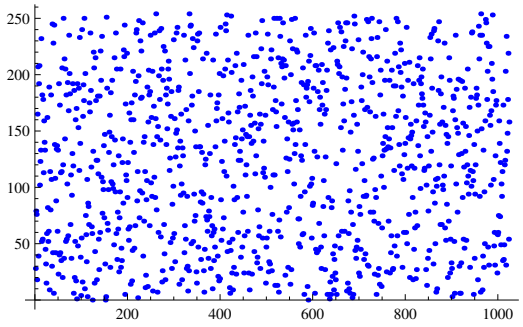
This is almost as fast and easy to implement as LCG (though there is of course more work involved in calculating modulo a prime).

This is still an iteration but over vectors of numbers, there is more state: $x_{n-1}, x_{n-2}, \ldots, x_{n-k}$.

A Run



Truly Random Bits



These are generated with a Geiger counter and krypton 85, see `http://www.fourmilab.ch/hotbits`.

WTF?

The gentle reader might wonder what Geiger counters have to do with randomness.

More precisely, there is a strange and not really well-understood connection between the purely mathematical notion of randomness (which, by the way, is rather difficult to define in any halfway satisfactory manner) and physical processes that seem to involve some intuitive property of "randomness."

See the notes on the website for more background.

Inverse Congruential Generator

If the modulus N is prime one can use multiplicative inverses in $\mathbb{Z}_N$. Write

$$\overline{x} = \begin{cases} 0 & \text{if } x = 0, \\ x^{-1} & \text{otherwise.} \end{cases}$$

Then we can define a pseudo-random sequence by

$$f(x) = a\,\overline{x} + b \bmod N$$

Computing the inverse can be handled by the extended Euclidean algorithm. Again, it is crucial to choose the proper values for the coefficients.

Quadratic Congruential Generators

As one might suspect, one can also use non-linear functions:

$$f(x) = a \cdot x^2 + b \cdot x + c \bmod N$$

Again, selecting the right parameters is tricky and requires some care.

More generally one could use some polynomial

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_0 \bmod N$$

But note that higher degrees are computationally more expensive.

Factoring Integers

There is a factoring method due to John Pollard in 1975 that works well if the given number has a small factor.

Suppose n is some given non-prime. The idea is to use a pseudo-random function f to generate a sequence (x_n) modulo n and to check if

$$\gcd(x_i - x_{2i}, n) > 1$$

If so, we terminate and return the gcd. Otherwise we terminate in failure when we reach $x_i = x_{2i}$ as in Floyd's algorithm.

Pollard's Algorithm


```
x = y = 2;   // or some such
d = 1;

while( d == 1 )
       x = f(x);
       y = f(f(y));
       d = gcd( x - y, n );
       if( d == n ) return failure;
       if( 1 < d ) return d;
```

This assumes that the gcd can handle negative arguments; if not use $|x - y|$.

8th Fermat Number

Using the simple QCG

$$f(x) = x^2 - 1 \bmod n$$

this method produced a huge success story at the time: in 1975 Pollard and Brent found the factor 1238926361552897 of the 8th Fermat number, in two hours compute time on a UNIVAC 1100/42.

$$F_8 = 2^{2^8} + 1 = 115792089237316195423570985008687907853269984665640564039457584007913129639937$$

But Why?

If the sequence generated by f were truly random then by probability theory one should expect transient and period to be $\Theta(\sqrt{n})$, so the algorithm would terminate after some $\Theta(\sqrt{n})$ steps.

But now suppose n has a small divisor m. Since we are using a polynomial f we can think of tacitly computing the sequence modulo m, with the stopping condition not equality but equality modulo m: from $x = y \bmod m$ it follows that $\gcd(x - y, n)$ must be at least m and our algorithm promptly reports a factor.

Thus we should expect the algorithm to return a correct answer in just $\Theta(\sqrt{m})$ time, a much smaller value.

The Power of Wishful Thinking

To the purist, this will sound very disturbing: we pretend there is randomness where it is clearly absent.

However, experimental results show that the algorithm behaves as the argument above would suggest. A really precise analysis seems very difficult, though.

