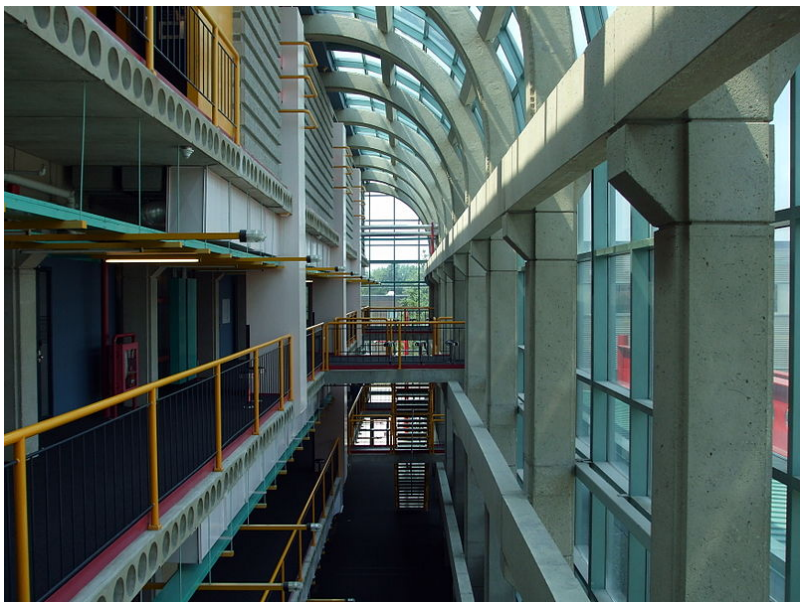
Developing `Walnut` Commands for Sequence Properties

Jeffrey Shallit

School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1 Canada
shallit@uwaterloo.ca
https://cs.uwaterloo.ca/~shallit/



Introduction

As you know, `Walnut` is free software to allow you to rigorously prove or disprove first-order statements about (generalized) automatic sequences.

The goal of this talk is to develop a few `Walnut` commands to solve interesting problems in combinatorics on words.

I'll assume you have some familiarity with `Walnut` and understand the basic syntax of a `Walnut` command.

I'll also assume you know what automata and regular expressions are and what an automatic sequence is.

Along the way I'll give some hints about how to avoid common pitfalls, debug `Walnut` predicates, and make them run faster.

We'll also see some open problems nobody currently knows how to solve.

Terminology and notation

A *word* or *string* is a list of letters.

A *factor* (sometimes called a *subword*) is a contiguous block sitting inside another word. For example, `woord` is a factor of `zakwoordenboek`.

We can specify a factor by giving its starting and ending positions inside a word, as in $\mathbf{f}[i..j]$.

We say x is an *n'th power* if $x = y^n = \overbrace{yy\cdots y}^{n}$ for some string y.

By x^ω we mean the infinite word $xxx\cdots$.

By $(n)_k$ we mean the string representing n in base k, with no leading zeros.

Some things to remember about Walnut

We can't quantify over words in Walnut. We cannot say things like "For all words $w \in \{0,1\}^*$" or "There exists a word w having the property that...".

We can only quantify over variables taking integer values.

So when we talk about factors of infinite words, we're not really talking about factors, but rather *occurrences* of factors.

Occurrences can be specified by starting and ending positions, as in $\mathbf{f}[i..j]$, or by starting position and length, as in $\mathbf{f}[i..i+n-1]$.

Sometimes one is more natural than the other.

Problem 1: The **vtm** sequence

The **vtm** sequence

$$2102012101202102012021012102012 1 \cdots$$

has been studied since Axel Thue defined it in 1912.

It has many equivalent definitions.

