CDM

Iteration

Klaus Sutner

Carnegie Mellon University

Spring 2025

Computational Memes

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

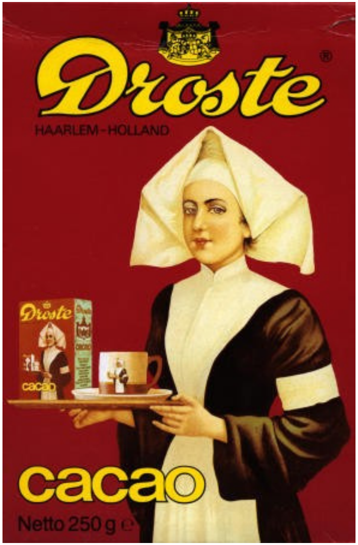
- Recursion (top-down, declarative)
- Iteration (bottom-up, imperative)

Recursion is quite popular and can be used directly as an elegant model of computation (Herbrand-Gödel equations).

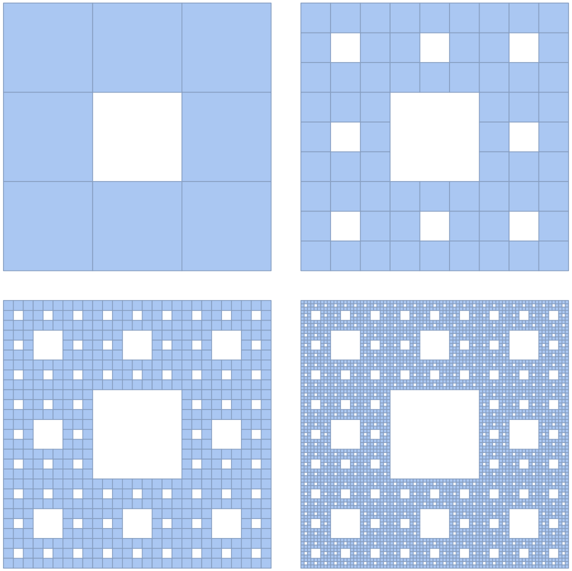
Iteration is equally powerful, but usually requires extra work (and may be more efficient, in time and space).

Both are predicated on the notion of self-similarity.

Droste Effect



Sierpinski Carpet



Iteration

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 with itself, $(k-1)$-times.

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

General Laws

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.

Fase Exponentiation

Exponentiation is a standard example of iteration:

$$a^n = \big(\lambda z \,.\, a \cdot z\big)^n(1)$$

There is a standard speedup 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^n in $O(\log n)$ multiplications.

This is used everywhere in computer algebra.

Wurzelbrunft's Idea

Prof. Dr. Alois Wurzelbrunft[†] stares at the iteration equations and immediately recognizes a deep analogy to exponentiation.

Wurzelbrunft's Conjecture:
There is a "fast iteration" method analogous to fast exponentiation.

So we can compute $f^n(a)$ in $O(\log n)$ applications of f.

[†]A famous if fictitious professor in the Bavarian hinterland. Well-known for his unconvential and often controversial ideas.

Aside

A mathematician is a person who can find analogies between theorems; a better mathematician is one who can see analogies between proofs and the best mathematician can notice analogies between theories. One can imagine that the ultimate mathematician is one who can see analogies between analogies.

S. Banach

So is Wurzelbrunft brilliant?

Fast Iteration

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 would try to exploit the equations

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

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

What could possibly go wrong?

Computational Compressibility

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). Computational hardness is just a reference to incompressible computations.

Closed Forms

Consider the orbit of a under the rational function (a clear abuse of a Möbius transformation):

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

A little fumbling shows that

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

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

But if you insist . . .

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

Fibonacci Numbers

The obvious bottom-up method to compute F_n requires essentially n additions (we will ignore the growing size of the numbers).

We can exploit matrix multiplication to get an alternative description:

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \qquad \text{implies} \qquad M^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$$

Using standard fast exponentiation we obtain M^n in only $\log n$ matrix multiplications (8 integer multiplications plus 4 integer additions).

