Introduction to Walnut

Narad Rampersad and Manon Stipulanti

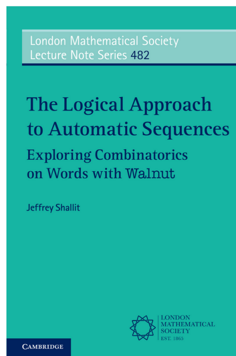
Walnut

- Walnut is a theorem prover that can prove theorems about automatic sequences.
- For example, it can prove classical results in combinatorics on words like:
  - The Thue–Morse word is overlap-free.
  - The squares in the Thue–Morse word have lengths of the form 2^n or $3 \cdot 2^n$.
  - The Fibonacci word contains exactly one palindrome of length n if n is even and exactly two palindromes of length n if n is odd.

Walnut book

Most of the material in this course is based on content from Jeffrey Shallit's book.

The Logical Approach To Automatic Sequences: Exploring Combinatorics on Words with `Walnut`



Jeffrey Shallit
The Logical Approach To Automatic Sequences: Exploring Combinatorics on Words with **`Walnut`**
Cambridge University Press, September 29 2022.

Theoretical basis for Walnut

Walnut is based on the theory of

- numeration systems
- finite automata and regular languages
- automatic sequences
- logic (extensions of Presburger arithmetic)

Integer base numeration systems

Let $k \geq 2$ be an integer. The base-k numeration system is based on the following result:

Theorem

Every non-negative integer n can be represented as a sum

$$n = \sum_{0 \leq i \leq t} a_{t-i} k^i,$$

where $a_i \in \{0, 1, \ldots, k-1\}$ for $i = 0, \ldots, t$ and $a_0 \neq 0$.

We then represent n in base k by the string $(n)_k = a_0 a_1 \cdots a_t$.

Representing several integers

- t-tuples of integers are represented as words over the alphabet $\{0, 1, \ldots, k-1\}^t$
- representations are padded with leading zeros so that they all have the same length
- For example, we represent the pair $(23, 6)$ in binary by

  $$[1,0][0,0][1,1][1,1][1,0];$$

  the first component gives the binary representation 10111 of 23 and the second component gives the (padded) binary representation 00110 of 6.

Fibonacci numeration

- In the Fibonacci (or Zeckendorf) numeration system, the place values are the Fibonacci numbers

$$1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- Let $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.
- Then every non-negative integer can be represented as a sum

$$\sum_{0 \leq i \leq t} a_{t-i} F_{i+2} \text{ with } a_i \in \{0, 1\}.$$

- However, the sum may not be unique.

Getting a unique representation

- e.g., $10 = 8 + 2$, $10 = 5 + 3 + 2$.
- To get a unique representation (up to leading 0's), we forbid $a_i = a_{i+1} = 1$ for all i.
- So, for example, the Fibonacci representation of 12 is $(12)_F = \mathtt{10101}$, since $12 = 8 + 3 + 1$.

Counting in base-Fibonacci

Here are the first few numbers written in the Fibonacci numeration system:

n	$(n)_F$	n	$(n)_F$
0	ϵ	9	10001
1	1	10	10010
2	10	11	10100
3	100	12	10101
4	101	13	100000
5	1000	14	100001
6	1001	15	100010
7	1010	16	100100
8	10000	17	100101

Finite automata

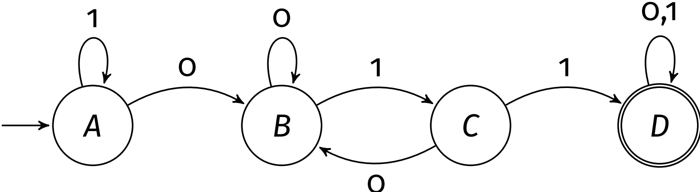
- A finite automaton is a computing machine that has a finite memory (i.e., a finite number of states.)
- It takes as input a finite string (word). Input is processed one symbol at a time, from left to right, starting from a designated initial state.
- The automaton has a transition function, which is a table that determines the next state based on the current state and current input symbol.

Acceptance

- In the basic model, states are designated final or non-final: the machine either accepts or rejects its input depending on whether it ends in a final or non-final state after reading the entire input.
- The language accepted by the automaton is the set of all strings accepted by the automaton.
- The class of all languages that can be accepted by finite automata is the class of regular languages.

Example of a finite automaton

This automaton accepts any binary string containing 011.



Non-determinism

- If the transition function maps the current state and current input symbol to a single state the automaton is deterministic.
- If it maps the current state and current input symbol to symbol to a set of states the automaton is non-deterministic.

Equivalence of the two models

- A classical result states that the two models of automata have the same computing power: they accept the same class of languages.
- However, converting a non-deterministic automaton to a deterministic one can result in an exponential blowup in the size (number of states) of the automaton.
- More precisely, given a non-deterministic automaton with n states, the smallest equivalent deterministic automaton could have, in the worst case, 2^n states.

Regular expressions

- Regular languages can also be described by regular expressions.
- A regular expression describes how to build a regular language by applications of three basic operations: union, concatenation, and Kleene closure.
- Represent the empty set by $\emptyset$, the language $\{\epsilon\}$ (the language containing the empty word) by ϵ, and the language $\{a\}$ by a.