Maybe the easiest is the following: consider the Thue-Morse sequence

$$\mathbf{t} = 0110100110010110 \cdots ,$$

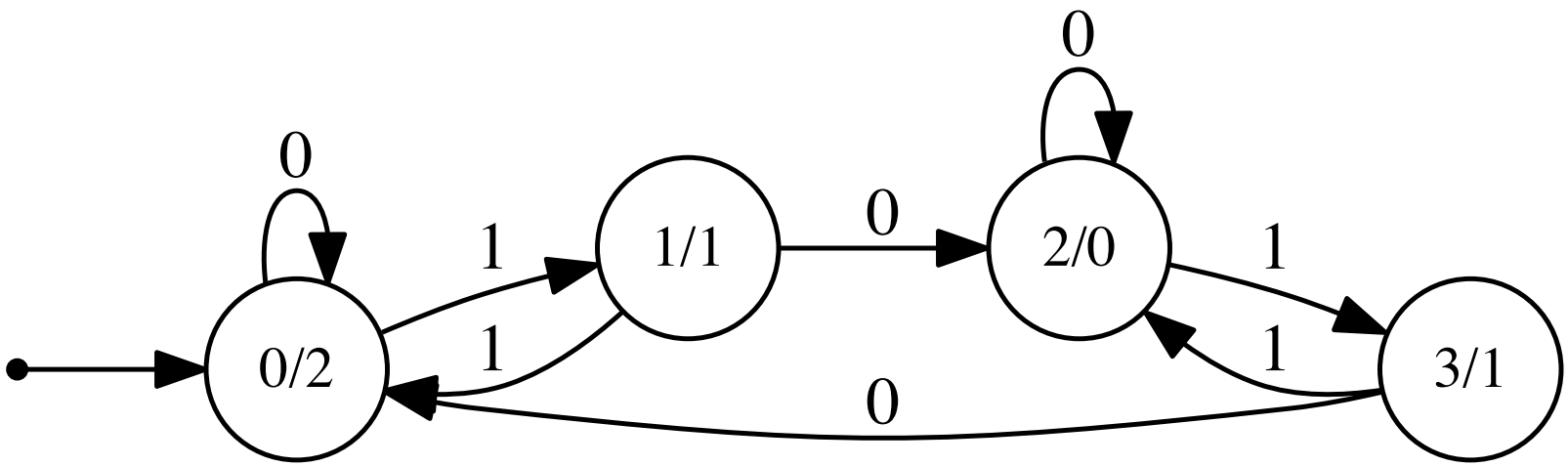
and count the number of 1's between consecutive 0's:

$$0 \underbrace{11}_{2} 0 \underbrace{1}_{1} 0 \underbrace{}_{0} 0 \underbrace{11}_{2} 0 \underbrace{}_{0} 0 \underbrace{1}_{1} 0 \underbrace{11}_{2} 0 \cdots$$

The lengths of the runs of 1's gives **vtm**.

Problem 1: The **vtm** sequence

Another definition, and the one `Walnut` uses, is that **vtm** is generated by the following DFAO (deterministic finite automaton with output).



Input is n in base 2; output of the last state reached is **vtm**$[n]$.

Problem 1: The **vtm** sequence

Let us show that these two definitions are the same.

To do that, we need to find the position of the n'th 0 in the Thue-Morse word $\mathbf{t}$.

That's actually pretty easy, since the Thue-Morse sequence consists of blocks of 01 or 10, and the n'th block is determined by $\mathbf{t}[n]$.

So we can make an automaton that accepts a pair (n, x) if and only if x is the position of the n'th 0, as follows:

```
def tm0pos "x=2*n+T[n]":
```

Problem 1: The **vtm** sequence

And now we check that the number of 1's between the n'th and $(n+1)$'st 0 is **vtm**$[n]$:

```
eval defcheck "An,x,y ($tm0pos(n,x) & $tm0pos(n+1,y)) =>
   y=x+1+VTM[n]":
```

And `Walnut` returns `TRUE`.

Problem 1: The **vtm** sequence

Another definition of **vtm** is that it is the first difference sequence $(\mathbf{t}[n+1] - \mathbf{t}[n])_{n \geq 0}$, but *recoded* as follows:

$$1 \to 2, \quad 0 \to 1, \quad -1 \to 0.$$

Exercise. Create a `Walnut` command that verifies that this defines **vtm**. You have to be a little careful since the default domain for numbers in `Walnut` is $\mathbb{N}$ and not $\mathbb{Z}$.

Problem 1: The **vtm** sequence

The most interesting thing about the **vtm** sequence is that it is *squarefree*: it does not contain a square (two consecutive identical blocks).

Let's make a first-order logic formula that asserts the existence of two consecutive identical blocks in **vtm**.

Let's say the first block begins at position i, and is of length $n \geq 1$. So the second block would begin at position $i + n$.

Then a square means that the symbol at position i is the same as that at $i + n$, $i + 1$ the same as at $i + n + 1$, and so forth, up to $i + n - 1$ the same as $i + 2n - 1$.

