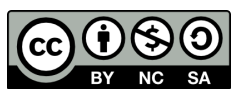
CDM

Iteration

Klaus Sutner

Carnegie Mellon University

Spring 2023



1 **Iteration, Trajectories and Orbits**

2 **∗ Finding Cycles**

3 **Goodstein Sequences**

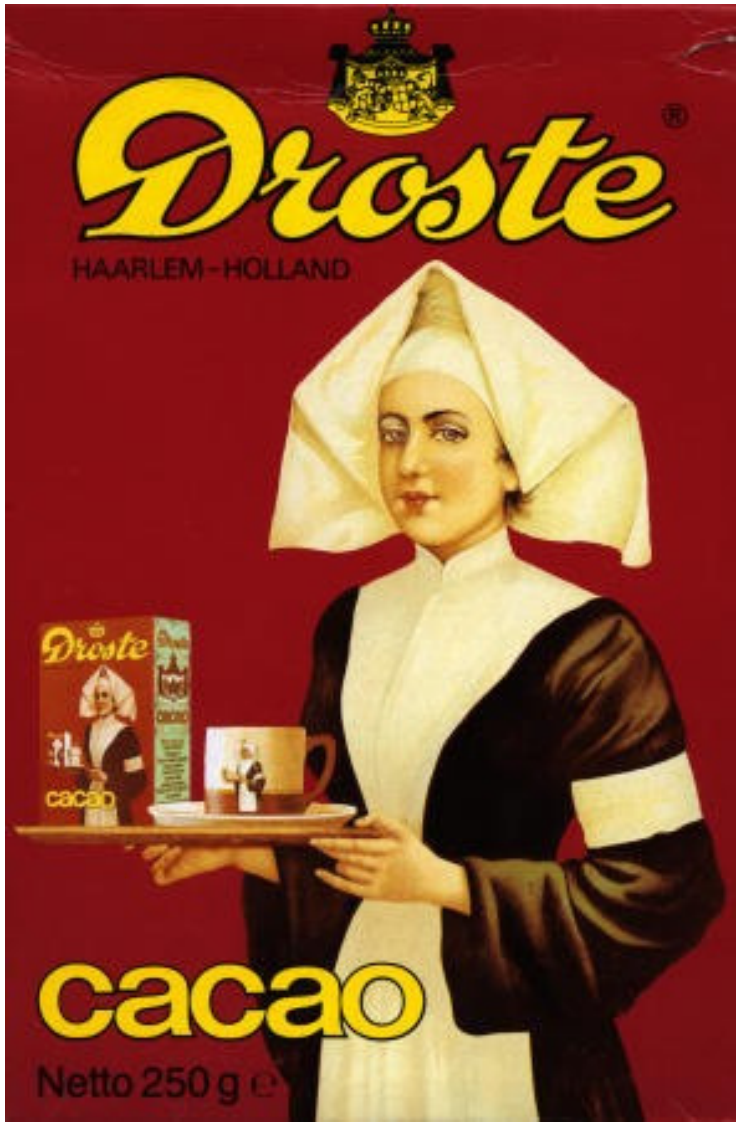
4 **∗ Finding Cycles**

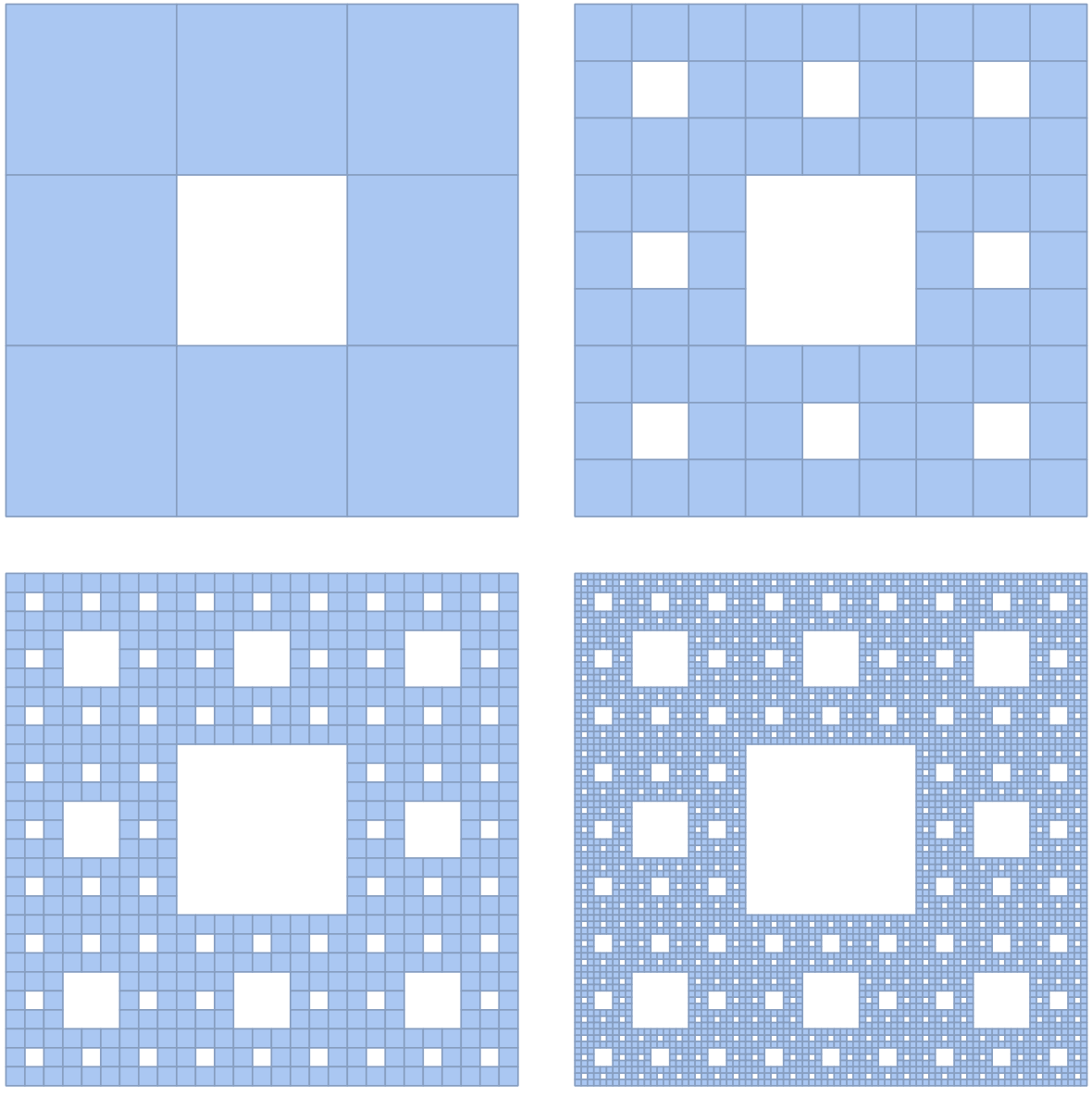
There are several general ideas that are useful to organize computation, perhaps the two most important ones being

- Recursion (self-similarity)
- Iteration (repetition)

Recursion is quite popular and directly supported in many programming languages.

Iteration usually requires some amount of extra work (and, to really make sense, support for functions as first class citizens).


Droste
HAARLEM - HOLLAND
Droste
cacao
cacao
Netto 250 g



Definition

Let $f : A \to A$ be an endofunction. The kth power of f (or kth iterate of f) is defined by induction as follows:

$$f^0 = I_A$$
$$f^k = f \circ f^{k-1}$$

Here I_A denotes the identity function on A and $f \circ g$ denotes composition of functions.

Informally, this just means: compose function f $(k-1)$-times with itself.

$$f^k = \underbrace{f \circ f \circ f \circ \ldots \circ f}_{k \text{ terms}}$$

Without any further knowledge about f there is not much one can say about the iterates f^k. But the following always holds.

Lemma (Laws of Iteration)

- $f^n \circ f = f^{n+1}$
- $f^n \circ f^m = f^{n+m}$
- $(f^n)^m = f^{n \cdot m}$

Exercise

Prove these laws by induction using associativity of composition.

Prof. Dr. Alois Wurzelbrunft* stares at these equations and immediately recognizes a deep analogy to exponentiation.

He also remembers that there is a method for fast exponentiation based on squaring:

$$a^{2e} = (a^e)^2$$
$$a^{2e+1} = (a^e)^2 \cdot a$$

which allows us to compute a^e in $O(\log e)$ multiplications.

Wurzelbrunft's Conclusion:
There is an analogous "fast iteration" method.

*A famous if fictitious professor in the Bavarian hinterland.

> Good mathematicians see analogies between theorems or theories; the very best ones see analogies between analogies.
>
> S. Banach

So is Wurzelbrunft brilliant?

Suppose we want to compute f^{1000}. The obvious way requires 999 compositions of f with itself.

By copying the standard divide-and-conquer approach for fast exponentiation we could try

$$f^{2n} = (f^n)^2$$
$$f^{2n+1} = f \circ (f^n)^2$$

This seems to suggest that we can compute $f^n(x)$ in $O(\log n)$ applications of the basic function f.

After all, it's just like exponentiation, right?