Exercise

What happens with the algorithm when n is prime? What if $n = pq$ where p and q are prime? How could other QCGs $f(x) = x^2 - c \bmod n$ be used?

Undecidability

So what is the computational complexity of iteration?

It is not hard to express the operation of a register machine as a fairly simple function f and set things up in such a way that $\mathrm{FP}(f, x)$ exists iff the computation starting on configuration x is halting.

Hence the existence of fixed points of computable arithmetic functions is undecidable.

Exercise

Figure out the details.

As often, undecidability casts a shadow: iterating very simple functions can create enormous complexity.

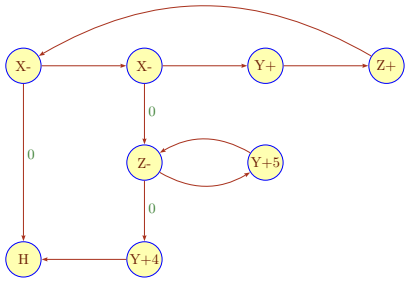
A Termination Question

Here is a seemingly innocent question: Does the following program halt for all $x \geq 1$?

```
while x > 1:              // x positive integer
  if x even:
      x = x/2
  else:
      x = 3 * x + 1
```

The body of the while-loop is rather trivial, just some very basic arithmetic and one if-then-else. This should not be difficult, right?

Collatz Register Machine



This computes the basic step $x \rightsquigarrow 3x + 1$ or $x/2$.

The Collatz Loop Program


```
//  Collatz:  x --> z

    e = 1; o = 0;
        do x : t = e; e = o; o = t; od

        u = 0; v = 0;
        do x : t = u; u = v; v = t; u++; od

        w = 0;
        do x : w++; w++; w++; od
        w++;

        do e : z = u; od
        do o : z = w; od
```

Exercise

Figure out exactly how this program works.

Collatz $3x+1$

The Collatz Problem revolves around the following function C on the positive integers. There are several variants of this in the literature, under different names.

$$C(x) = \begin{cases} 1 & \text{if } x = 1, \\ x/2 & \text{if } x \text{ even}, \\ (3x+1)/2 & \text{otherwise.} \end{cases}$$

This definition is slightly non-standard; usually case 1 is omitted and case 3 reads $3x+1$. Here are the first few values.

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
$C(x)$	1	1	5	2	8	3	11	4	14	5	17	6	20	7	23	8	...

Arithmetic Only

The definition by cases for C is arguably the most natural.

But, we can get by without logic and make the arithmetic slightly more complicated. The following version does not treat argument 1 separately and does not divide by 2 when the argument is odd.

$$C_{\text{ari}}(x) = x/2 - (5x+2)((-1)^x - 1)/4$$

Digression: The Name of the Game

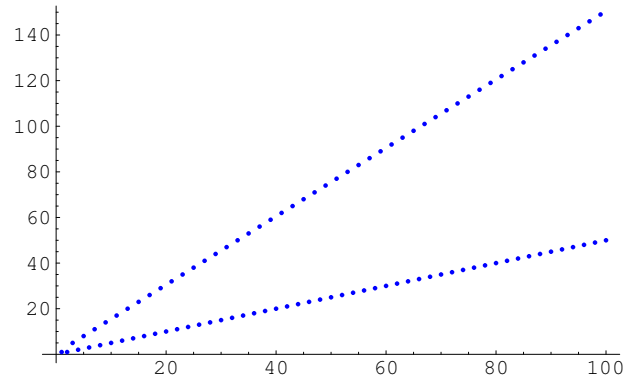
The Collatz problem was invented by Lothar Collatz around 1937, when he was some 20 years old.

Since then, it has assumed a number of aliases:

Ulam, Hasse, Syracuse, Hailstone, Kakutani, ...

Amazingly, in 1985 Sir Bryan Thwaites wrote a paper titled "My Conjecture" claiming fatherhood. Talking about ethics ...

Boring Plot



Repetition

Of course, we are interested not in single applications of C but in repeated application. In fact, the Collatz program keeps computing C until 1 is reached, if ever.