So the formula would be

$$\exists i, n \; \forall t \; (t < n) \Longrightarrow \mathbf{vtm}[i + t] = \mathbf{vtm}[i + n + t].$$

Problem 1: The **vtm** sequence

So in Walnut we have

```
eval has_square "Ei,n At (t<n) => VTM[i+t]=VTM[i+n+t]":
```

and we expect to get the result FALSE. But we don't! We get TRUE!

What did we do wrong?!?

To figure out where we went wrong, let's make an automaton that accepts those n for which there are two consecutive blocks of length n.

```
def check "Ei At (t<n) => VTM[i+t]=VTM[i+n+t]":
```

And when we look at the automaton check we see that it only accepts $n = 0$.

That was our mistake: we needed to specify that the consecutive identical blocks were *nonempty*.

Problem 1: The **vtm** sequence

We *should* have written

$$\exists i, n\ (n \geq 1)\ \wedge\ \forall t\ (t < n) \Longrightarrow \mathbf{vtm}[i+t] = \mathbf{vtm}[i+n+t].$$

and

```
eval has_square "Ei,n (n>=1) & At (t<n) =>
   VTM[i+t]=VTM[i+n+t]":
```

and this time we do indeed get `FALSE` as a result. So **vtm** is indeed squarefree.

Moral of the story: be careful about edge conditions (empty strings, starting at $i = 1$ instead of $i = 0$, etc.)

Open Problem #1

Here is a problem currently nobody knows how to solve.

What is the lexicographically least squarefree string over the alphabet $\{0, 1, 2\}$?

It starts $01020120210120102012\cdots$.

Prove something (anything!) interesting about it.

For example, do the letters 0, 1, 2 occur with some limiting frequency?

Is it generated by an automaton? (almost certainly not).

Does every finite block that can be extended infinitely to the right occur in it?

It doesn't look like `Walnut` can be helpful for this problem at all.

Problem 2: Avoiding the pattern xxx^R in binary strings

We can't avoid the pattern xx (two consecutive identical blocks) over the alphabet $\{0, 1\}$.

But we *can* avoid the pattern xxx^R, where x^R denotes the reverse of x.

For example, the periodic string $(01)^\omega = 01010101\cdots$ avoids xxx^R.

But are there any *aperiodic* infinite binary strings avoiding this pattern?

Problem 2: Avoiding the pattern xxx^R in binary strings

Here is an approach that often succeeds in problems like this: use breadth-first search in the tree of all strings to identify strings with the desired property that *also* can be generated by an automaton with a relatively small number of states.

The number of states can be deduced using a heuristic variation on the Myhill-Nerode theorem.

If we are lucky, we'll find a small automaton, and then we can use `Walnut` to prove that the sequence generated by the automaton actually has the desired property.

Problem 2: Avoiding the pattern xxx^R in binary strings

When we try to do this for xxx^R, one complicating issue is that aperiodicity is a property of infinite strings, not finite prefixes!

Nevertheless, we can still use the technique by fixing some n and making sure the word contains no blocks repeated n times.

Problem 2: Avoiding the pattern xxx^R in binary strings

When we do this using automata in base 2, and avoiding both xxx^R and $yyyy = y^4$, we don't succeed in finding any small automata that work.

But we *do* succeed if we use automaton based on a different way of representing integers: the Zeckendorf representation based on Fibonacci numbers.

Recall $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

Zeckendorf's theorem says that every integer has a unique representation as a sum of distinct Fibonacci numbers F_i, $i \geq 2$, provided that you never use both F_i and F_{i+1}.

We can write such a representation in the form $\sum_{2 \leq i \leq t} e_i F_i$ with $e_i \in \{0, 1\}$.

Problem 2: Avoiding the pattern xxx^R in binary strings

In this case we quickly find an 8-state automaton defining a binary sequence **rf** $= 00100110110110010011011\cdots$. This is a candidate we need to test.

```
eval chkxxxr "?msd_fib ~Ei,n (n>=1) &
    (At (t<n) => RF[i+t]=RF[i+n+t]) &
    (At (t<n) => RF[i+t]=RF[(i+3*n)-(t+1)])"::

eval chkaperiod "?msd_fib ~Ei,n (n>=1) &
    At (t>i) => RF[i+t]=RF[i+n+t]":
```

and both return TRUE.