There is an interesting idea here: we would like to take a plain computation

$$C = C_0, C_1, C_2, \ldots, C_{42}, \ldots, C_n$$

and somehow translate it into another computation

$$C' = C'_0, C'_1, \ldots, C'_m$$

such that

- the result is the same, but
- $m \ll n$

Of course, this won't always be possible, but sometimes we might be able to "compress" a computation (by using a smarter algorithm).

Consider the orbit of a under the rational function (this is a clear case of abuse of a Möbius transformation)

$$f(x) = \frac{2 + 2x}{3 + x}$$

One can show that

$$f^t(a) = 1 + \frac{3(a-1)}{1 - a + (a+2)4^t}$$

So there is no need to iterate f, we can simply do the arithmetic.

$$f^5(a) = \cfrac{2 + \cfrac{2\left(2 + \cfrac{2\left(2 + \cfrac{2\left(2+\frac{2(2+2a)}{3+a}\right)}{3+\frac{2+2a}{3+a}}\right)}{3+\cfrac{2+\frac{2(2+2a)}{3+a}}{3+\frac{2+2a}{3+a}}}\right)}{3+\cfrac{2+\cfrac{2\left(2+\frac{2(2+2a)}{3+a}\right)}{3+\frac{2+2a}{3+a}}}{3+\cfrac{2+\frac{2(2+2a)}{3+a}}{3+\frac{2+2a}{3+a}}}}}{3 + \cfrac{2 + \cfrac{2\left(2 + \cfrac{2\left(2+\frac{2(2+2a)}{3+a}\right)}{3+\frac{2+2a}{3+a}}\right)}{3+\cfrac{2+\frac{2(2+2a)}{3+a}}{3+\frac{2+2a}{3+a}}}}{3+\cfrac{2+\cfrac{2\left(2+\frac{2(2+2a)}{3+a}\right)}{3+\frac{2+2a}{3+a}}}{3+\cfrac{2+\frac{2(2+2a)}{3+a}}{3+\frac{2+2a}{3+a}}}}}$$

If the function f in question is linear it can be written as

$$f(x) = M \cdot x$$

where M is a square matrix over some suitable algebraic structure. Then

$$f^t(x) = M^t \cdot x$$

and M^t can be computed in $O(\log t)$ matrix multiplications.

So this is an exponential speed-up over the standard method.

Another important case is when f is a polynomial map

$$f(x) = \sum a_i x^i$$

given by a coefficient vector $\boldsymbol{a} = (a_d, \ldots, a_1, a_0)$.

In this case the coefficient vector for $f \circ f$ can be computed explicitly by substitution. This is useful in particular when computation takes place in a quotient ring such as $R[x]/(x^n - 1)$ so that the expressions cannot blow up.

Again, an exponential speed-up over the standard method.

But Beware of Hasty Conclusions

But we cannot conclude that $f^t(x)$ can always be computed in $O(\log t)$ operations.

The reason fast exponentiation and the examples above work is that we can explicitly compute a representation of $f \circ f$, given the representation of f.

But, in general, there is no fast representation for $f \circ f$, we just have to evaluate f twice.

Just think of f as being given by an executable, a compiled piece of C code. We can wrap a loop around the executable to compute f^t, but that just evaluates f t-times, in the obvious brute-force way. No speed-up whatsoever.

Exercise

Ponder deeply. Assume the speed-up trick always works and figure out what that would mean for complexity theory.

Speaking about hasty conclusions, here is a simple inductively defined sequence of integers.

$$a_1 = 1$$
$$a_n = a_{n-1} + (a_{n-1} \bmod 2n)$$

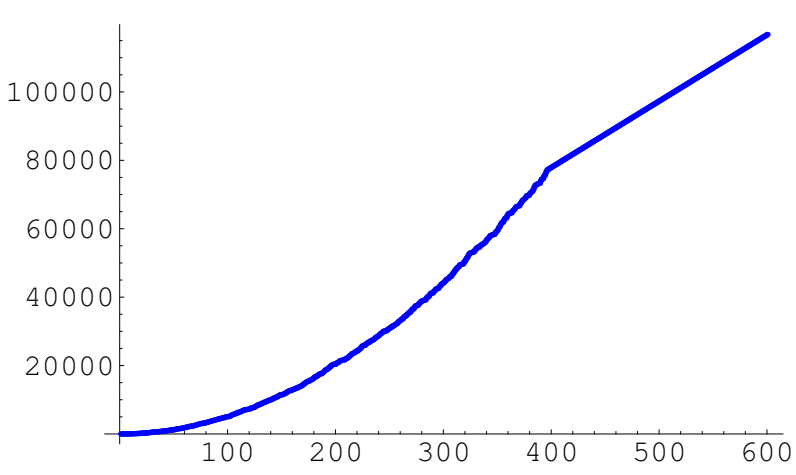
Thus, the sequence starts like so:

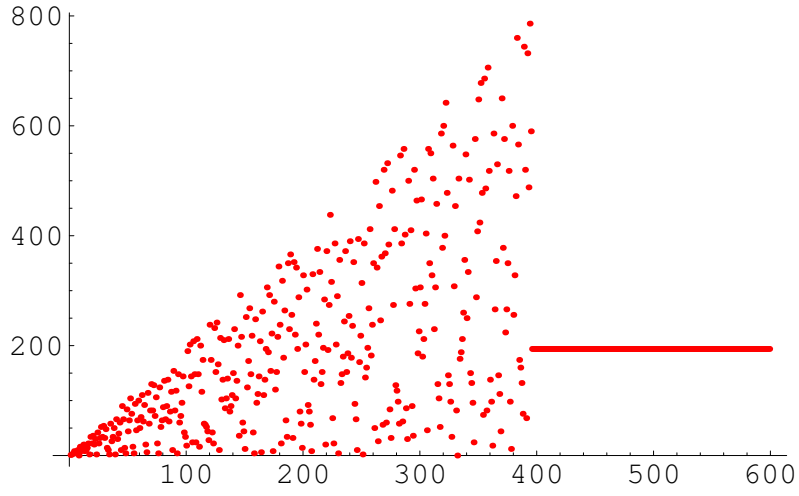
$$1, 2, 4, 8, 16, 20, 26, 36, 36, 52, 60, 72, 92, 100, 110, 124, 146, 148, 182, 204$$

This seems rather complicated. The function appears to be increasing in a somewhat complicated manner.

Alas, there is a rude surprise.

The sequence is ultimately linear: $a_{396+k} = a_{396} + k \cdot 194$ for $k \geq 0$.





The plot on the left is the sequence, on the right (in red) are the forward differences.

Exercise

Figure out why the sequence is ultimately linear.

Here is another strange integer sequence:

$$a_n = \lceil 2/(2^{1/n} - 1) \rceil - \lfloor 2n/\log 2 \rfloor$$

This time, the sequence starts like so:

$$0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots$$

and continues like this for a long, long time, for trillions of terms.

Note that it is not so easy to compute the terms. At any rate, it sure looks like the sequence is constant 0. Alas

$$a_{777\,451\,915\,729\,368} = 1$$

Iteration can be construed as a special case of primitive recursion.

$$F(0, y) = y$$
$$F(x + 1, y) = f(F(x, y))$$

Then $F(x, y) = f^x(y)$.

This is really no more than the standard bottom-up approach to computing an primitive recursive function, expressed in an elegant and concise way.

Conversely, iteration can be used to express recursion. Suppose

$$\begin{aligned} f(0, \boldsymbol{y}) &= g(\boldsymbol{y}) \\ f(x+1, \boldsymbol{y}) &= h(x, f(x, \boldsymbol{y}), \boldsymbol{y}) \end{aligned}$$

Define a new function H by

$$H : \mathbb{N} \times \mathbb{N} \times \mathbb{N}^k \longrightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}^k$$
$$H(x, z, \boldsymbol{y}) = (x+1, h(x, z, \boldsymbol{y}), \boldsymbol{y})$$

Then

$$f(x, \boldsymbol{y}) = \mathsf{snd}(H^x(0, g(\boldsymbol{y}), \boldsymbol{y}))$$

This is perhaps the most natural definition, but if we wanted to we could make H unary by coding everything up as a sequence number.

More precisely, suppose we have some simple basic functions such as

$$x + y \qquad x * y \qquad x \dot{-} y \qquad \mathsf{rt}(x)$$

Here $\mathsf{rt}(x)$ is the integer part of $\sqrt{x}$. These suffice to set up coding machinery, which can then be used to replace primitive recursion by iteration. It suffices to define functions via

$$f(x) = g^x(0)$$

to get the same class as from the recursions above.

Exercise

Come up with a precise version of this statement (define a clone) and give a detailed proof.

Definition

The trajectory or orbit of $a \in A$ under f is the infinite sequence

$$\mathsf{orb}_f(a) = a, f(a), f^2(a), \ldots, f^n(a), \ldots$$

The set of all infinite sequences with elements from A is often written A^ω. Hence the we can think of the trajectory as an operation of type

$$(A \to A) \times A \to A^\omega$$

that associates a function on A and element in A with an infinite sequence over A.

Sometimes one is not interested in the actual sequence of points but rather in the set of these points:

$$\{ f^i(a) \mid i \geq 0 \}$$

While the sequence is always infinite, the underlying set may well be finite, even when the carrier set is infinite.

In a sane world one would refer to the sequences as trajectories, and use the term orbit for the underlying sets. Alas, in the literature the two notions are hopelessly mixed up.

So, when we refer to a "trajectory" we will always mean the sequence, but, bending to custom, we will use "orbit" for both.