Regular expressions

- Let L_1 and L_2 be languages.
- The union of L_1 and L_2 is written $L_1 \cup L_2$.
- The concatenation of L_1 and L_2 is the set

$$\{xy : x \in L_1, y \in L_2\}$$

  and is represented by juxtaposition; i.e., $L_1 L_2$.
- This can be extended in the obvious way to the concatenation of several languages.

Regular expressions

- The concatenation of a language L with itself k times is written L^k.
- The Kleene closure of a language L is the language

$$\bigcup_{k \geq 0} L^k$$

and is written L^*.

Examples

- The language of all binary strings is
$$(0 \cup 1)^*.$$
- The language of binary strings ending with 0's is
$$(0 \cup 1)^* 0.$$
- The language of binary strings containing 101 is
$$(0 \cup 1)^* 101 (0 \cup 1)^*.$$
- The language of binary strings avoiding 11's is
$$(\epsilon \cup 1)(0 \cup 01)^*.$$

Equivalence of finite automata and regular expressions

- There are standard constructions that, given finite automata for L_1 and L_2, produce automata for $L_1 \cup L_2$, $L_1 L_2$, and L_1^*.
- Similarly, there are algorithms that, given a finite automaton, can produce a regular expression defining the language accepted by the automaton.
- Therefore the two models of languages are equivalent.

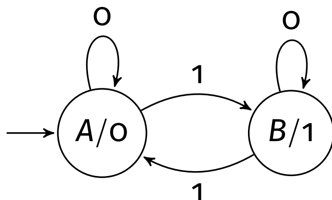
Finite automata with output

- We want to compute sequences over some arbitrary alphabet, so rather than designating states as final/non-final we associate an output symbol with each state.
- The output corresponding to a given input is the output associated with the last state reached after reading the input.
- The resulting model is the deterministic finite automaton with output (DFAO).

Automatic sequences

- Let $\mathbf{a} = (a_n)_{n \geq 0}$ be a sequence over a finite alphabet.
- To compute $\mathbf{a}$ we would like to provide n to a DFAO and have the DFAO output a_n.
- How do we represent n?
- Fix a base k and write n in base k.
- If there is a DFAO M that outputs a_n whenever M is given the base-k representation of n as input, then $\mathbf{a}$ is a k-automatic sequence.

The Thue–Morse sequence



This DFAO computes the Thue–Morse sequence

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

Parity definition of Thue–Morse

The Thue–Morse sequence

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

is defined by the simple rule

$$t_n = \begin{cases} 0 & \text{if binary rep. of } n \text{ has an even number of 1's} \\ 1 & \text{if binary rep. of } n \text{ has an odd number of 1's} \end{cases}$$

The paperfolding sequence

- another 2-automatic sequence defined by a similar rule is the paperfolding sequence

$$\mathbf{f} = 0010011000110 11\cdots$$

- the rule is: for $n \geq 1$, write $n = n'2^k$, where n' is odd. Then

$$f_n = \begin{cases} 0 & \text{if } n' \equiv 1 \pmod 4 \\ 1 & \text{if } n' \equiv 3 \pmod 4. \end{cases}$$

Uniform morphisms and codings

- Let Σ, Δ be finite alphabets.
- A morphism is a map $h : \Sigma^* \to \Delta^*$ (i.e., from the set of all words over Σ to the set of all words over Δ) satisfying $h(xy) = h(x)h(y)$ for all $x, y \in \Sigma^*$.
- A morphism is k-uniform if $h(a)$ has length k for all $a \in \Sigma$.
- It is uniform if it is k-uniform for some k.
- A 1-uniform morphism is a coding.

Morphic sequences

- If $h(a)$ begins with a, the sequence of iterates

$$a, h(a), h^2(a), h^3(a)$$

  has the property that $h^i(a)$ is a prefix of $h^{i+1}(a)$.
- The limit sequence $h^\omega(a)$ is a fixed point of h.
- It is a called a purely morphic sequence.
- The image of a purely morphic sequence by a coding is a morphic sequence.

The Thue–Morse morphism

- $\mu : \{0,1\}^* \to \{0,1\}^*$ defined by

$$\begin{aligned} 0 &\to 01 \\ 1 &\to 10, \end{aligned}$$

  is 2-uniform.
- Iterating on 0

$$0 \to 01 \to 0110 \to 01101001 \to 0110100110010110 \to \cdots$$

  gives the Thue–Morse sequence

$$\mathbf{t} = \mu^\omega(0) = 0110100110010110\cdots$$

The characteristic sequence of 2^n

- Iterate the 2-uniform morphism

$$a \to ab, b \to bc, c \to cc$$

  to get the infinite sequence

$$abbcbcccbcccccccbcccccccccccccccbcc \cdots .$$

- Now apply the coding $a, c \to 0$; $b \to 1$:

$$01101000100000001000000000000000100 \cdots .$$

- We get the characteristic sequence of the powers of 2.

k-automatic sequences and k-uniform morphisms

Theorem (Cobham)

A sequence $\mathbf{a}$ is k-automatic if and only if $\mathbf{a} = h(g^\omega(a))$ for some k-uniform morphism g, some coding h, and some letter a.