Problem 3: Balanced words

Roughly speaking, a word is *balanced* if every pair of equal-length blocks occurring in it do not differ very much in the number of occurrences of each letter.

More precisely, let $|x|_a$ denote the number of occurrences of the letter a in the string x. We say a word z (finite or infinite) is balanced if all factors x, y of z with $|x| = |y|$ satisfy the inequality

$$||x|_a - |y|_a| \leq 1$$

for all letters a.

We would like to be able to check this property with `Walnut`, but to do so, it seems that we would need to be able to count the number of a's occurring in two different blocks and compare them.

There is no obvious way to do this with a first-order logical statement.

Problem 3: Balanced words

So it seems we are at an impasse.

However, for *binary* words, there is a way around this!

Coven and Hedlund proved that a binary word z is *not* balanced if and only if z contains two blocks of the form $0v0$ and $1v1$, for some (possibly empty) word v.

Now *that* is first-order expressible!

Problem 3: Balanced words

We can check if the block $\mathbf{z}[i..i+n]$ is *unbalanced* as follows:

$$\begin{aligned}\exists k,l,m \quad & m \geq 1 \ \wedge \ k \geq i \ \wedge \ l \geq i \ \wedge \ k+m \leq i+n \ \wedge \\ & l+m \leq i+n \ \wedge \ \mathbf{z}[k] = \mathbf{z}[k+m] \ \wedge \ \mathbf{z}[l] = \mathbf{z}[l+m] \ \wedge \\ & \mathbf{z}[k] \neq \mathbf{z}[l] \ \wedge \ \forall t \ (t \geq 1 \ \wedge t < m) \Longrightarrow \mathbf{z}[k+t] = \mathbf{z}[l+t].\end{aligned}$$

Let's use this idea to verify that the Fibonacci word $\mathbf{f}$ is balanced:

```
def unbalanced "?msd_fib Ek,l,m m>=1 & k>=i & l>=i &
   k+m<=i+n & l+m<=i+n & F[k]=F[k+m] & F[l]=F[l+m] &
   F[k]!=F[l] & At (t>=1 & t<m) => F[k+t]=F[l+t]":
eval balanced "?msd_fib ~Ei,n (n>=1) & $unbalanced(i,n)":
```

Open Problem #2

Is there a similar first-order criterion for balanced blocks over larger alphabets?

Problem 4: Bordered and unbordered words

A word is *bordered* if it begins and ends with the same word in a nontrivial way (we exclude the empty word, and we exclude the word itself). Otherwise it is *unbordered*.

For example, the Dutch word `verstuivers` is bordered, as it begins and ends with `vers`.

Currie and Saari studied the unbordered factors of the Thue-Morse word $\mathbf{t}$. They proved the presence of an unbordered factor of $\mathbf{t}$ for all lengths $n \not\equiv 1 \pmod 6$.

This is a sufficient condition, but not a necessary one, as

$$\mathbf{t}[39..69] = 00110100101101001100101101001 01$$

is an unbordered factor of length 31.

So it is natural to ask for which lengths n there is an unbordered factor of length n of $\mathbf{t}$.