Here is a clever definition due to Dedekind: given an endofunction f and a point a, the corresponding chain is defined to be

$$\bigcap \{ X \subseteq A \mid a \in X, f(X) \subseteq X \}$$

Thus, the chain is the least set that contains a and is closed under f. That is exactly the orbit of a under f, considered as a set.

Who cares?

Dedekind's definition does not require the natural numbers. In fact, it can be used to define them. In Dedekind's view, this means that arithmetic can be reduced to logic.

Here is how. Suppose we have a function $f : A \to A$ and a point $a \in A$ such that

- f is an injection,
- a is not in the range of f,
- A is the chain of f and a.

Dedekind calls these sets simply infinite.

We can think of a as 0 and, more generally, we can think of $f^n(a)$ as n.

So this is a way of describing the natural numbers, the smallest infinite set, without any hidden references to the naturals.

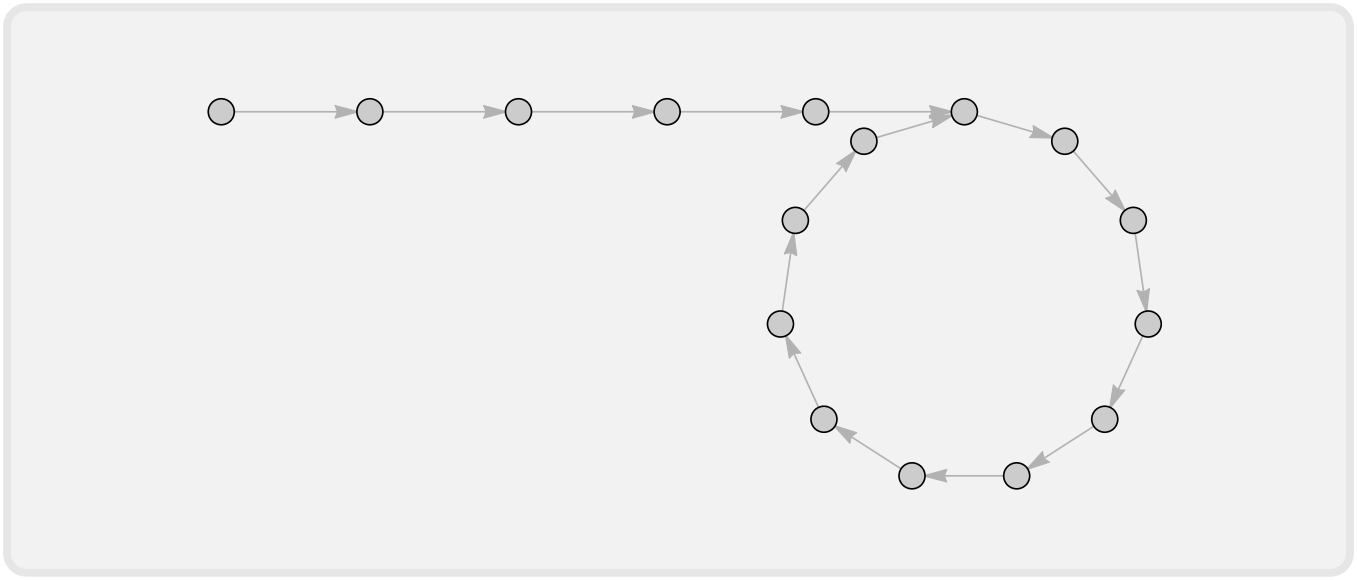
According to Dedekind, the chain C defined by f and a has the form

$$C = \bigcap \{\, X \subseteq A \mid a \in X, f(X) \subseteq X \,\}$$

But note that C is one of the X's on the right hand side. So there is some (non-vicious) circularity in this approach. Most mathematicians would not bat an eye when confronted with definitions like this one, they are totally standard.

And the payoff is huge. For example, when Bernstein told Dedekind about his correct proof of the "Cantor-Schröder-Bernstein" theorem, he was shocked to hear that Dedekind had a much better proof, based on his chains.

At any rate, if the carrier set is finite, all trajectories must ultimately wrap around and all orbits must be finite:



What changes is only the length of the transient part and the length of the cycle (in the picture 5 and 11).

Definition

Let $f : A \to A$ be an endofunction.

- $a \in A$ is a fixed point of f if $f(a) = a$.
- A sequence $a_0, \ldots, a_{n-1}$ in A is a cycle of f if $f(a_i) = a_{i+1 \bmod n}$.
- A cycle of length n is also called an n-cycle.
- The orbit of a under f is periodic if $\exists\, p > 0 : f^p(a) = a$.
- The orbit of a under f is ultimately periodic $\exists\, t \geq 0, p > 0 : f^{t+p}(a) = f^t(a)$.

Cycles and fixed points are closely related:

$a_0, \ldots, a_{n-1}$ is an n-cycle of f iff a_0 is a fixed point of f^n.

If A is finite, then any orbit of $f : A \to A$ must be ultimately periodic:

$$f^t(x) = f^{t+p}(x)$$

for some $t \geq 0$, $p > 0$, which values depend on x.

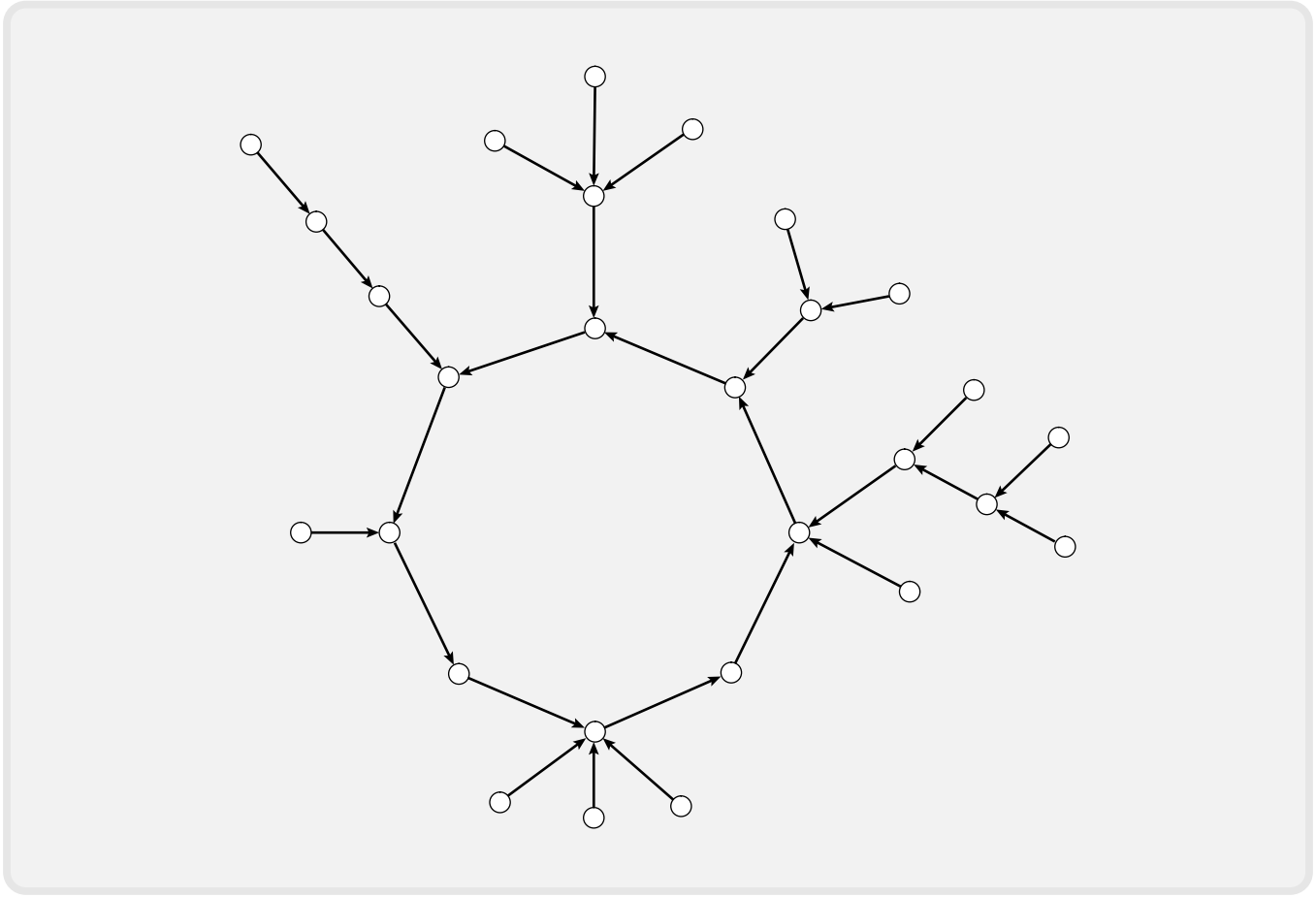
Definition

The least t and p such that $f^t(x) = f^{t+p}(x)$ is the transient length and the period length of the orbit of x (wrt. f).

Thus, an orbit is periodic iff the transient is 0.

Also, a function on a finite set has only transients of length 0 iff the function is injective iff it is a permutation.

The lasso shows the general shape of any single orbit, but in general orbits overlap. All orbits with the same limit cycle are called a basin of attraction in dynamics.



As the last picture shows, it is natural to think of f as a directed graph on the carrier set where the edges indicate the action of f.