Asymptotically this is much faster, even though multiplication is substantially more expensive than addition.

Linear Maps

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.

Polynomials

Another important case is when f is a polynomial

$$f(x) = \sum_{i \leq d} a_i x^i$$

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

Squaring comes down to computing the coefficients for $f(f(x))$. This turn out to be a bit messy.

$$b_\ell = \sum_{k \leq d} a_k \sum_{\boldsymbol{m}} \frac{k!}{m_0! \ldots m_d!} \prod_{i \leq d} a_i^{m_i}$$

Here the middle sum ranges of all $\boldsymbol{m} = m_0, \ldots, m_d$ such that $\sum m_i = k$ and $\sum i\, m_i = \ell$.

Really?

$$a_0+a_0a_1+xa_1^2+a_0^2a_2+x^2a_1a_2+2xa_0a_1a_2+x^2a_1^2a_2+2x^2a_0a_2^2+2x^3a_1a_2^2+x^4a_2^3+a_0^3a_3+x^3a_1a_3+3xa_0^2$$
$$a_1a_3+3x^2a_0a_1^2a_3+x^3a_1^3a_3+2x^3a_0a_2a_3+3x^2a_0^2a_2a_3+2x^4a_1a_2a_3+6x^3a_0a_1a_2a_3+3x^4a_1^2a_2a_3+2x^5$$
$$a_2^2a_3+3x^4a_0a_2^2a_3+3x^5a_1a_2^2a_3+x^6a_2^3a_3+3x^3a_0^2a_3^2+6x^4a_0a_1a_3^2+3x^5a_1^2a_3^2+x^6a_2a_3^2+6x^5a_0a_2$$
$$a_3^2+6x^6a_1a_2a_3^2+3x^7a_2^2a_3^2+3x^6a_0a_3^3+3x^7a_1a_3^3+3x^8a_2a_3^3+x^9a_3^4+a_0^4a_4+x^4a_1a_4+4xa_0^3a_1a_4+6x^2$$
$$a_0^2a_1^2a_4+4x^3a_0a_1^3a_4+x^4a_1^4a_4+2x^4a_0a_2a_4+4x^2a_0^3a_2a_4+2x^5a_1a_2a_4+12x^3a_0^2a_1a_2a_4+12x^4a_0a_1^2$$
$$a_2a_4+4x^5a_1^3a_2a_4+2x^6a_2^2a_4+6x^4a_0^2a_2^2a_4+12x^5a_0a_1a_2^2a_4+6x^6a_1^2a_2^2a_4+4x^6a_0a_2^3a_4+4x^7a_1a_2^3$$
$$a_4+x^8a_2^4a_4+3x^4a_0^2a_3a_4+4x^3a_0^3a_3a_4+6x^5a_0a_1a_3a_4+12x^4a_0^2a_1a_3a_4+3x^6a_1^2a_3a_4+12x^5a_0a_1^2a_3$$
$$a_4+4x^6a_1^3a_3a_4+2x^7a_2a_3a_4+6x^6a_0a_2a_3a_4+12x^5a_0^2a_2a_3a_4+6x^7a_1a_2a_3a_4+24x^6a_0a_1a_2a_3a_4+12x^7$$
$$a_1^2a_2a_3a_4+3x^8a_2^2a_3a_4+12x^7a_0a_2^2a_3a_4+12x^8a_1a_2^2a_3a_4+4x^9a_2^3a_3a_4+6x^7a_0a_3^2a_4+6x^6a_0^2a_3^2$$
$$a_4+6x^8a_1a_3^2a_4+12x^7a_0a_1a_3^2a_4+6x^8a_1^2a_3^2a_4+6x^9a_2a_3^2a_4+12x^8a_0a_2a_3^2a_4+12x^9a_1a_2a_3^2a_4+6x^{10}$$
$$a_2^2a_3^2a_4+3x^{10}a_3^3a_4+4x^9a_0a_3^3a_4+4x^{10}a_1a_3^3a_4+4x^{11}a_2a_3^3a_4+x^{12}a_3^4a_4+4x^4a_0^3a_4^2+12x^5a_0^2$$
$$a_1a_4^2+12x^6a_0a_1^2a_4^2+4x^7a_1^3a_4^2+x^8a_2a_4^2+12x^6a_0^2a_2a_4^2+24x^7a_0a_1a_2a_4^2+12x^8a_1^2a_2a_4^2+12x^8a_0$$
$$a_2^2a_4^2+12x^9a_1a_2^2a_4^2+4x^{10}a_2^3a_4^2+3x^8a_0a_3a_4^2+12x^7a_0^2a_3a_4^2+3x^9a_1a_3a_4^2+24x^8a_0a_1a_3a_4^2+12$$
$$x^9a_1^2a_3a_4^2+3x^{10}a_2a_3a_4^2+24x^9a_0a_2a_3a_4^2+24x^{10}a_1a_2a_3a_4^2+12x^{11}a_2^2a_3a_4^2+3x^{11}a_3^2a_4^2+12$$
$$x^{10}a_0a_3^2a_4^2+12x^{11}a_1a_3^2a_4^2+12x^{12}a_2a_3^2a_4^2+4x^{13}a_3^3a_4^2+6x^8a_0^2a_4^3+12x^9a_0a_1a_4^3+6x^{10}a_1^2$$
$$a_4^3+12x^{10}a_0a_2a_4^3+12x^{11}a_1a_2a_4^3+6x^{12}a_2^2a_4^3+x^{12}a_3a_4^3+12x^{11}a_0a_3a_4^3+12x^{12}a_1a_3a_4^3+12x^{13}$$
$$a_2a_3a_4^3 + 6x^{14}a_3^2a_4^3 + 4x^{12}a_0a_4^4 + 4x^{13}a_1a_4^4 + 4x^{14}a_2a_4^4 + 4x^{15}a_3a_4^4 + x^{16}a_4^5$$