Problem 4: Bordered and unbordered words

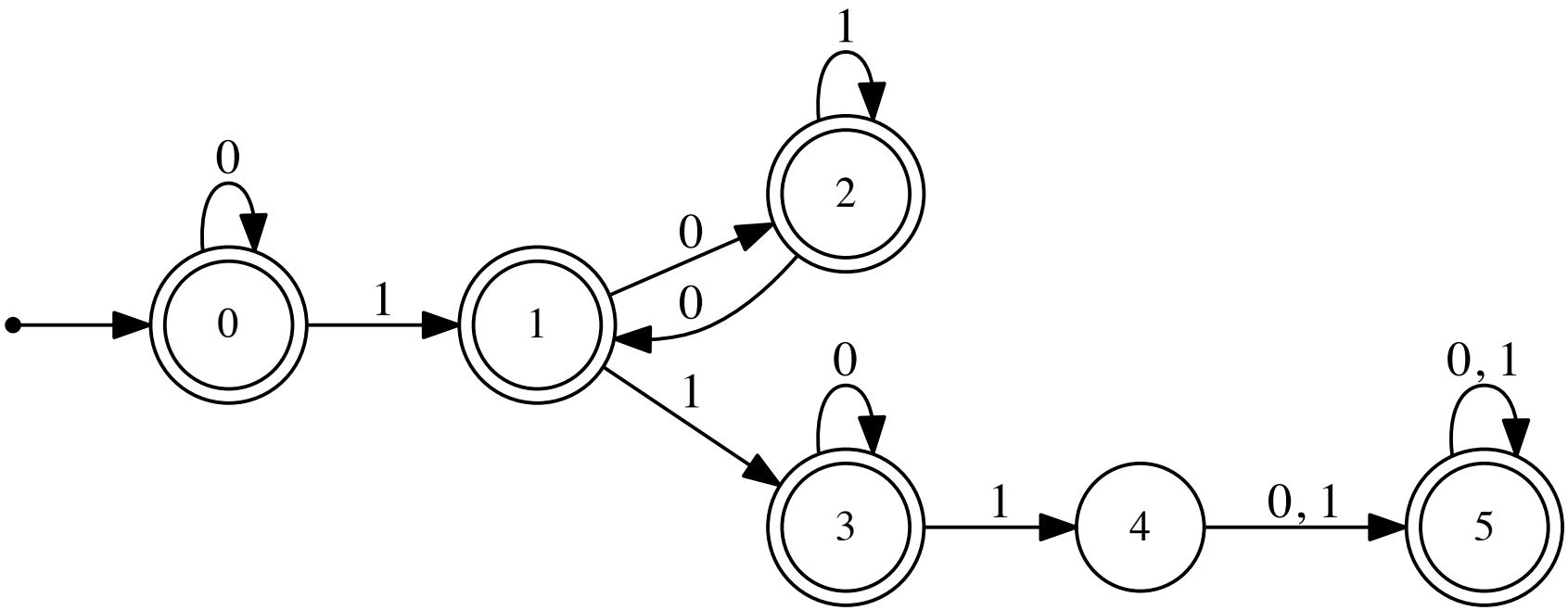
This is exactly the kind of thing Walnut can answer with ease.

Let us define a Walnut formula for the factor $\mathbf{t}[i..i+n-1]$ having a border of length m:

```
def bordtm "At (t<m) => T[i+t]=T[(i+n+t)-m]":
# does T[i..i+n-1] have border of length m
def bordered "Em m>=1 & m<n & $bordtm(i,m,n)":
# is T[i..i+n-1] bordered?
def unbordered "Ei ~$bordered(i,n)":
# is there an unbordered factor of length $n$?
```

This gives the following automaton...

Problem 4: Bordered and unbordered words



As you can see from the resulting automaton, we now get

Theorem. There is an unbordered factor of length n if and only if $(n)_2 \notin 1(01^*0)^*10^*1$.

Problem 5: Shevelev's pseudoperiodicity problem

Shevelev called a sequence $(a_n)_{n\geq 0}$ *k-pseudoperiodic* if there is a set of k positive integers $S = \{p_1, p_2, \ldots, p_k\}$ such that $a_i \in \{a_{i+p_1}, a_{i+p_2}, \ldots, a_{i+p_k}\}$ for all $i \geq 0$. The set S is called a pseudoperiod.

He proved that $\mathbf{t}$ is not 2-pseudoperiodic, but it is 3-pseudoperiodic.

He proposed, as an open problem, to characterize all the triples $0 < a < b < c$ such that $\{a, b, c\}$ is a pseudoperiod for $\mathbf{t}$.

We can do this in `Walnut` as follows:

```
def triple "(a>=1) & (a<b) & (b<c) &
   Ai (T[i]=T[i+a] | T[i]=T[i+b] | T[i]=T[i+c])":
```

Problem 5: Shevelev's pseudoperiodicity problem

The resulting automaton has 53 states and completely answers Shevelev's question.

Determining it was a major calculation in Walnut, requiring 1949 seconds of CPU time and 18 GB of RAM. The largest intermediate automaton had 2,952,594 states.

So it is not that surprising that Shevelev could not find a simple characterization of these triples!

Open Problem #3

Characterize all the 3-pseudoperiods of the Tribonacci sequence.

This is in principle doable with `Walnut`, but currently it seems far out of reach in terms of the number of states needed for intermediate automata.