Definition

The functional digraph (or diagram) of $f : A \to A$ is defined as $G_f = \langle A, E \rangle$ where $E = \{ (x, f(x)) \mid x \in A \}$.

Note that every vertex in G_f has outdegree 1, but indegrees may vary.

The non-trivial strongly connected components of the digraph are the limit cycles of the function. The weakly connected components are the basins of attraction.

There are several natural parameters associated with the digraph that provide useful information about the function in question.

- Indegree. If all nodes have the same indegree k the function is k-to-1. Otherwise, determine the maximum/minimum indegree, the distribution of values.
- Periods. Count the number of limit cycles, and their length.
- Transients. Determine the length of the transients leading to limit cycles.

At least when the carrier set is finite we would like to be able to determine these parameters easily. Alas, even for relatively simple maps this turns out to be rather difficult.

The geometric perspective afforded by the diagram also suggests to study path-existence problems.

Definition

Let f be a function on A and $a, b \in A$ two points in A. Then point b is reachable from a if for some $i \geq 0$:

$$f^i(a) = b$$

In other words, point y belongs to the orbit of x.

Proposition

Reachability is reflexive and transitive but in general not symmetric.

Reachability is symmetric when A is finite and f injective (and therefore a permutation): each orbit then is a cycle and forms an equivalence class.

Definition

Let f be a function on A and $a, b \in A$ two points in A. Points a and b are confluent if for some $i, j \geq 0$:

$$f^i(a) = f^j(b)$$

In other words, the orbits of a and b merge, they share the same limit cycle (which may be infinite and not really a cycle).

Reachability implies confluence but not conversely. For finite carrier sets reachability is the same as confluence iff the map is a bijection.

Proposition

Confluence is an equivalence relation.

Reflexivity and symmetry are easy to see, but transitivity requires a little argument.

Let $f^i(x) = f^j(y)$ and $f^k(y) = f^l(z)$, assume $j \leq k$. Then with $d = k - j \geq 0$ we have

$$f^{i+d}(x) = f^{j+d}(y) = f^k(y) = f^l(z).$$

Each equivalence class contains exactly one cycle of f, and all the points whose orbits lead to this cycle – just as in the last picture.

How do we compute the transient t and period p of the orbit of $a \in A$ under $f : A \to A$ for finite carrier sets A?

The obvious brute force approach is to use a container to keep track of everything we have already seen:

$$a, f(a), f^2(a), \ldots, f^i(a)$$

and then to compare $f^{i+1}(a)$ to all these previous values.

In most cases, the data structure of choice is a hash table or tree: we can check whether $f^{i+1}(a)$ is already present in expected constant time or logarithmic time, respectively. Memory requirement is linear in the size of the orbit assuming the elements in A require constant space (a fairly safe assumption, if the elements are big use pointers).

A (simplified version of a) classical problem from the early days of Lisp: Suppose we have a pointer-based linked list structure in memory and we want to check if there are any cycles in the structure (as opposed to having all lists end in `nil`).

We can think of this as an orbit problem:

- A is the set of all nodes of the structure,
- $f(x) = y$ means there is a pointer from x to y.

The Problem:

Suppose further the structure consumes 90% of memory, so we cannot afford to build a large hash table or tree.

Can we compute transients and periods in $O(1)$ space?

At first glance, this may seem quite impossible: if we forget already discovered elements we *obviously* cannot detect cycles. Right?

Not at all: we have an element $b = f^t(a)$, and we want to check if it is new.

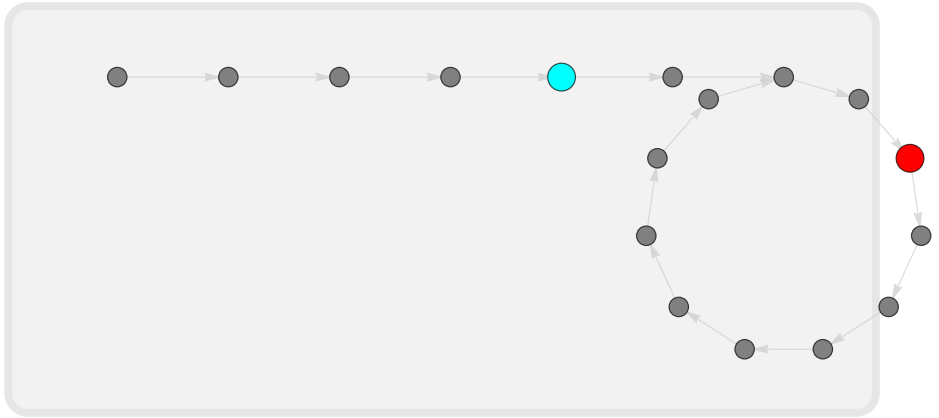
We can simply compare b to all $f^s(a)$ for $s < t$.

This requires an absurd amount of recomputation and is thus highly inefficient, but it trivially works and it uses only constant memory.

The method is actually quite simple: instead of storing an object, we recompute whenenver necessary.

Here is a better way to handle the time/space tradeoff: race two pebbles down the orbit.

```
u = f(a);
v = f(u);
while( u != v ) {
    u = f(u);
    v = f(f(v));
}
```



Claim

Upon termination, $u = v$ is a position on the cycle.

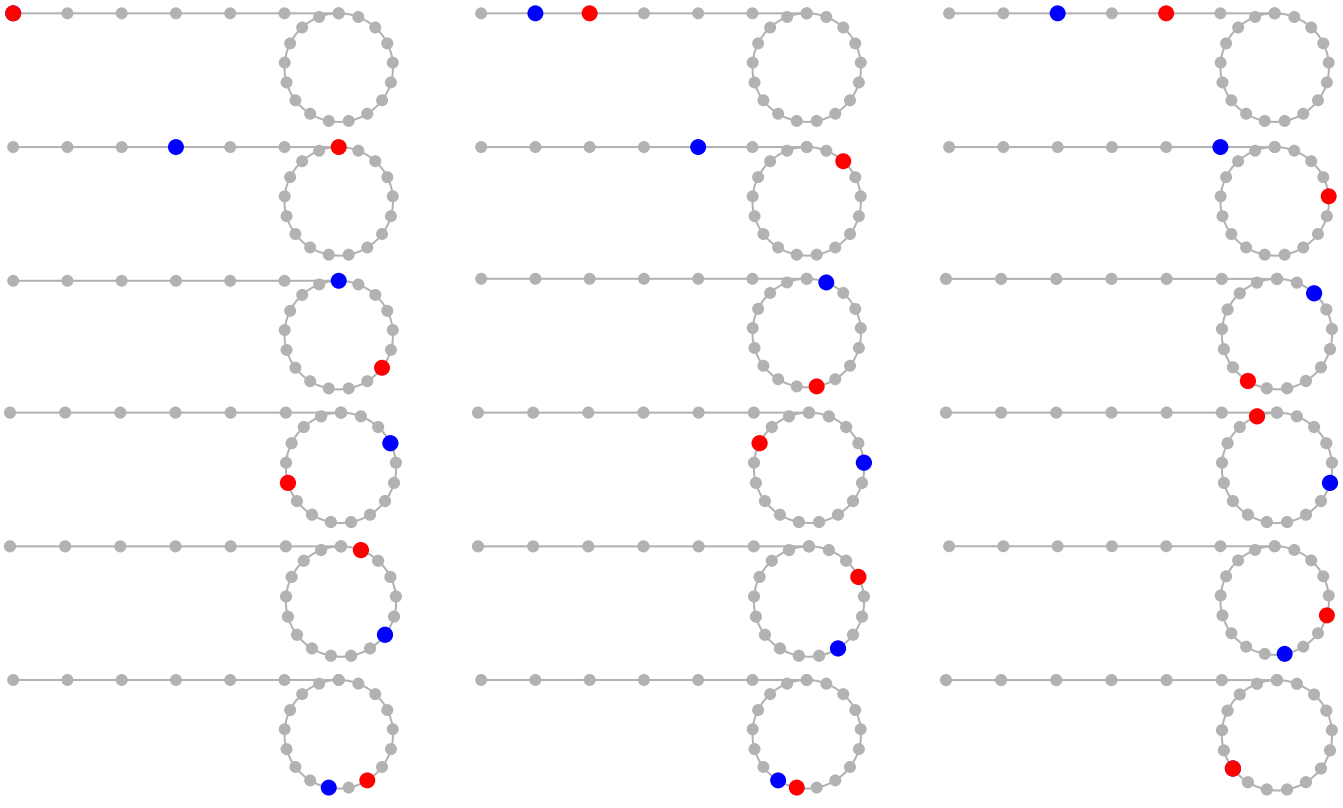
Think of two pebbles u and v, moving at speed 1 and 2, respectively.

The slow pebble u enters the limit cycle at time t, the transient, when the fast pebble v is already there. From now on, v gains one place on u at each step. So pebble v must catch up at time s where $s \leq t + p$, where p is the period. The meeting time is called the Floyd-time.

Once we have a foothold on the cycle it is not hard to compute transient and period, see below.

One can make a nice movie out of this. OK, it is pretty boring after all, but what do you expect.

Here the transient is 6, and the period 17.



The Floyd-time here is 17.