Warning

We do get an exponential speedup, but one really needs to keep track of the actual cost of squaring here.

- The degree doubles at each step.
- The coefficients can grow exponentially.

These problems go away if we compute over a finite structure such as the modular numbers $\mathbb{Z}_m$ and in particular in a quotient like $\mathbb{Z}_m/\big(x^n - 1\big)$.

But Beware of Hasty Conclusions

We cannot conclude that $f^t(x)$ can always be computed in $O(\log t)$ applications of f.

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.

In general, we have no 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.

Wurzelbrunft's Conjecture

…is wishful thinking.

To see why, recall that checking a Boolean formula $\phi(x_1, \ldots, x_n)$ for satisfiability is $\mathbb{NP}$-hard.

Define a function f on $\mathbf{2}^n \cup \{\top\}$ as follows

$$f(\boldsymbol{x}) = \begin{cases} \top & \text{if } \phi(\boldsymbol{x}) \text{ is true} \\ \boldsymbol{x}+1 & \text{otherwise.} \end{cases}$$

$$f(\top) = \top$$

Then ϕ is satisfiable iff $f^{2^n}(\mathbf{0}) = \top$.

Another Hasty Conclusion

Here is a strange integer sequence based on logs, floors and ceilings:

$$a_n = \left\lceil \frac{2}{2^{1/n} - 1} \right\rceil - \left\lfloor \frac{2n}{\ln 2} \right\rfloor$$

This time, the sequence starts like so:

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

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

At any rate, it sure looks like the sequence is constant 0. Alas

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

[†] Note that it is a minor pain to compute the terms; it's not even clear that $n \mapsto a_n$ is primitive recursive (the expression looks like real arithmetic, but it can be handled with just integer arithmetic).

Iteration versus Recursion

Intuitively, there are two basic ways to evaluate a recursive function:

bottom-up Use a loop to calculate result for large arguments, starting at small arguments.

top-down Unfold the recursion starting from large arguments and tracing things downward till termination occurs at small arguments.

The unfolding part requires a bit of bureaucracy, one needs to keep track of pending calls. On the other hand, the number of calls may be smaller than in the bottom-up approach.