Problem 6: Rich words

A result of Droubay, Justin, and Pirillo says that a word of length at most n contains at most n distinct palindromic factors (not counting the empty word). A word achieving this maximum is said to be *rich*.

The shortest non-rich binary word is 00101100.

Once again at first glance there seems to be no obvious way to express this definition in first-order logic, because it seems to require counting the number of distinct palindromic factors.

However, in this case there is an alternative characterization of rich words: a word w is rich if every nonempty prefix p of w has a nonempty palindromic suffix s that occurs only once in p. *That* is first-order expressible.

Let's build up some `Walnut` formulas for it step-by-step, for the Fibonacci word $\mathbf{f}$.

Problem 6: Rich words

First we need a formula asserting that $\mathbf{f}[j..k]$ is a palindrome.

```
def palf "?msd_fib At (t+j<=k) => F[j+t]=F[k-t]":
```

Next we need a formula that asserts that a particular block $\mathbf{f}[i..j]$ occurs inside another block $\mathbf{f}[m..n]$, say starting at some position k.

```
def occursf "?msd_fib Ek k>=m & k+j<=i+n &
   At (t+i<=j) => F[i+t]=F[k+t]":
```

Now we assert that every nonempty prefix $p = \mathbf{f}[i..k]$ of $\mathbf{f}[i..i+n-1]$ has some nonempty palindromic suffix $s = \mathbf{f}[j..k]$ that occurs nowhere else.

```
def richf "?msd_fib Ak (k>=i & k<i+n) => Ej j>=i &
   j<=k & $palf(j,k) & ~$occursf(j,k,i,k-1)":
eval isfibrich "?msd_fib Ai,n $richf(i,n)":
```

Problem 6: Rich words

Now that we have this, we can easily carry out the same thing for any automatic word, such as Thue-Morse:

```
def palt "At (t+j<=k) => T[j+t]=T[k-t]":
def occurst "Ek k>=m & k+j<=i+n &
   At (t+i<=j) => T[i+t]=T[k+t]":
def richt "Ak (k>=i & k<i+n) => Ej j>=i & j<=k &
   $palt(j,k) & ~$occurst(j,k,i,k-1)":
```

As it turns out, no factor of **t** of length > 16 is rich.

```
eval tmp1 "Ei $richt(i,16)":
eval tmp2 "Ei,n $richt(i,n) & n>16":
```

Problem 6: Rich words

On the other hand, the period-doubling word

$$\mathbf{pd} = 10111010\cdots$$

is rich. We can show this by demonstrating that every factor is rich:

```
def palpd "At (t+j<=k) => PD[j+t]=PD[k-t]":
def occurspd "Ek k>=m & k+j<=i+n &
   At (t+i<=j) => PD[i+t]=PD[k+t]":
def richpd "Ak (k>=i & k<i+n) => Ej j>=i & j<=k &
   $palpd(j,k) & ~$occurspd(j,k,i,k-1)"
eval testpd "Ai,n $richpd(i,n)":
```

which returns TRUE.

Problem 7: Antipowers

An k'th power in a word consists of k consecutive identical blocks.

Fici et al. defined a notion of k-antipower, which is k consecutive blocks, no two of which are equal. For example, `orange` is a 3-antipower, but `banana` is not.

We can define a first-order formula for $\mathbf{z}[i..i+3n-1]$ being a 3-antipower, as follows:

$$\begin{aligned}\text{faceq}(i,j,n) &:= \forall t\ (t<n) \Longrightarrow \mathbf{z}[i+t] = \mathbf{z}[j+t] \\ \text{anti3}(i,n) &:= \neg\,\text{faceq}(i,i+n,n)\ \wedge\ \neg\,\text{faceq}(i,i+2n,n) \\ &\qquad \wedge\ \neg\,\text{faceq}(i+n,i+2n,n).\end{aligned}$$

and similar for other kinds of antipowers. Notice that to check a k-antipower seems to require $k(k-1)/2$ clauses.