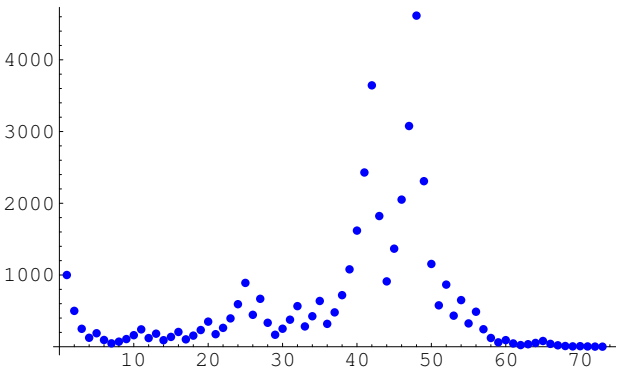
Starting at 18:

$$18, 9, 14, 7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1, 1, 1, \ldots$$

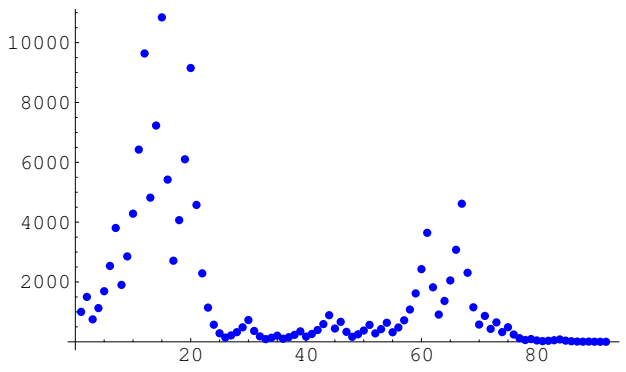
Starting at 1000:

1000,500,250,125,188,94,47,71,107,161,242,121,182,91,137,
206,103,155,233,350,175,263,395,593,890,445,668,334,167,251,
377,566,283,425,638,319,479,719,1079,1619,2429,3644,1822,911,
1367,2051,3077,4616,2308,1154,577,866,433,650,325,488,244,
122,61,92,46,23,35,53,80,40,20,10,5,8,4,2,1,1,1,1,...

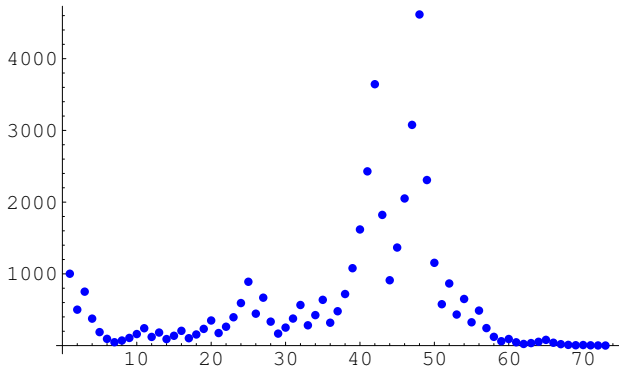
$x = 1000$



$x = 1001$

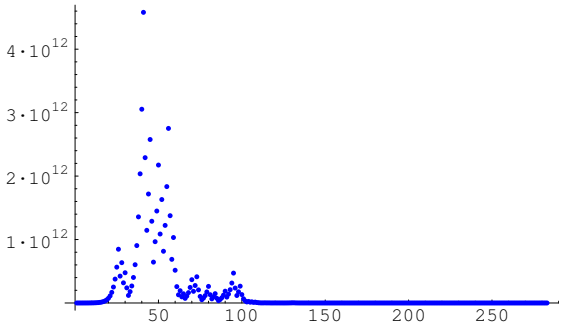


$x = 1002$



Near Powers of 2

Starting at $2^{25} - 1 \approx 3.35 \times 10^7$.



It takes 282 steps to get to 1.

Collatz Conjecture

More computation shows that for all

$$x \leq 3 \cdot 2^{53} \approx 2.7 \cdot 10^{16}$$

the program always halts: C reaches the fixed point 1. Many other values of x have also been tested.