Decidable properties

- The Thue–Morse word was introduced by Thue in 1906, who proved that it has many interesting combinatorial properties.
- Are there algorithms to decide if an automatic sequence
  - is aperiodic?
  - is recurrent?
  - avoids repetitions?
  - etc.
- To answer this question we need a third characterization of k-automatic sequences.

A logic-based characterization

- Another characterization of k-automatic sequences is based on an an extension of Presburger arithmetic.
- Presburger arithmetic is the first-order theory of the structure $\langle \mathbb{N}, + \rangle$; i.e., the first-order theory of the natural numbers with addition.

Presburger arithmetic

- $\langle \mathbb{N}, + \rangle$ is sometimes written $\langle \mathbb{N}, +, <, 0, 1 \rangle$, where the "less than" predicate is explicitly included, as well as the constants 0 and 1.
- In fact we can define $x = 0$ by $\forall y,\ x + y = y$, and then
- $x < y$ by $\exists t,\ \neg(t = 0) \wedge x + t = y$, and then
- $y = 1$ by $(0 < y) \wedge (\forall z,\ (z < y) \Rightarrow z = 0)$.

Definable sets

- A set $X \subseteq \mathbb{N}$ is definable in $\langle \mathbb{N}, + \rangle$ if there is a first-order formula φ of $\langle \mathbb{N}, + \rangle$ such that

$$X = \{n \in \mathbb{N} : \langle \mathbb{N}, + \rangle \models \varphi(n)\}$$

- The sets definable in Presburger arithmetic are finite unions of arithmetic progressions.

Extending Presburger arithmetic

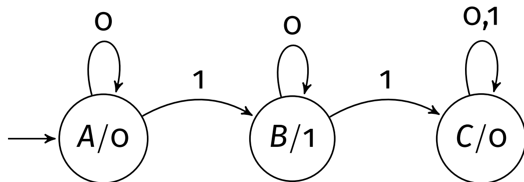
- Presburger arithmetic is a decidable theory; i.e., there is an algorithm, that, given a first-order formula with no free variables, will decide whether the formula is TRUE or FALSE.
- On the other hand, Presburger arithmetic is a fairly weak theory.
- It is possible to extend Presburger arithmetic as follows and still have a decidable theory.
- Let $V_k(x)$ denote the largest power of k that divides x.

Extending Presburger arithmetic

- (Büchi–Bruyère): A sequence $\mathbf{a}$ is k-automatic if it is definable in the logical structure $\langle \mathbb{N}, +, V_k \rangle$.
- Let $\mathbf{a}^{-1}(b)$ denote the set of positions of occurrences of b in $\mathbf{a}$.
- Then there exists a first-order formula φ_b of $\langle \mathbb{N}, +, V_k \rangle$ such that

$$\mathbf{a}^{-1}(b) = \{n \in \mathbb{N} : \langle \mathbb{N}, +, V_k \rangle \models \varphi_b(n)\}.$$

An automaton for the powers of 2



Defining the powers of 2 using logic

- The characteristic sequence **a** of the powers of 2 has a simple definition in this formulation:

$$
\begin{aligned}
\mathbf{a}^{-1}(1) &= \{n \in \mathbb{N} : \langle \mathbb{N}, +, V_k \rangle \models (V_2(n) = n)\} \\
\mathbf{a}^{-1}(0) &= \{n \in \mathbb{N} : \langle \mathbb{N}, +, V_k \rangle \models \neg(V_2(n) = n)\}
\end{aligned}
$$

Decidability

Theorem (Bruyère 1985)

The first order theory of $\langle \mathbb{N}, +, V_k \rangle$ is decidable.

A consequence of decidability

Theorem (Charlier, R., Shallit 2011)

If we can express a property of a k-automatic sequence $\mathbf{x}$ using quantifiers, logical operations, integer variables, the operations of addition, subtraction, indexing into $\mathbf{x}$, and comparison of integers or elements of $\mathbf{x}$, then this property is decidable.

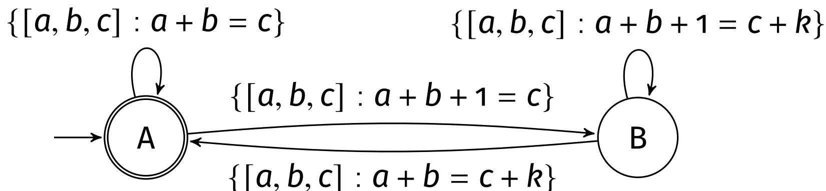
Implementing the decidability algorithm

Representing values of variables:

- alphabet is $\Sigma_k = \{0, 1, \ldots, k-1\}$
- integers are represented by their base-k expansions (most-significant-digit first)
- t-tuples of integers are represented as words over the alphabet Σ_k^t
- representations are padded with leading zeros so that they all have the same length
- digits of the integers in the t-tuple are therefore read in parallel

Addition

The addition relation $x + y = z$ in base k can be computed by the automaton below:



State A corresponds to the case where no carry is pending; state B corresponds to the case where a carry is pending.

Arithmetic operations

- the equality relation is trivial to check with an automaton
- the less than relation can also be checked with an automaton
- multiplication by a constant can be viewed as repeated addition, so it is permitted in our logic;
- however, it is important to note that multiplication of variables is not permitted.
- Indeed, a classical result of logic states that any first-order theory of arithmetic in which both addition and multiplication is definable is undecidable.