One can also write out a simple table of the process. Here we think of the points on the orbit as $-\tau, \ldots, -1, 0, 1, \ldots, \pi - 1$. To avoid visual clutter, we write $-k$ as $\underline{k}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\underline{6}$	$\underline{5}$	$\underline{4}$	$\underline{3}$	$\underline{2}$	$\underline{1}$	0	1	2	3	4	5	6	7	8	9	10	**11**
$\underline{6}$	$\underline{4}$	$\underline{2}$	**0**	2	4	6	8	10	12	14	16	1	3	5	7	9	**11**

Not as pretty, but potentially more useful. Note that when the slow pebbles enters the cycle at time 6, the fast one is in position 6. $6 + 11 = 17$.

How about the Period?

Suppose we already have a point b on the cycle.

```
t = 1;
u = f(b);

while( u != b ) {
  u = f(u);
  t++;
}
return t;
```

We walk around the cycle, and count steps.

Suppose we already know p, the period.

```
t = 0;
u = a;
v = iterate( f, a, p );   // v = f^p(a)
while( u != v ) {
  u = f(u);
  v = f(v);
  t++;
}
return t;
```

v has a headstart of p. So, when u first enters the cycle, v has just gone around once, and they meet at the contact point.

Let us assume f to be computable in time $O(1)$ and elements of the carrier set A to take space $O(1)$.

Theorem

One can determine the transient t and period p of a point in A under f in time $O(t+p)$, and space $O(1)$.

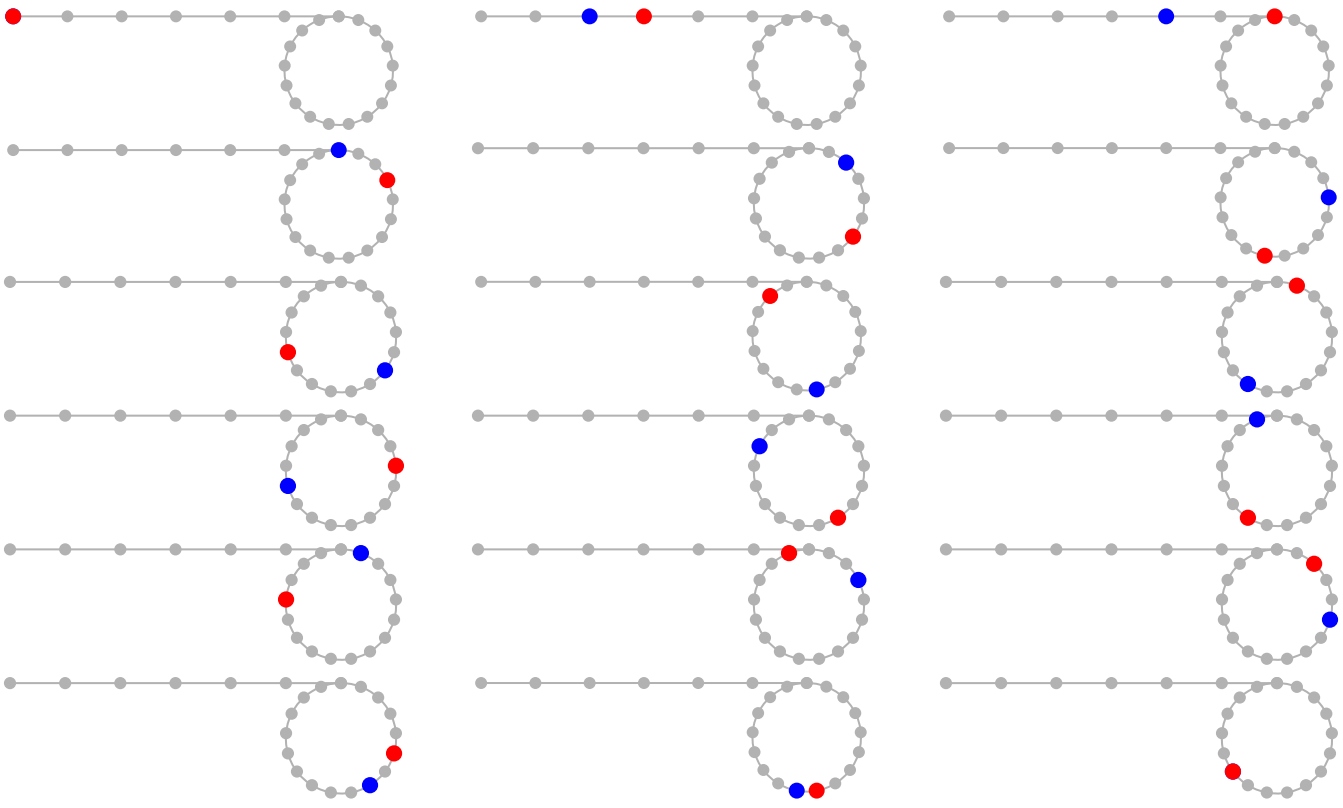
Linear time cannot be avoided in general (why?), so this is optimal.

Floyd's cycle finding algorithm is an excellent general purpose tool in particular when the evaluation of the function in question is cheap.

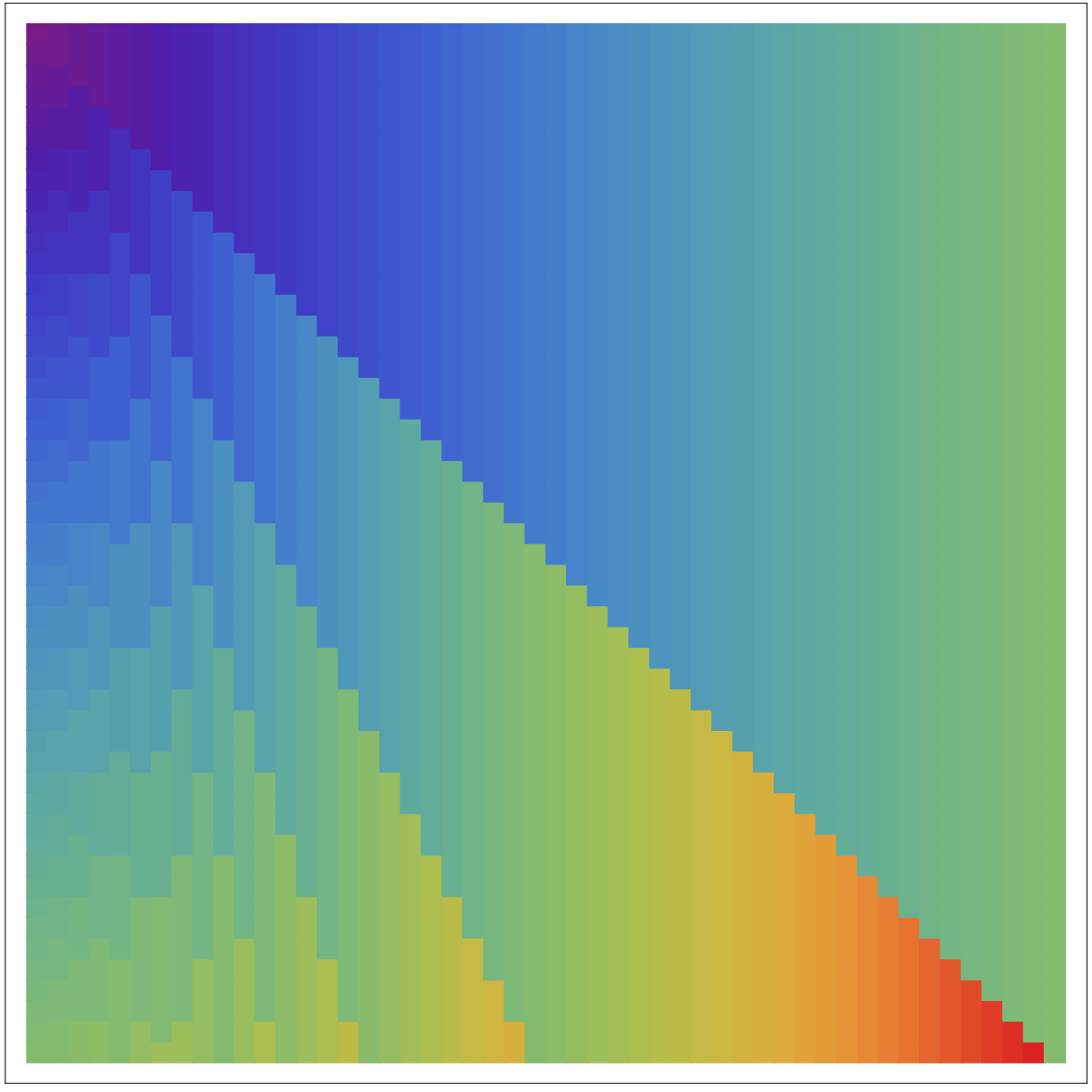
But note that in the special case where the function is known to be a permutation on a finite domain there is, of course, no need to use Floyd's or similar cycle finding algorithms: since the components of the diagram are all cycles we can simply trace a cycle once to determine its length. So the natural method to compute cycle length is automatically memoryless (if we assume the objects in question can be stored in constant space).