Based on computational evidence as well as various clever arguments one has the following conjecture:

Collatz Conjecture:
All orbits under C end in fixed point 1.

Stopping Times

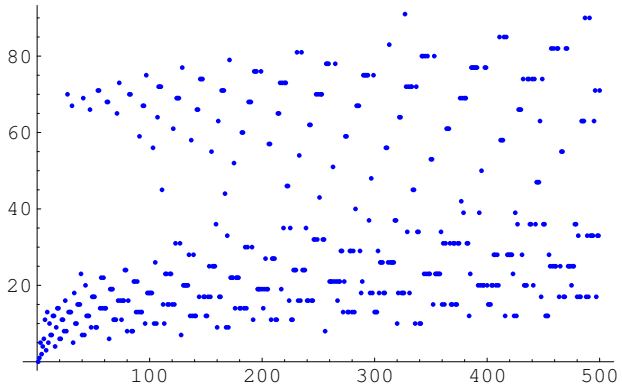
In an attempt to prove the Collatz Conjecture it is natural to try to investigate the stopping time: number of executions of the loop before 1 is reached.

$$\sigma(x) = \begin{cases} \min\big(t \mid C^t(x) = 1\big) & \text{if } t \text{ exists,} \\ \infty & \text{otherwise.} \end{cases}$$

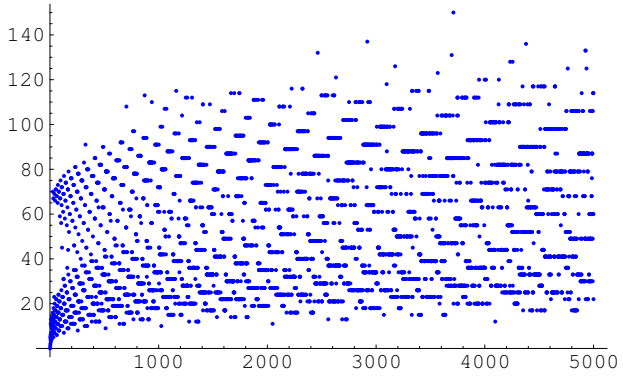
So the Collatz Conjecture holds iff $\sigma(x) < \infty$ for all x.

The stopping time function σ seems slightly more regular than C itself, but it's still rather complicated.

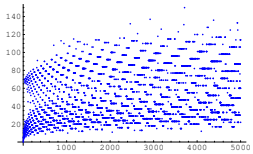
Stopping Times up to 500



Stopping Times up to 5000



Stopping Times up to 5000



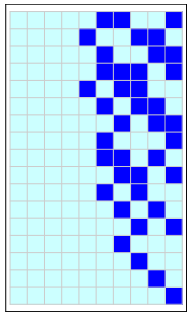
There clearly is some structure here (the dots are certainly not random), but what exactly is this mysterious structure?

A good way to make this precise is prediction: if we already know the first 5000 dots, how hard is it to predict the position of dot 5001?

At this point, no one knows how to do this without computing the whole orbit (at least not for arbitrary values of 5001).

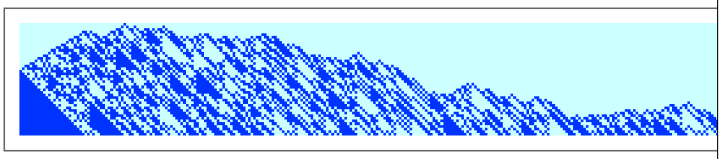
Orbit of 25 in Binary

Another potential way to gain insight in the behavior of the Collatz function is to plot the numbers in binary (an implementation of C for large arguments would use binary arrays anyways).

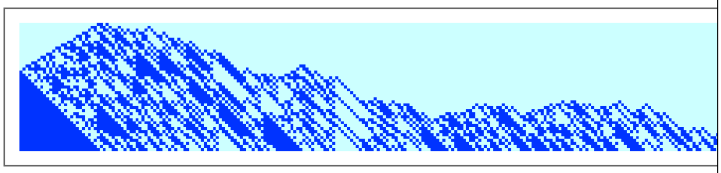


Collatz Mountains

$x = 2^{25} - 1 \approx 3.35 \times 10^7$



$x = 2^{27} - 1 \approx 1.34 \times 10^8$



Computing the Collatz Function

The question arises: what is the best way to compute the Collatz function? We would like to be able to cope with numbers with thousands of bits.