Logical operations

- the existential quantifier is implemented using non-determinism (the resulting non-deterministic automaton must then be determinized)
- the universal quantifier is implemented using complementation and the existential quantifier
- boolean operations ($\wedge, \vee, \Rightarrow, \Leftrightarrow$, etc.): standard automata constructions for $\cap, \cup$, etc.

Runtime of the algorithm

- Each alternation of quantifiers in the logical formula results in a conversion from a non-deterministic automaton (NFA) to a deterministic automaton (DFA).
- If the NFA has n states, in the worst case, the DFA could have 2^n states.
- The final automaton could have a number of states equal to a tower of exponentials as high as the number of quantifier alternations in the formula.
- In practice, the automata we get are not this big.
- However, some formulas may require a large amount of RAM to hold the intermediate automata.

Fibonacci-automatic sequences

- Let $\mathbf{a} = (a_n)_{n \geq 0}$ be a sequence.
- If there is a DFAO M that outputs a_n whenever M is given the Fibonacci representation of n as input, then $\mathbf{a}$ is a Fibonacci-automatic sequence.
- Furthermore, inputs with consecutive 1's are rejected or not considered.

The Fibonacci word

- The most important Fibonacci-automatic sequence is the Fibonacci word

$$\mathbf{f} = 0100101001001010010100100 \cdots$$

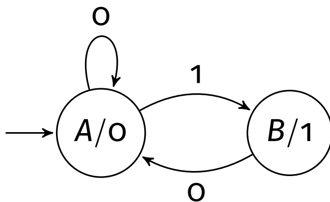
- This word is most commonly defined as the fixed point of the non-uniform morphism

$$0 \rightarrow 01, \quad 1 \rightarrow 0.$$

- It can be equivalently defined as follows: the n-th term of $\mathbf{f}$ is equal to the rightmost bit of the Fibonacci-representation of n.

The Fibonacci DFAO

This can easily be checked with the following DFAO:



Decision algorithm for Fibonacci-automatic

- As with k-automatic sequences, we can also decide the truth of first-order logical statements about Fibonacci-automatic sequences.
- The equivalent of $V_k(x)$ is $V_u(x)$, which is defined to be the least Fibonacci number that appears in the Fibonacci base representation of x.

Addition in the Fibonacci numeration

- Most of the various relations and logical operations can be checked with automata in the same way, but the addition relation is a bit more complicated.
- There is a (17 state) automaton that computes the addition relation $x + y = z$, so everything described previously for integer bases can also be done in the Fibonacci numeration system.

Walnut

- This has all been implemented by Hamoon Mousavi in a Java software package called Walnut.
- The user inputs a logical formula describing some property of a given automatic sequence.
- If the formula has no free variables, Walnut will determine whether the formula is true or false and output TRUE or FALSE, accordingly.
- If the formula has i free variables, Walnut outputs an automaton accepting the representations (in the appropriate numeration system) of the i-tuples of natural numbers that satisfy the formula.

Walnut syntax

The Walnut symbols for the first-order logic symbols are:

- `E` for $\exists$
- `A` for $\forall$
- `~` for logical negation
- `=>` for logical implication, `&` for logical AND, `|` for logical OR, and `<=>` for logical IFF.
- Both `E` and `A` can be followed by a single variable name, or a list of variables.

Specifying the numeration system

- The numeration system is specified as `?msd_k` for base-k or `?msd_fib` for Fibonacci numeration.
- `msd`, of course, refers to most-significant-digit first; `lsd`, for least-significant-digit first, is also possible.
- If the numeration system is not specified, the default is `msd_2`.

Regular expressions for certain properties

An integer n is even if there exists $k \in \mathbb{N}$ such that $n = 2k$.
Walnut:

```
def iseven "Ek n=2*k":
```

produces an automaton called ISEVEN that accepts the even numbers.

The property of being even can also be defined by a regular expression:
Expansions of even numbers in base 2 end with a 0.
Walnut:

```
reg evenreg msd_2 "()|0*1(1|0)*0":
```

We can check the equivalence with

```
eval evencheck "An $iseven(n) <=> $evenreg(n)":
```

which returns `TRUE`.

Morphisms, codings, and fixed points

- The twisted Thue–Morse sequence,

$$\mathbf{ttm} = 00100110100101\cdots,$$

  is defined by counting the number of 0's, modulo 2, in base-2 representations.
- It is built-in to Walnut (as `TTM`).
- Show that **ttm** is the image, under the coding $0, 1 \mapsto 0$ and $2 \mapsto 1$, of the fixed point of the morphism $0 \mapsto 01$, $1 \mapsto 21$, and $2 \mapsto 12$.

Morphisms, codings, and fixed points

Walnut:

```
morphism fmorph "0->01 1->21 2->12":
promote TTM1 fmorph:
morphism coding "0->0 1->0 2->1":
image TTM2 coding TTM1:
eval test "An TTM[n]=TTM2[n]":
```

returns `TRUE`

Explanation of the Walnut commands

- the `promote` command converts a morphism to a DFAO
- the `image` command applies a morphism to a DFAO and produces a new DFAO

Finding positions of symbols

- The period-doubling word

$$\mathbf{pd} = 101110101011101110111010\cdots$$

  is the fixed point of the morphism that maps $0 \to 11$ and $1 \to 10$.
- It is built-in to Walnut (as `PD`).
- It is obvious from the definition that every even position contains a 1.
- Let's check this with Walnut.