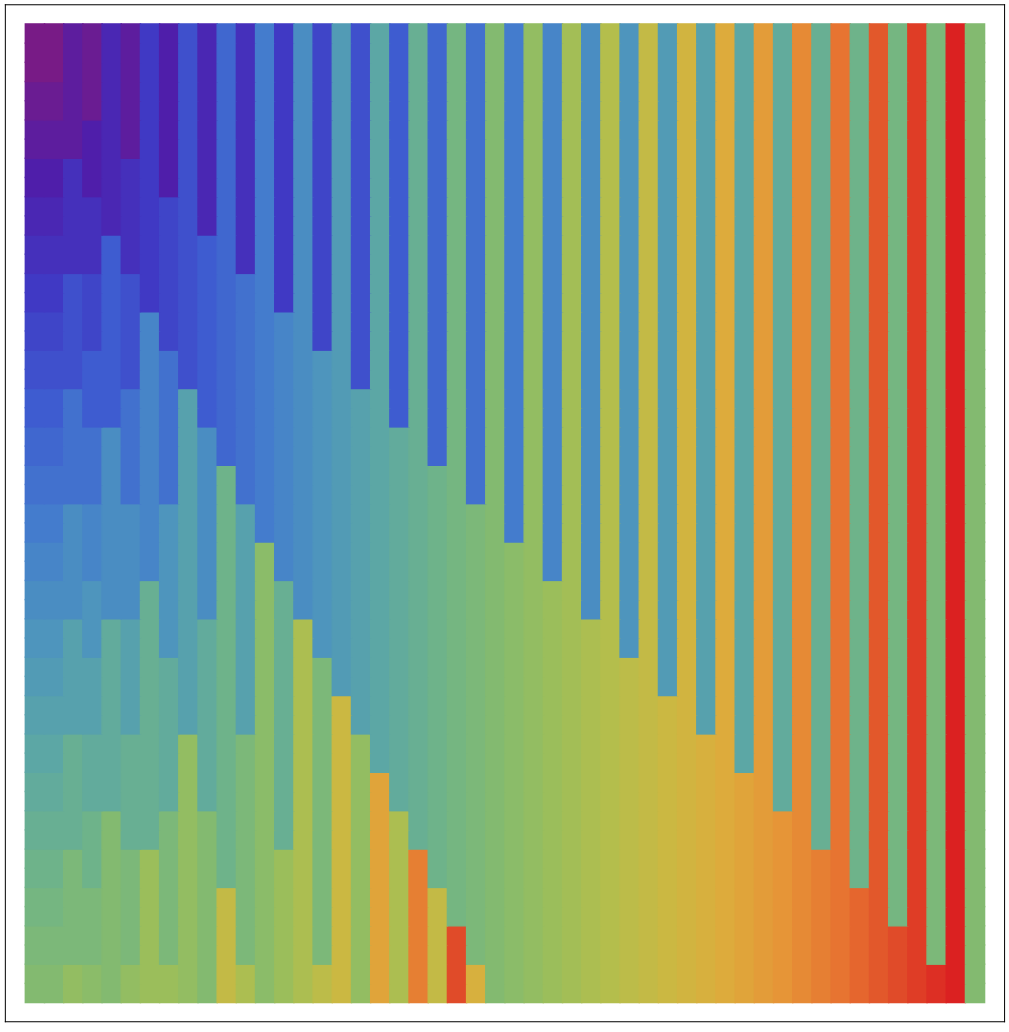
Incidentally, determining cycle lengths of permutations is very important for some advanced counting methods, more later.

It is tempting to try different pebble speeds. Here is transient 6, period 17, pebbles at speeds 2 and 3, respectively.



Surprisingly, the pebbles meet at time 17, just like the ordinary algorithm. Ponder deeply.





This uses speeds 2 and 4. Seems fairly complicated.

Exercise

What would happen to the Floyd-time if we changed the pebble speeds to u and v, where $1 \leq u < v$? Would the algorithm even work for all transients and period?

Exercise

Try to find an algebraic way to compute the Floyd-time directly from the parameters τ and π. Do this for the $(1, 2)$ version first, then generalize to speeds $u < v$.

Exercise

Call the place where the pebbles meet the Floyd-point. Study it.

Here is a method to compute the period using "teleportation."

```
slow = a;
fast = f(a);
cnt = pow2 = 1;
while( fast != slow )
     if( cnt == pow2 )
       { slow = fast; cnt = 0; pow2 *= 2; }
     fast = f(fast);
     cnt++;

return cnt;
```

Exercise

Figure out how this works. Compare its performance to Floyd's method.

There is another algorithm due to Nivasch that uses a bit of extra memory (a small stack) to speed up the search.

Applications:

- discrete dynamical systems (such as cellular automata)
- Pollard's factorization method
- analysis of hash functions

We have already seen that iteration can produce very rapidly growing functions (much like recursion). Here is another example where iteration produces a rather perplexing result: every orbit ends in fixed point 0, though it looks like it should diverge towards infinity.

Suppose we write a number in base 2, say

$$266 = 2^8 + 2^3 + 2$$

We can turn this into the complete binary expansion by writing the exponents also in base 2, and so on.

$$266 = 2^{2^{2+1}} + 2^{2+1} + 2$$

where we really should write 2^0 instead of 1, but c'mon.

Now suppose we replace 2 in the representation everywhere by 3:

$$3^{3^{3+1}} + 3^{3+1} + 3$$

Unsurprisingly, this new number is much larger:

$$443426488243037769948249630619149892887 \approx 4 \times 10^{38}$$

Next, we write this number in complete base 3, and bump the base to 4. We get something like 3×10^{616}.

Then we write this number in complete base 4 and bump to 5 . . .

Obviously, this process leads to a very rapidly increasing sequence of numbers.

Now suppose we follow the base bump by subtracting 1, so the result will be a tiny little bit smaller than with a pure base bump. Call such a sequence a Goodstein sequence.

We expect Goodstein sequences to diverge since the base bump causes a huge increase, subtracting 1 should really not matter much. Alas ...

Theorem

All Goodstein sequences converge to 0.

Example?

It is very hard to come up with good examples.

Starting at 2 and 3 we get the short sequences

$$2, 2, 1, 0$$
$$3, 3, 3, 2, 1, 0$$

But starting at 4 things spin out of control already:

$$4, 26, 41, 60, 83, 109, 139, 173, 211, 253, 299, \ldots, 402653183^2, \ldots$$

It takes some $10^{121,210,695}$ steps to get to 0!

Goodstein's proof of his theorem uses ordinals and is extremely elegant.

More precisely, Goodstein's argument uses induction up to

$$\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$$

But induction to ϵ_0 is enough to prove that Peano arithmetic is consistent. It is thus not too surprising that (PA) cannot handle Goodstein's theorem.

Theorem

Goodstein's theorem is not provable in Peano arithmetic.

But Why?

Why does this really work? Because once the base is sufficiently large, we keep chipping away at the constant term until we ultimately have to borrow from one of the previous terms.

Write $G_b(n)$ for the term in the sequence that will undergo a bump from base b to $b+1$. Let $b = 402653183$. Then

$$\begin{aligned} G_b(4) &= b^2 \\ G_{b+1}(4) &= b(b+1) + b \\ G_{b+2}(4) &= b(b+2) + b - 1 \\ G_{b+3}(4) &= b(b+3) + b - 2 \\ &\cdots \\ G_{2b+1}(4) &= b(2b+1) \end{aligned}$$

Ponder deeply.

So the convergence proof is very hard in a sense, but note that the stopping time function

$$n \mapsto \min\Big(b \mid G_b(n) = 0 \Big)$$

is trivially computable: just compute the damn sequence.

Again: Computable functions can be monsters, even when they are total.

How do we compute the transient t and period p of the orbit of $a \in A$ under $f : A \to A$ for finite carrier sets A?

The obvious brute force approach is to use a container to keep track of everything we have already seen:

$$a, f(a), f^2(a), \ldots, f^i(a)$$

and then to compare $f^{i+1}(a)$ to all these previous values.

In most cases, the data structure of choice is a hash table or tree: we can check whether $f^{i+1}(a)$ is already present in expected constant time or logarithmic time, respectively. Memory requirement is linear in the size of the orbit assuming the elements in A require constant space (a fairly safe assumption, if the elements are big use pointers).

A (simplified version of a) classical problem from the early days of Lisp: Suppose we have a pointer-based linked list structure in memory and we want to check if there are any cycles in the structure (as opposed to having all lists end in `nil`).

We can think of this as an orbit problem:

- A is the set of all nodes of the structure,
- $f(x) = y$ means there is a pointer from x to y.

The Problem:

Suppose further the structure consumes 90% of memory, so we cannot afford to build a large hash table or tree.

Can we compute transients and periods in $O(1)$ space?

At first glance, this may seem quite impossible: if we forget already discovered elements we *obviously* cannot detect cycles. Right?

Not at all: we have an element $b = f^t(a)$, and we want to check if it is new.

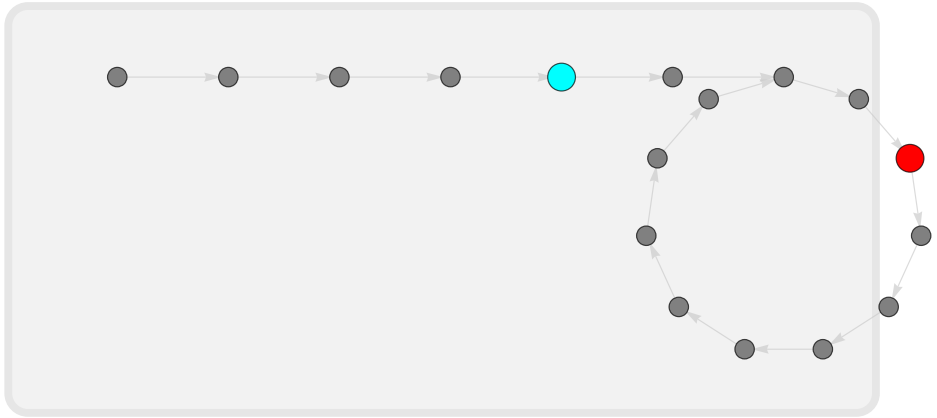
We can simply compare b to all $f^s(a)$ for $s < t$.