The easy way out is to use a high-quality unbounded precision arithmetic library such as GNU's GMP.

```
BigInt  Collatz( const BigInt& x ) {
  if( x even )
      return x/2;
  else
      return 3*x + 1;
}
```

Big advantage: This is really easy to implement and errors are just about impossible.

Faster

The disadvantage: we are firing intercontinental missiles at sparrows.

Furthermore, memory management is important, we should avoid multiple allocations and deallocations.

Observation 1: We don't really need arithmetic.

If the given number is represented by a bit-vector we can do

```
    return x >> 1;
// or
    return (x<<1) + x + 1;
```

The last operation can be handled without arithmetic.

No Arithmetic

We only need to worry about the case $x_0 = 1$. Let's assume the LSD is first.

x	1	x_1	x_2	x_3	x_4	...
$2x+1$	1	1	x_1	x_2	x_3	...
$3x+1$	0	x_1	$x_1 + x_2 + 1$	$x_2 + x_3 + c$	$x_3 + x_4 + c'$	...

So, we can scan over the input once and compute the output as we go along. In fact, we can write the output into the same bit-vector (more or less).

The whole computation is linear in the number of bits with very small constants.

No Allocation

Observation 2: We can avoid allocations by using one large bit-vector plus two pointers `lo` and `hi` that delimit the currently active block of digits.

Division by 2 is then $O(1)$: `lo++`

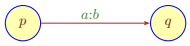
$3x + 1$ requires linear time and moves the active block to the right.

Every once in a while we reach the right end of the allocated block and have to shift everything back to beginning (actually, for reasonable size n-bit numbers allocating $6n$ bits seems to avoid this problem).

Digression: Wordprocessing

One can think of the Collatz function as a simple wordprocessing-type operation on binary strings (the reverse binary expansion of the number in question).

The operation is extremely simple, it requires no extra memory and can be modeled by a system performing state transitions while reading input bits and writing output bits:

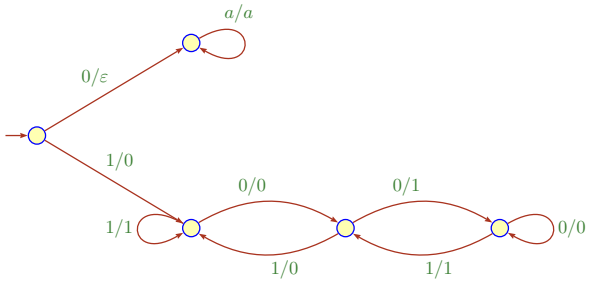


meaning that the device, starting at state p and under input a, outputs b and performs a transition to state q.

Repeating these transitions each input word $x \in \mathbf{2}^\star$ produces an output word $y \in \mathbf{2}^\star$.

Such a device is called a transducer: it translates binary strings into binary strings (similar to the example from lecture 1).

A Collatz Transducer



A transducer for the standard Collatz function. One little glitch: the input must be coded as $x00$ where x is the reverse binary expansion (LSD first, pad right by two 0's).

A Challenge

Implement a faster program to compute the Collatz function.

For example, the following output is generated by a program that runs in 0.2 seconds on a 2.66 GHz machine. Here $x = 2^k - 1$, $k = 1000, \ldots, 1010$.

k	stop	up	down	width
1000	7841	4316	3525	1585
1001	8804	4923	3881	1587
1002	8805	4923	3882	1589
1003	8806	4923	3883	1590
1004	8807	4923	3884	1592
1005	8808	4923	3885	1595
1006	8809	4923	3886	1595
1007	9127	5123	4004	1597
1008	9128	5123	4005	1598
1009	9129	5123	4006	1600
1010	9130	5123	4007	1601