Logical formula

- The formula asserting that every even position of **pd** contains a 1 is

$$\forall i,\ \text{ISEVEN}(i) \Rightarrow \mathbf{pd}[i] = 1$$

- In Walnut this becomes:

```
def iseven "Ek n=2*k":
eval PD_even "Ai $iseven(i) => PD[i]=@1":
```

Walnut outputs TRUE, verifying our claim.

Now let’s check that **pd** does not contain 00. We can probably go straight to the Walnut commands:

```
eval PD_00 "~(Ei PD[i]=@0 & PD[i+1]=@0)":
```

Walnut outputs TRUE, verifying our claim.

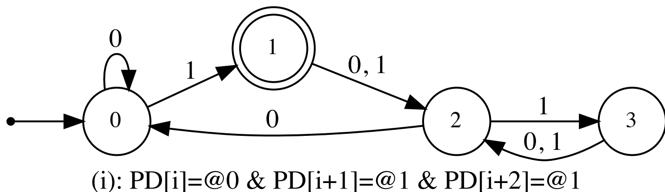
Next let's find all the occurrences of 011 in

$$\mathbf{pd} = 10111010101110111011101 0\cdots$$

```
eval PD_011 "PD[i]=@0 & PD[i+1]=@1 & PD[i+2]=@1":
```

This formula has a free variable, *i*, so instead of outputting TRUE or FALSE, Walnut instead returns an automaton accepting the binary representations of all *i* for which the formula is true.

The result is the automaton



We can see that, for example, this automaton accepts the string `1001`, which is the binary representation of 9, so there is an occurrence of 011 in **pd** at position 9.

Periodicity

- A sequence **a** is ultimately periodic if and only if

$$\exists p, n \ (p \geq 1) \wedge (n \geq 0) \wedge \forall i \ (i \geq n) \Rightarrow a_i = a_{i+p}$$

- Clearly, the fixed point (starting with 0) of the map $0 \rightarrow 010$, $1 \rightarrow 101$ is periodic. Let us check this with Walnut.

Periodicity

Walnut:

```
morphism m "0->010 1->101":
promote M_word m
eval M_per "?msd_3 Ep,n (p>=1) & (Ai (i>=n) =>
  M_word[i]=M_word[i+p])":
```

outputs TRUE. (Notice the `?msd_3`!)

Overlaps

- Iterated morphism constructions, such as the Thue–Morse word, are typically used to generate words with certain combinatorial properties.
- We will now look at several of the most common such combinatorial properties.
- An overlap is a word (such as `entente`) of length $2n+1$ and period n for some $n \geq 1$,
- A sequence $\mathbf{a}$ contains an overlap if and only if

$$\exists i, n \ (n \geq 1) \wedge \forall j \ (j \leq n) \Rightarrow a_{i+j} = a_{i+j+n}$$

Walnut:

```
eval tm_ofree "?msd_2 Ei,n (n>=1) & (Aj (j<=n) =>
  T[i+j]=T[i+j+n])":
```

outputs FALSE, so the Thue–Morse word is overlap-free.

Squares

- Over larger alphabets there are infinite words avoiding squares.
- An square is a non-empty word (such as `froufrou`) of the form xx.
- Equivalently it is a word of length $2n$ and period n for some $n \geq 1$.
- A sequence $\mathbf{a}$ contains a square if and only if

$$\exists i, n \ (n \geq 1) \wedge \forall j \ (j < n) \Rightarrow a_{i+j} = a_{i+j+n}$$

A squarefree word

We can construct a 2-automatic squarefree word as follows: let h be the morphism that maps

$$0 \to 01, \quad 1 \to 20, \quad 2 \to 23, \quad 3 \to 02$$

and let g be the coding

$$0 \to 0, \quad 1 \to 1, \quad 2 \to 2, \quad 3 \to 1.$$

Then

$$g(h^{\omega}(0)) = 012021012102\cdots$$

is squarefree.

Walnut:

```
morphism h_morph "0->01 1->20 2->23 3->02":
promote H_word h_morph:
morphism g_morph "0->0 1->1 2->2 3->1":
image GH g_morph H_word:
eval GHsq "?msd_2 Ei,n (n>=1) & (Aj (j<n) =>
  GH[i+j]=GH[i+j+n])":
```

outputs FALSE, so the word is square-free.

Cubes

- An cube is a non-empty word of the form xxx.
- Equivalently it is a word of length $3n$ and period n for some $n \geq 1$.
- A sequence $\mathbf{a}$ contains a cube if and only if

$$\exists i, n \ (n \geq 1) \wedge \forall j \ (j < 2n) \Rightarrow a_{i+j} = a_{i+j+n}$$

A cubefree word

Let us show that the infinite word

$$f^{\omega}(0) = 01001101100100101001 \cdots$$

obtained by iterating the morphism f that maps

$$0 \longrightarrow 01001, \quad 1 \longrightarrow 10110$$

is cubefree.