Problem 7: Antipowers

The Cantor sequence **ca** $= 101000101\cdots$ is defined as follows:

$$\mathbf{ca}[n] := \begin{cases} 1, & \text{if } (n)_3 \text{ contains no 1's;} \\ 0, & \text{otherwise.} \end{cases}$$

Fici et al. proved that the Cantor sequence contains no 11-antipowers. We can optimally improve this to "contains no 10-antipowers". This would seem to require 45 clauses, but actually we can get by with fewer:

```
def cfaceq "?msd_3 At t<n => CA[i+t]=CA[j+t]":
eval cno10 "?msd_3 Ai,n n>=1 => (
   $cfaceq(i+0*n,i+1*n,n) | $cfaceq(i+0*n,i+5*n,n) |
   $cfaceq(i+2*n,i+3*n,n) | $cfaceq(i+2*n,i+8*n,n) |
   $cfaceq(i+3*n,i+4*n,n) | $cfaceq(i+3*n,i+7*n,n) |
   $cfaceq(i+3*n,i+9*n,n) | $cfaceq(i+4*n,i+5*n,n) |
   $cfaceq(i+4*n,i+9*n,n) | $cfaceq(i+5*n,i+6*n,n) |
   $cfaceq(i+5*n,i+8*n,n) | $cfaceq(i+6*n,i+7*n,n) |
   $cfaceq(i+7*n,i+8*n,n) | $cfaceq(i+8*n,i+9*n,n))":
```

Problem 8: What kinds of properties can `Walnut` check?

So far all of the problems we have looked at have the same general form.

We are given some property of a finite word (being a square, or being rich, or balanced, etc.)

We want to know which factors of a particular infinite word (like **f** or **t**) have this property.

We would like to know what kinds of properties we can check with `Walnut`.

Problem 8: What kinds of properties can Walnut check?

There are some properties that we know with certainty we *cannot* check, in general.

For example, being an abelian square: two consecutive blocks where one is a permutation of the other.

We can prove this is not specifiable in the first-order logic that Walnut uses, because Luke Schaeffer found a specific example of an infinite word generated by an automaton (the paperfolding word) where the positions of the abelian squares do not form a regular language.

Problem 8: What kinds of properties can Walnut check?

But there are other even simpler properties that are not checkable, such as "containing an even number of 1's".

It seems remarkable that we cannot specify such a simple property in Walnut's logical language.

Open problem: characterize the class $FO[+]$

The $FO[+]$ definable languages are those languages L for which membership in L is definable in `Walnut`'s logical language, namely, $\langle N, +, V_k \rangle$, where $V_k(n)$ is the highest power of k dividing n.

It would be nice to have some more "constructive" definition of the language class $FO[+]$, one that would immediately let you figure out whether $L \in FO[+]$.

Problem 8: What kinds of properties can `Walnut` check?

Even so, for the property "having an even number of 1's", there *is* a way to check factors of an infinite word with `Walnut`!

The idea is to use a *finite-state transducer*. This is an automaton that takes an infinite word **x** as input and produces an infinite word as output, by making transitions between states and outputting on each transition.

So we just make a simple finite-state transducer that records the parity of the number of 1's seen so far, transduce any given word using it, and get a new automaton that computes a new infinite word **y** that is the running sum (mod 2) of the number of 1's of the original word.

Then to determine if any block $\mathbf{x}[i..i+n-1]$ has an even number of 1's, we just compute the difference in parity between $\mathbf{y}[i]$ and $\mathbf{y}[i+n]$.

Thus, even if a language is not in $FO[+]$ there could be a way to check a property with `Walnut`.

Tips on writing Walnut formulas

Sometimes the obvious way of writing a predicate is not the best. For example, for

```
def tribsquarelen "?msd_trib n>0 &
   Ei At (t<n) => TR[i+t]=TR[i+n+t]":
```

required 893 seconds of CPU time and 85 gigabytes of RAM, with the largest intermediate automaton having 13,159,141 states. while

```
def tribsquarelen2 "?msd_trib n>0 & Ei Aj (j>=i & j<i+n) =>
   TR[j]=TR[j+n]":
```

terminates in less than one second, with largest intermediate automaton having 1742 states.

In general, one should try to minimize the number of different variables used, especially when used as indexes to automatic sequences.

Tips on writing Walnut formulas

Also, sometimes it doesn't pay to be general.

Remember that we defined a Walnut automaton occursf(i, j, m, n) for the property that the block $\mathbf{f}[i..j]$ appeared somewhere within the block $\mathbf{f}[m..n]$.

But then later on we only used it for the case $(j, k, i, k-1)$.

Therefore we could have gotten rid of n entirely by defining it to be $k-1$.

The more general automaton needs 297 states; the more specific one needs only 63 states.