More Formally . . .

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

$$\begin{aligned} F(0, y) &= y \\ F(x^+, y) &= f(F(x, y)) \end{aligned}$$

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

Again, this is just the standard bottom-up approach to computing an primitive recursive function, expressed in an elegant and concise way.

As Iteration

Conversely, iteration can be used to express recursion. Suppose

$$f(0, \boldsymbol{y}) = g(\boldsymbol{y})$$
$$f(x^+, \boldsymbol{y}) = h(x, f(x, \boldsymbol{y}), \boldsymbol{y})$$

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.

Unary Iteration

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

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

Here $\mathsf{rt}(x)$ is the integer part of $\sqrt{x}$. These functions suffice to set up the usual coding machinery. If we add an additional operation of iteration

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

we can replace primitive recursion by unary iteration.

Exercise

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

Trajectories and Orbits

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.

Terminology Warning

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

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

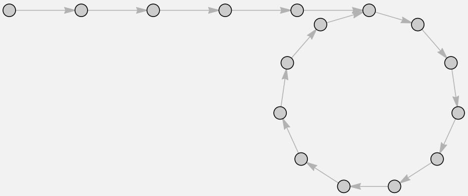
While the sequence is always infinite, the underlying set may well be finite, even when the carrier set is infinite.

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.

The Lasso

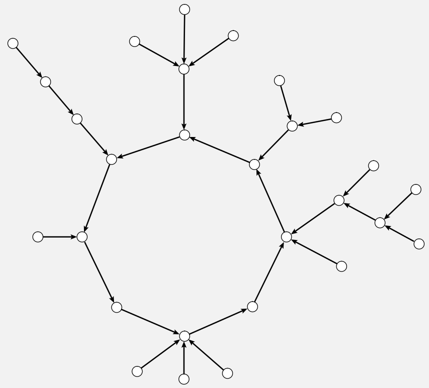
At any rate, if the carrier set is finite, all trajectories must ultimately wrap around and all orbits must be finite:



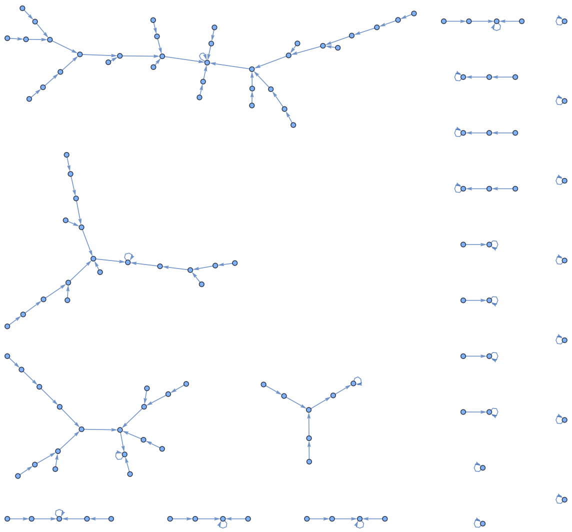
What changes is only the length of the transient part and the length of the cycle (in the picture 5 and 11).

Limit Cycles

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.



Recall Topswops



Reachability

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.

Confluence (aka Basins of Attraction)

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.

Confluence is an Equivalence

Proposition

Confluence is an equivalence relation.

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

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

$$f^{i+d}(a) = f^{j+d}(b) = f^k(b) = f^l(c).$$

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.

Calculating Transients and Periods

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).

Floyd's Trick

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?

Time/Space Tradeoff

At first glance, this might even seem 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 $s < t$.

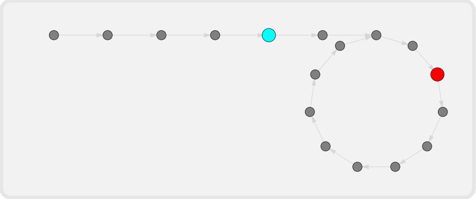
This requires an absurd amount of recomputation and is thus highly inefficient, but it trivially works and it uses only constant memory.

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

A Memoryless Approach

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.