Walnut:

```
morphism f_morph "0->01001 1->10110":
promote F_word f_morph:
eval F_word_cube "?msd_5 Ei,n (n>=1) & (Aj (j<2*n)
  => F_word[i+j]=F_word[i+j+n])":
```

outputs FALSE, so the word is cube-free.

Fractional powers

- Squares (xx) and cubes (xxx) are called 2-powers and 3-powers.
- For any integer $k \geq 2$, a k-power is defined in the obvious way.
- For a rational number $p/q \geq 1$, a (p/q)-power of period n is a word of length $(p/q)n$ and period n.
- A $(p/q)^+$-power of period n is a word of length $> (p/q)n$ and period n.

Fractional powers

- e.g., 1001001 is a 7/3-power of period 3 and a 2^+-power of period 3.
- A sequence **a** contains a (p/q)-power if and only if

$$\exists i, n \ (n \geq 1) \wedge \forall j \ (j \geq i \wedge j < i + (p/q - 1)n) \Rightarrow a_j = a_{j+n}.$$

- A sequence **a** contains a $(p/q)^+$-power if and only if

$$\exists i, n \ (n \geq 1) \wedge \forall j \ (j \geq i \wedge j \leq i + (p/q - 1)n) \Rightarrow a_j = a_{j+n}.$$

Fractional powers

We are only allowed to multiply by postive integer constants in our logical formulas (not rational numbers), so we have to rewrite the inequality

$$j < i + (p/q - 1)n$$

as

$$qj < qi + (p - q)n.$$

A word avoiding $(8/3)^+$-powers

The infinite word

$$f^\omega(0) = 01001101100100101001 \cdots$$

we constructed previously contains the $(8/3)$-power 10010010 but we can show that it avoids $(8/3)^+$-powers. Walnut:

```
eval F_word_83 "?msd_5 Ei,n (n>=1) &
  (Aj (j>=i & 3*j<=3*i+5*n) =>
  F_word[j]=F_word[j+n])":
```

outputs FALSE.

Patterns

- Squares (xx) and cubes (xxx) are examples of patterns with a single variable.
- We can also define patterns with several variables (x, y, z, etc.)
- For example, the word 201012 is an instance of the pattern $xyyx$.
- Let us show that the Thue–Morse word contains no instance of the pattern $xyyxyy$.

Equality of factors

First-order formula: $\mathbf{x}[i..i+n-1]$ and $\mathbf{x}[j..j+n-1]$ are equal iff

$$\textsc{FactorEq}(i,j,n) := \forall t < n, \mathbf{x}[i+t] = \mathbf{x}[j+t]$$

Walnut: in the Thue–Morse word $\mathbf{t}$

```
def tmfactoreq "At t<n => T[i+t]=T[j+t]":
```

Outputs the next 13-state automaton.

(i,j,n): At t<n => T[i+t]=T[j+t]

The Thue–Morse word avoids *xyyxyy*

Walnut:

```
def tmfactoreq "?msd_2 At t<n => T[i+t]=T[j+t]":
def xyyxyy "?msd_2 Ei,m,n (m>=1) & (n>=1) &
  $tmfactoreq(i,i+m+2*n,m) &
  $tmfactoreq(i+m,i+m+n,n) &
  $tmfactoreq(i+m,i+2*m+2*n,n) &
  $tmfactoreq(i+m,i+2*m+3*n,n)":
```

outputs FALSE.

Novel factors

An occurrence of a factor is novel if it is the first occurrence of that factor in $\mathbf{x}$.

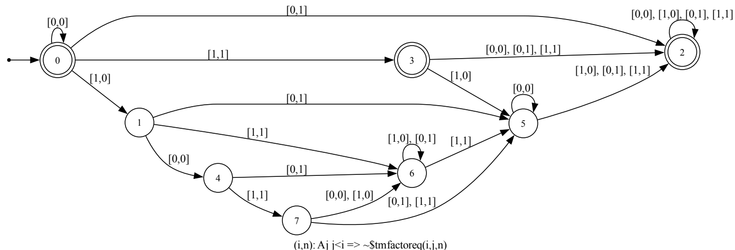
First-order formula: $\mathbf{x}[i..i+n-1]$ is a novel factor iff $\forall j (j < i) \Rightarrow \neg \textsc{FactorEq}(i, j, n)$

What are the positions and lengths of all novel factors of the Thue–Morse sequence $\mathbf{t}$?

Walnut:

```
def tmnovelfactor "Aj j<i => ~$tmfactoreq(i,j,n)":
```

Output:



Conjugates

Two words are conjugates if one is a rotation of the other.