This requires an absurd amount of recomputation and is thus highly inefficient, but it trivially works and it uses only constant memory.

The method is actually quite simple: instead of storing an object, we recompute whenenver necessary.

Here is a better way to handle the time/space tradeoff: race two pebbles down the orbit.

```
u = f(a);
v = f(u);
while( u != v ) {
    u = f(u);
    v = f(f(v));
}
```



Claim

Upon termination, $u = v$ is a position on the cycle.

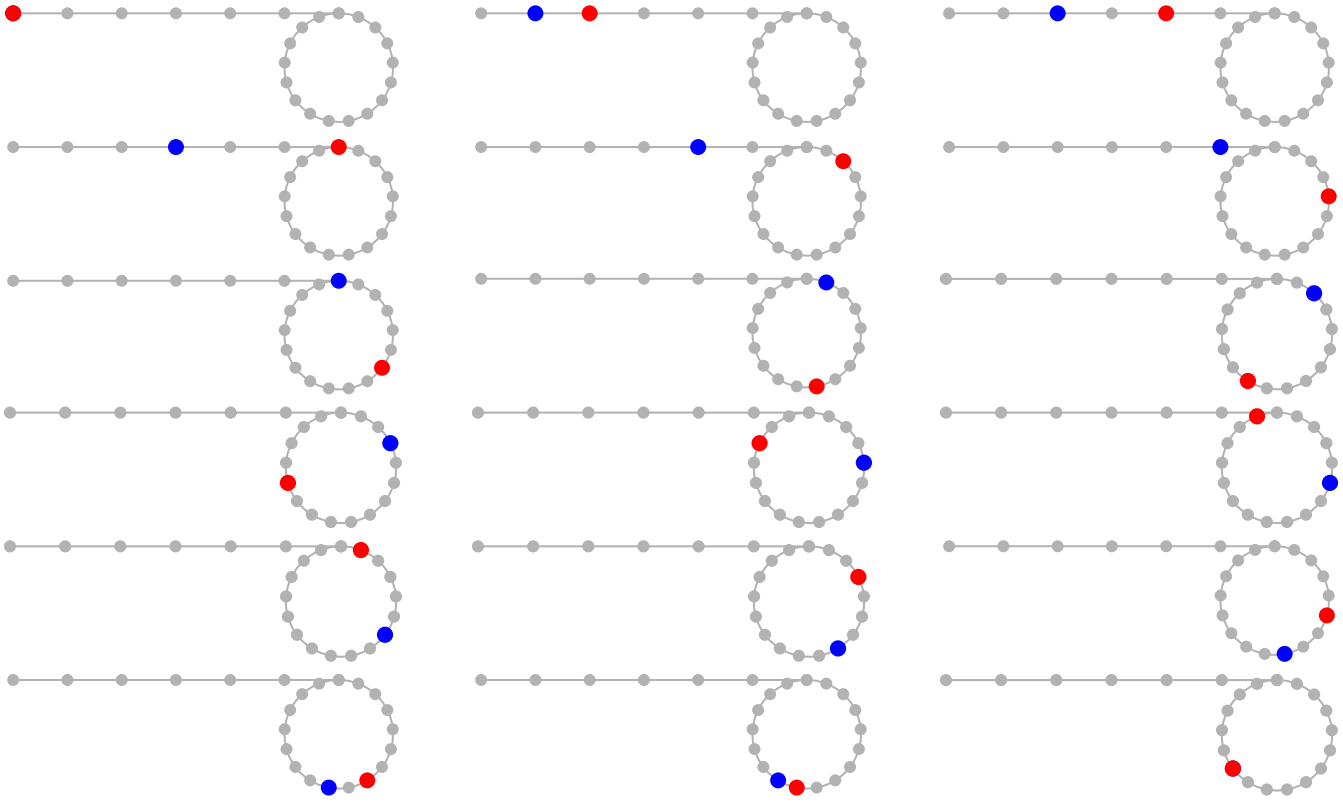
Think of two pebbles u and v, moving at speed 1 and 2, respectively.

The slow pebble u enters the limit cycle at time t, the transient, when the fast pebble v is already there. From now on, v gains one place on u at each step. So pebble v must catch up at time s where $s \leq t + p$, where p is the period. The meeting time is called the Floyd-time.

Once we have a foothold on the cycle it is not hard to compute transient and period, see below.

One can make a nice movie out of this. OK, it is pretty boring after all, but what do you expect.

Here the transient is 6, and the period 17.



The Floyd-time here is 17.

One can also write out a simple table of the process. Here we think of the points on the orbit as $-\tau, \ldots, -1, 0, 1, \ldots, \pi - 1$. To avoid visual clutter, we write $-k$ as $\underline{k}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\underline{6}$	$\underline{5}$	$\underline{4}$	$\underline{3}$	$\underline{2}$	$\underline{1}$	0	1	2	3	4	5	6	7	8	9	10	**11**
$\underline{6}$	$\underline{4}$	$\underline{2}$	**0**	2	4	6	8	10	12	14	16	1	3	5	7	9	**11**

Not as pretty, but potentially more useful. Note that when the slow pebbles enters the cycle at time 6, the fast one is in position 6. $6 + 11 = 17$.

How about the Period?

Suppose we already have a point b on the cycle.

```
t = 1;
u = f(b);

while( u != b ) {
  u = f(u);
  t++;
}
return t;
```

We walk around the cycle, and count steps.

Suppose we already know p, the period.

```
t = 0;
u = a;
v = iterate( f, a, p );   // v = f^p(a)
while( u != v ) {
  u = f(u);
  v = f(v);
  t++;
}
return t;
```

v has a headstart of p. So, when u first enters the cycle, v has just gone around once, and they meet at the contact point.

Let us assume f to be computable in time $O(1)$ and elements of the carrier set A to take space $O(1)$.

Theorem

One can determine the transient t and period p of a point in A under f in time $O(t+p)$, and space $O(1)$.

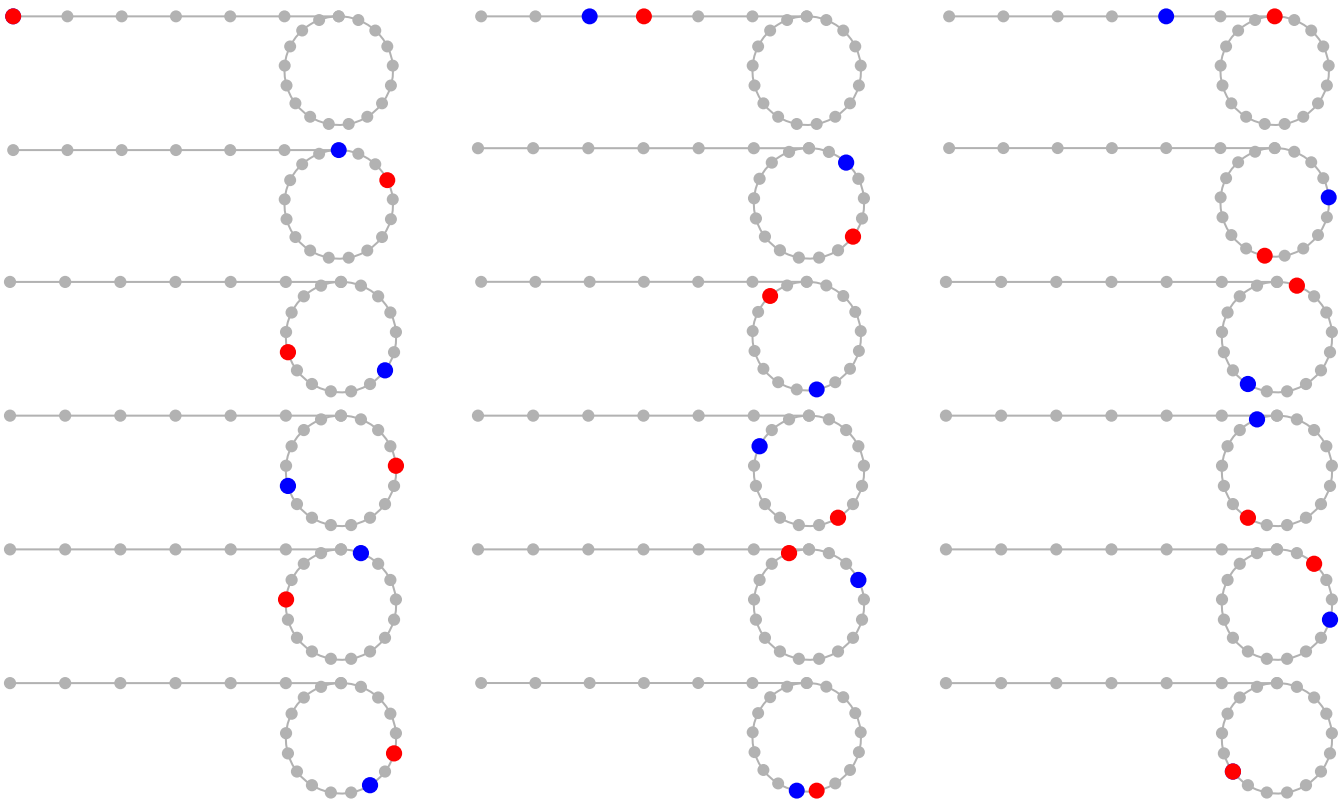
Linear time cannot be avoided in general (why?), so this is optimal.