Pebble Race

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

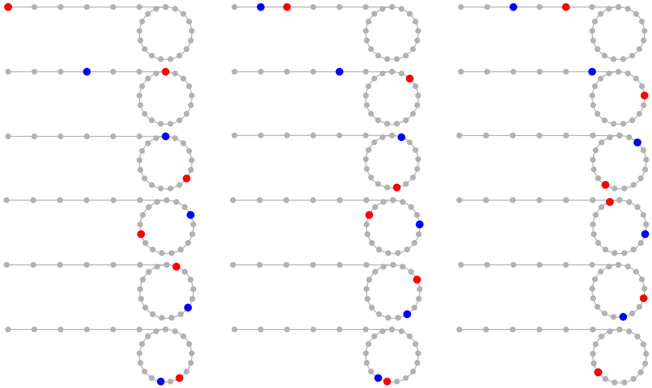
The slow pebble u enters the limit cycle at time t, the transient, when the fast pebble v is already there. From now on, v gains one place on u at each step. So pebble v must catch up at time s where $s \leq t + p$, where p is the period. The meeting time is called the Floyd-time.

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.

Example

Here the transient is 6, and the period 17.



The Floyd-time here is 17.

Tables

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.

How about the Transient?

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.

Floyd's Cycle Finding Algorithm

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 $O(1)$.

Here time really means "number of applications of f" and space means the size of an object in A: we only need to store a small constant number of elements in A.

Generalization

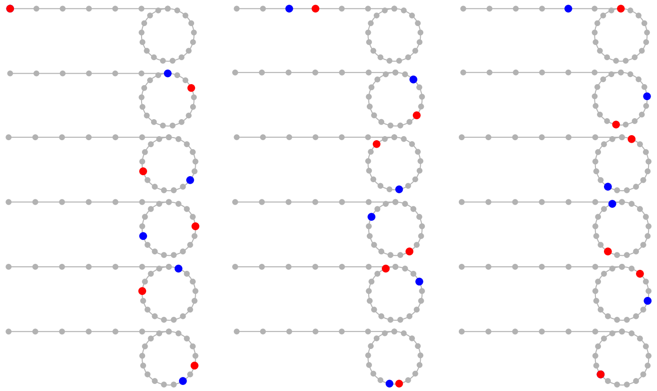
Innocent Question:

What happens if we change the pebble speeds to some value $1 \leq u < v$?

More precisely, for which values of u, v, t, p does Floyd's algorithm find a point on the cycle?

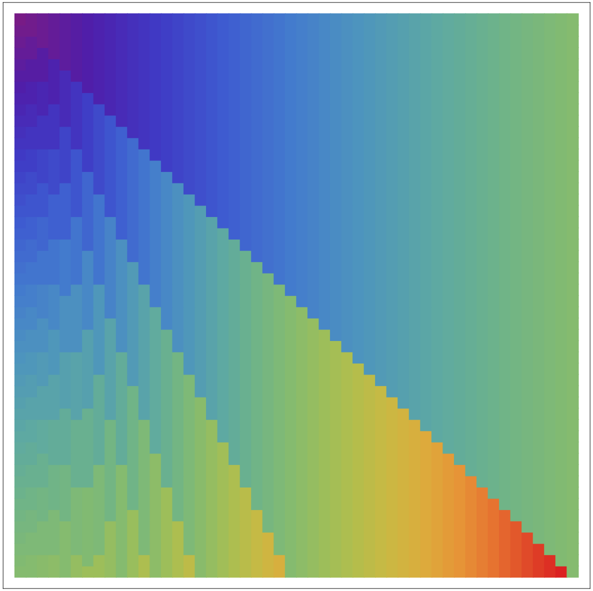
It might be that for some combinations the pebbles cycle forever and never meet.