First-order logical formula: $\mathbf{x}[j..j+n-1]$ is a conjugate of $\mathbf{x}[i..i+n-1]$ iff $\exists t \leq n$ such that

$$\mathbf{x}[j..j+n-1] = \mathbf{x}[i+t..i+n-1]\mathbf{x}[i..i+t-1]$$

which translates into

$$\begin{aligned} \textsc{Conj}[i,j,n] := \exists t (t \leq n) &\wedge \textsc{FactorEq}(j, i+t, n-t) \\ &\wedge \textsc{FactorEq}(i, (j+n)-t, t) \end{aligned}$$

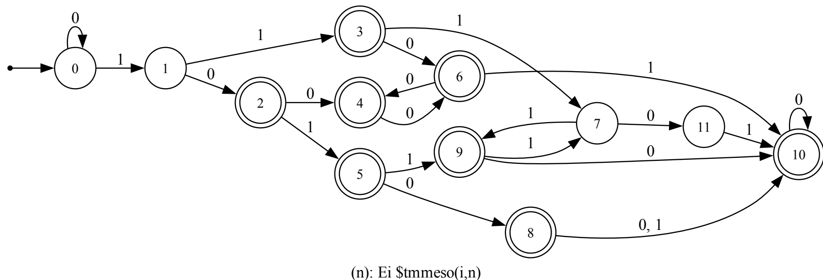
A mesosome is a word of the form xx' where x' is a conjugate of x and $x \neq x'$.

For which integers n is there a mesosome factor $\mathbf{t}[i..i+2n-1]$ of the Thue–Morse word?

Walnut:

```
def tmconj "Et t<=n & $tmfactoreq(j,i+t,n-t) &
   $tmfactoreq(i,(j+n)-t,t)":
def tmmeso "$tmconj(i,i+n,n) &
   ~$tmfactoreq(i,i+n,n)":
def tmmesolength "Ei $tmmeso(i,n)":
```

Output:



(n): Ei $tmmeso(i,n)

The automaton accepts and rejects infinitely many *n*.

Primitivity

A word is primitive if it is nonempty and a non-power.

Theorem

A word w is primitive if and only if no nontrivial rotation of w equals w.

First-order logical formula: $\mathbf{x}[i..i+n-1]$ is primitive iff

$$\begin{aligned}\textsc{Prim}(i,n) :=& \neg(\exists j (j > 0) \wedge (j < n) \\ & \wedge \textsc{FactorEq}(i, i+j, n-j) \\ & \wedge \textsc{FactorEq}(i, (i+n)-j, j))\end{aligned}$$

In the period-doubling word

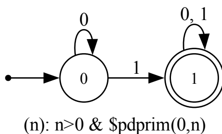
$$\mathbf{pd} = 1011101010111011101110 10 \cdots ,$$

what are the lengths of the primitive prefixes?

Walnut:

```
def pdfactoreq "At t<n => PD[i+t]=PD[j+t]":
def pdprim "~(Ej j>0 & j<n & $pdfactoreq(i,i+j,n-j)
    & $pdfactoreq(i,(i+n)-j,j))":
def pdprimlength "n>0 & $pdprim(0,n)":
```

Output:



This automaton accepts the binary representations of all positive integers, so every prefix of **pd** is primitive.

Lexicographic order

Let Σ be an alphabet. If it is ordered, then we can extend this total order from letters to words. We write $w <_d x$ if

- w is a proper prefix of x; or
- there exist words y, z, z' and letters $a < b$ such that $w = yaz$ and $x = ybz'$.

This is called the lexicographic or dictionary order. We write $w \leq x$ if either $w <_d x$ or $w = x$.

First-order logical formula: $\mathbf{x}[i..i+m-1] <_d \mathbf{x}[j..j+n-1]$ iff

$$
\begin{aligned}
\textsc{LexLess}(i,j,m,n) :=& \textsc{Pref}(i,j,m,n) \\
& \vee (\exists t < m,n : \mathbf{x}[i..i+t-1] = \mathbf{x}[j..j+t-1] \\
& \wedge \mathbf{x}[i+t] = 0 \\
& \wedge \mathbf{x}[j+t] = 1)
\end{aligned}
$$

where

$$
\textsc{Pref}(i,j,m,n) := (m < n) \wedge \textsc{FactorEq}(i,j,m)
$$

In the Thue–Morse word **t**, show that

$$\mathbf{t}[i..i+n-1] <_d \mathbf{t}[j..j+n-1] \Rightarrow \mathbf{t}[2i..2i+2n-1] <_d \mathbf{t}[2j..2j+2n-1].$$

Walnut:

```
def tmfactoreq "At t<n => T[i+t]=T[j+t]":
def tmpref "m<n & $tmfactoreq(i,j,m)":
def tmlesst "$tmpref(i,j,m,n) |
   (Et t<m & t<n & (Al (l<t) => T[i+l]=T[j+l])
   & T[i+t]=@0 & T[j+t]=@1)":
def tmtest "Ai,j,n $tmlesst(i,j,n,n) =>
   $tmlesst(2*i,2*j,2*n,2*n)":
```

returns TRUE.

Runs

A run is a nonempty block consisting of repetitions of a single symbol a. A run is maximal if it cannot be extended to the left or right.