Floyd's cycle finding algorithm is an excellent general purpose tool in particular when the evaluation of the function in question is cheap.

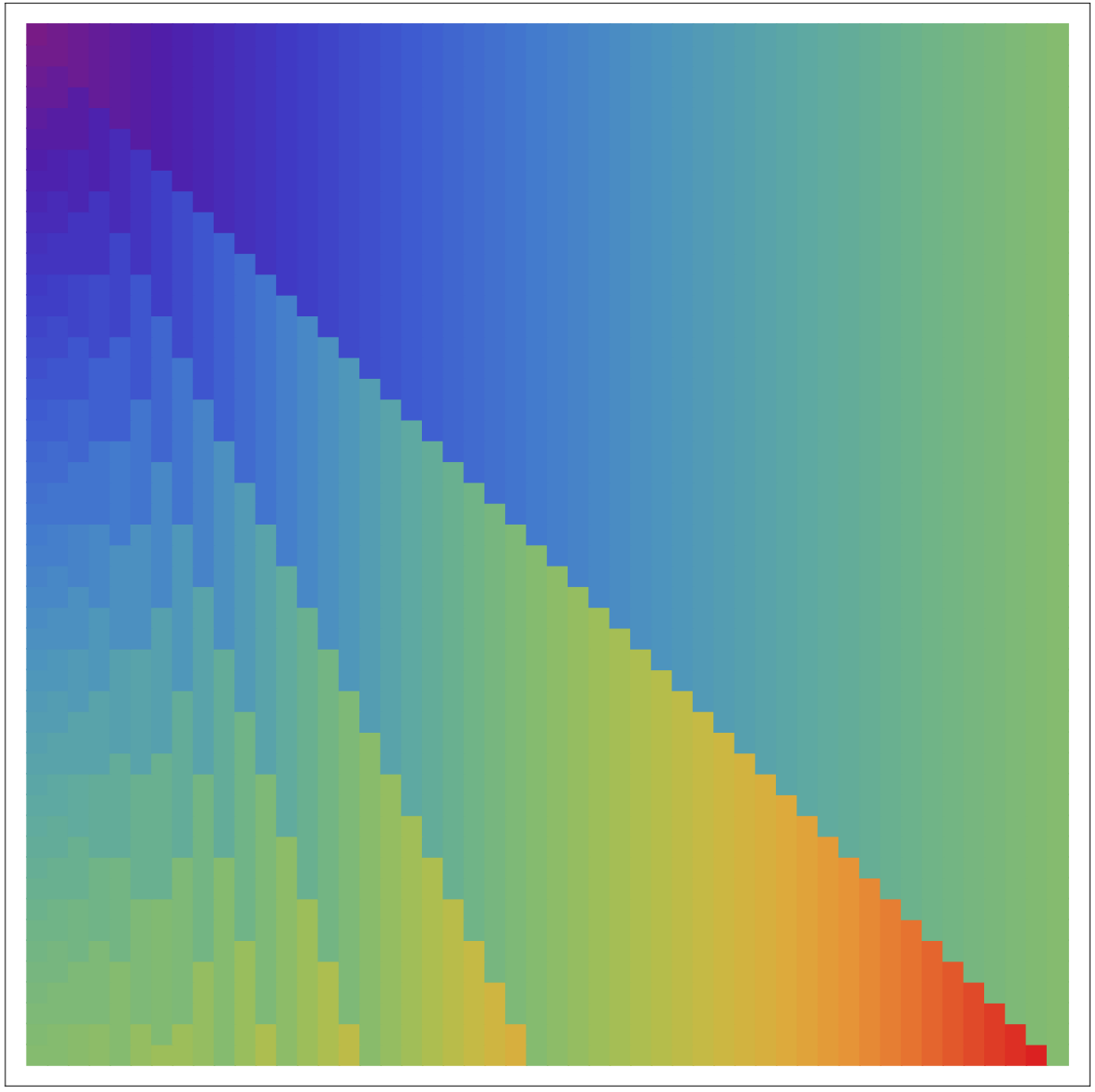
But note that in the special case where the function is known to be a permutation on a finite domain there is, of course, no need to use Floyd's or similar cycle finding algorithms: since the components of the diagram are all cycles we can simply trace a cycle once to determine its length. So the natural method to compute cycle length is automatically memoryless (if we assume the objects in question can be stored in constant space).

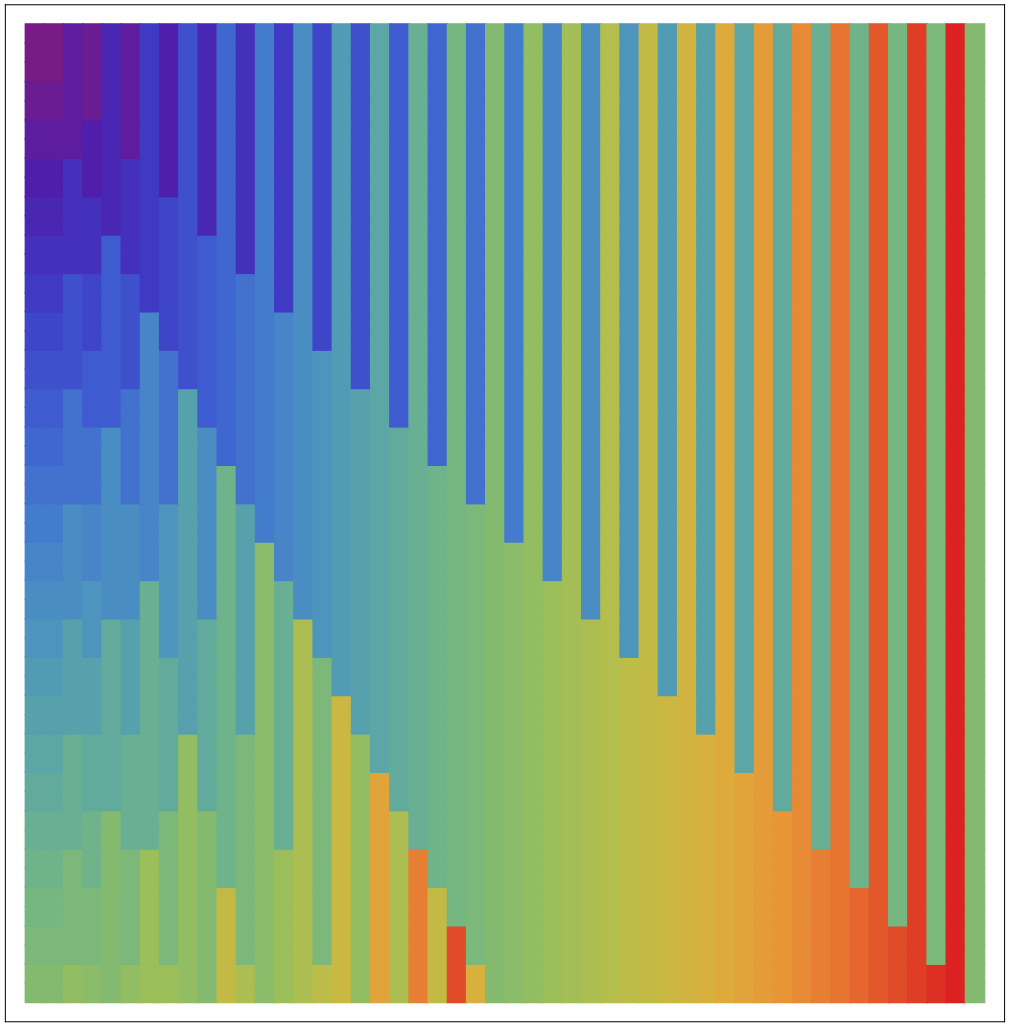
Incidentally, determining cycle lengths of permutations is very important for some advanced counting methods, more later.

It is tempting to try different pebble speeds. Here is transient 6, period 17, pebbles at speeds 2 and 3, respectively.



Surprisingly, the pebbles meet at time 17, just like the ordinary algorithm. Ponder deeply.





This uses speeds 2 and 4. Seems fairly complicated.

Exercise

What would happen to the Floyd-time if we changed the pebble speeds to u and v, where $1 \leq u < v$? Would the algorithm even work for all transients and period?

Exercise

Try to find an algebraic way to compute the Floyd-time directly from the parameters τ and π. Do this for the $(1, 2)$ version first, then generalize to speeds $u < v$.

Exercise

Call the place where the pebbles meet the Floyd-point. Study it.

Here is a method to compute the period using "teleportation."

```
slow = a;
fast = f(a);
cnt = pow2 = 1;
while( fast != slow )
     if( cnt == pow2 )
       { slow = fast; cnt = 0; pow2 *= 2; }
     fast = f(fast);
     cnt++;

return cnt;
```

Exercise

Figure out how this works. Compare its performance to Floyd's method.

There is another algorithm due to Nivasch that uses a bit of extra memory (a small stack) to speed up the search.

Applications:

- discrete dynamical systems (such as cellular automata)
- Pollard's factorization method
- analysis of hash functions