Example

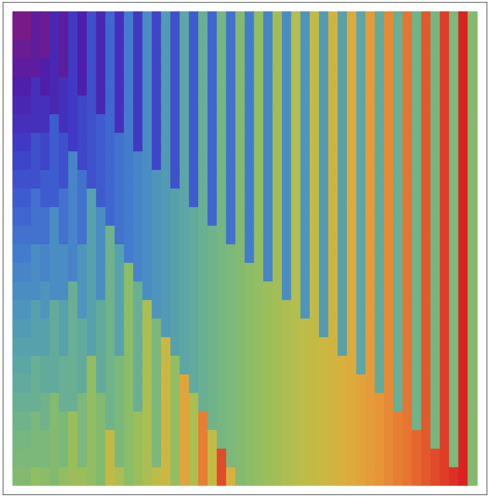


Pebbles speeds 2 and 3, transient 6, period 17. Everything works fine.

Floyd-Times



Floyd-Times Chaos



This uses speeds 2 and 4. Seems fairly complicated.

Some Questions

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.

Brent’s Algorithm

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.

Applications

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

Logarithms in a Group

Suppose G is some finite cyclic group with generator g and order n. We can easily exponentiate in G, the operation

$$e \rightsquigarrow g^e$$

takes $O(\mathtt{M} \log e)$ steps where $\mathtt{M}$ is the cost of a single multiplication in G. But going backwards is apparently hard in many groups:

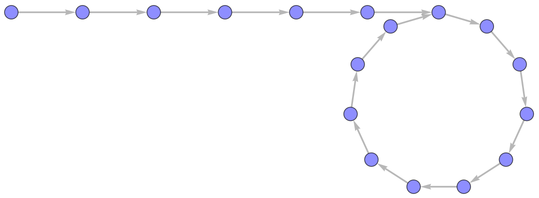
Discrete Logarithm Problem:

Given $a \in G$, find e such that $g^e = a$.

Of course, $e = \log_g a$ is trivially computable by a brute force search, but we are here interested in efficient computation when the group order n is sufficiently large.

Pollard's Rho Method

This should be called Pollard's Lasso method (in particular since the second algorithm in the paper is about "catching kangaroos"), but it's too late now.



A Rohrschach test:

- If you have a classical education, you will see a ρ.
- If you're a cowboy, you will see a lasso.

Random Maps

The motivation for this method is a bit strange. Consider a random function $f : A \to A$ where A has size n.

Then the expected value of some key parameters of the functional digraph of f are as follows:

# components	$\frac{1}{2} \log n$
# leaf nodes	$e^{-1} n$
# recurrent nodes	$\sqrt{\pi n/2}$
transient length	$\sqrt{\pi n/8}$
period length	$\sqrt{\pi n/8}$

The expected lengths of the longest transient/cycle are also $c_{1/2}\sqrt{n}$ where $c_1 \approx 1.74$ and $c_2 \approx 0.78$.

Computing Transient and Period

For simplicity we can think of the expected value of transient length t and period length p of a random point a in A as $\sqrt{n}$.

We know an elegant algorithm to compute these parameters: Floyd's trick. More precisely, we can compute t and p in expected time $O(\sqrt{n})$ using $O(1)$ space.

Wild Idea:

Can we compute a (pseudo-)random sequence (x_i) of group elements so that $x_i = x_{2i}$ helps us to compute a discrete logarithm?

The Map

We need a “random” map.

To this end we first split the group G into three sets G_1, G_2 and G_3 of approximately equal size (sets, not subgroups, so this will be easy in practical situations). Any ham-fisted approach will do.

Now, given a generator g and some element a, define $f : G \to G$ as follows:

$$f(x) = \begin{cases} gx & \text{if } x \in G_1, \\ x^2 & \text{if } x \in G_2, \\ ax & \text{otherwise.} \end{cases}$$

Of course, f is perfectly deterministic given the partition of G.

The Orbits

Consider the orbit (x_i) of 1 under f.

Clearly, all the elements have the form $a^{\alpha_i} g^{\beta_i}$ and the exponents are updated according to

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i + 1) & \text{if } x \in G_1, \\ (2\alpha_i, 2\beta_i) & \text{if } x \in G_2, \\ (\alpha_i + 1, \beta_i) & \text{otherwise.} \end{cases}$$

Since the partition of G is random, the three steps are chosen randomly.

Use Floyd

Use Floyd to find the minimal index e such that $x_e = x_{2e}$:

$$a^{\alpha_e} g^{\beta_e} = a^{\alpha_{2e}} g^{\beta_{2e}}$$

But then

$$a^{\alpha_e - \alpha_{2e}} = g^{\beta_{2e} - \beta_e}$$

This equality does not solve the discrete logarithm problem directly but it can help at least sometimes.

Again, for cryptographic applications any such weakness is potentially fatal: a good method must be secure under any and all circumstances.

Example: $\mathbb{Z}_{999959}$

Consider the multiplicative group of $\mathbb{Z}_p$ where $p = 999959$.
Pick generator $g = 7$ and let $a = 3$.

Perhaps the most simpleminded partition is to chop $[p-1]$ into thirds. This produces an orbit with transient and period

$$928 \qquad 587$$

A similarly obvious partition would use $x \bmod 3$. This produces an orbit with transient and period

$$919 \qquad 575$$

Note the values are order-of-magnitude close to $\sqrt{p}$, looks like our maps are sufficiently random.

Onward

Running a suitably modified version of Floyd's algorithm with the first partition produces $e = 1174$ and $x_e = 11400$, plus the identity

$$3^{310686} = 7^{764000} \pmod{p}$$

Close, but no cigar: we need to somehow clobber the exponent 310686.

The last identity lives in $\mathbb{Z}_p^\star$, a group of order $p-1$.

So we could try to simplify exponents modulo $p-1$.

Magic

Use the Extended Euclidean algorithm to get

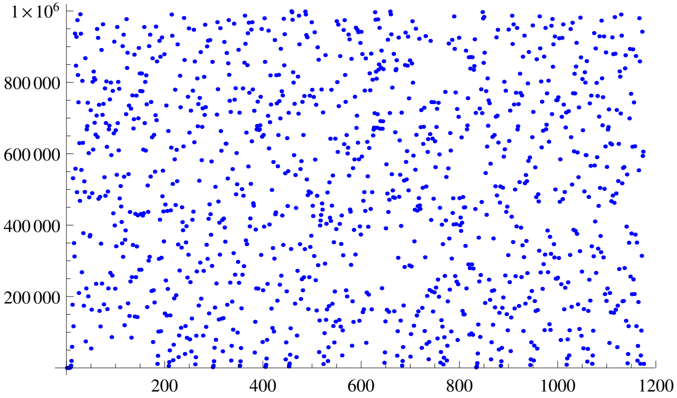
$$\begin{aligned}\gcd(310686, p-1) &= 2\\ &= 148845 \cdot 310686 - 46246 \cdot 999958\end{aligned}$$

Then raise 3^{310686} to the 148845 power mod p to obtain

$$\begin{aligned}3^2 &= 7^{356324} \pmod{p}\\ 3 &= \pm 7^{178162} \pmod{p}\end{aligned}$$

We can simply check the two cases and find that in $\mathbb{Z}_p$: $\log_7 3 = 178162$.

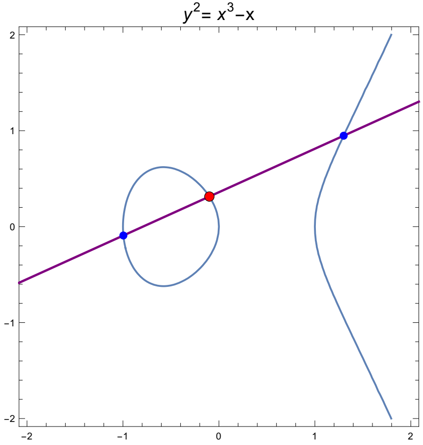
How Random Is It?



A plot of the orbit of 1 given our “random” partition.

Aside: Elliptic Curves

Curves of the form $y^2 = x^3 + ax + b$ over a finite field produce a nice group that can be used for discrete logarithm methods.


$y^2 = x^3 - x$
2
1
0
−1
−2
−2
−1
0
1
2