First-order logical formula: $\mathbf{x}[i..i+n-1]$ is a maximal run of n a's iff

$$\begin{aligned} \textsc{IsRun}(i,n) := (n \geq 1) &\wedge (\forall t < n, \mathbf{x}[i+t] = a) \\ &\wedge \mathbf{x}[i+n] \neq a \\ &\wedge (i = 0 \vee \mathbf{x}[i-1] \neq a) \end{aligned}$$

First-order logical formula: there is a maximal run of $n \geq 1$ letters a in $\mathbf{x}$ iff

$$\exists i, \textsc{IsRun}(i, n)$$

First-order logical formula: there are arbitrarily long finite maximal runs of letters a in $\mathbf{x}$ iff

$$\forall m, \exists i, n, (n > m) \wedge \textsc{IsRun}(i, n)$$

- Are there arbitrarily large maximal runs of symbols in the Cantor sequence?
- The Cantor integers are those integers whose base-3 representation contains no letter 1.
- The Cantor sequence $\mathbf{ca} = 101000101000000000 \cdots$ is the corresponding characteristic sequence.

Walnut:

```
def carun0 "?msd_3 n>=1 & (At t<n => CA[i+t]=@0)
    & CA[i+n]!=@0 & (i=0|CA[i-1]!=@0)":
def carun1 "?msd_3 n>=1 & (At t<n => CA[i+t]=@1)
    & CA[i+n]!=@1 & (i=0|CA[i-1]!=@1)":
eval cantor0 "?msd_3 Am Ei,n (n>m) & $carun0(i,n)":
eval cantor1 "?msd_3 Am Ei,n (n>m) & $carun1(i,n)":
```

The first query returns TRUE, while the second returns FALSE. Hence there are arbitrarily long blocks of 0's, but not 1's, in this sequence.

Palindromes

A length-n word w is a palindrome if $w[i] = w[n-1-i]$ for all $i \in \{0, 1, \ldots, n-1\}$.

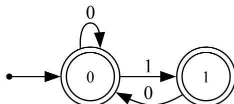
What are the length of the palindromic factors in the Fibonacci word $\mathbf{f}$?

Palindromes

Walnut:

```
def fibpal "?msd_fib Ei At t<n =>
    F[i+t] = F[(i+n)-(t+1)]":
```

Output:



(n): ?msd_fib Ei At t<n => F[i+t] = F[(i+n)-(t+1)]

This automaton accepts every word in the Fibonacci numeration language, so we see that there is a palindrome of every length in the Fibonacci word.

Limitations

- Now that we have seen many things that Walnut can do, let's consider its limitations.
- We can consider two questions:
  - What types of sequences can Walnut be applied to?
  - What properties of the sequence can be expressed in our first-order logic?

Types of sequences

- Walnut can be used with a wide variety of morphic sequences.
- Recall: k-uniform morphic $\iff$ k-automatic
- Can also be used with some non-uniformly morphic sequences.
- Crucial requirement: the sequence must have an underlying numeration system for which addition can be computed with an automaton.
- e.g., Fibonacci, Tribonacci, Pell, etc.

A morphic sequence not suitable for Walnut

Consider the morphism

$$a \to abcc, \quad b \to bcc, \quad c \to c.$$

Its fixed point, starting with a,

$$\mathbf{s} = abccbccccbccccccbcccccccc \cdots$$

encodes, in the position of the b's, the squares, $1, 4, 9, 16, \ldots$.

A morphic sequence not suitable for Walnut

- However, with the squares we can define multiplication.
- Recall: if addition and multiplication are both definable, the theory is undecidable.

Defining multiplication from the squares

Let S be a predicate for the squares. We can define $y = x^2$ by

$$S(y) \wedge S(y + 2x + 1) \wedge \neg\exists z(S(z) \wedge y < z \wedge z < y + 2x + 1)$$

Then multiplication $z = xy$ can be defined from $(x + y)^2 = x^2 + 2z + y^2$.

Which properties are expressible?

- Let's now consider which properties of an infinite word $\mathbf{w}$ can be verified with Walnut.
- By property, we simply mean a language L over the alphabet Σ of $\mathbf{w}$.
- We want to determine the pairs (i, n) such that $\mathbf{w}[i..i+n-1] \in L$.

Examples we already know

We have already seen many properties that can be tested:

- Squares: $\{xx : x \in \Sigma^*, |x| \geq 1\}$
- Palindromes: $\{x \in \Sigma^* : x = x^R\}$
- Primitive words:
  $\{x : \text{ there are no } y \text{ and } n \geq 2 \text{ such that } x = y^n\}$
- etc.

The class FO[+]

The entire class of such languages is called FO[+]. It is a somewhat mysterious class of languages. It includes some "complicated" languages, like the ones we have just mentioned, but provably does not contain the following languages:

- Parity:
  $\{x \in \{0,1\}^* : x \text{ contains an even number of 1's}\}$
- Dyck: $\{x \in \{0,1\}^* :$
  $x \text{ represents a balanced string of parentheses}\}$
- Abelian Squares: $\{xx' \in \Sigma^* : x' \text{ is an anagram of } x\}$

Inexpressible properties

These properties (Parity, Dyck, Abelian Squares, etc.) therefore cannot be directly expressed as a first-order predicate in Walnut. However, there are some other tricks that can be used to compute Parity, for example, for an automatic sequence using Walnut (see Shallit's book).

Balanced words

- Sometimes one definition of a property is not expressible, but another equivalent definition is. An example is balanced words.
- A word x is called balanced if the inequality $||y|_a - |z|_a| \leq 1$ holds for all identical-length factors y, z of x and all letters a of the alphabet
- (Here $|y|_a$ denotes the number of occurrences of the letter a in the word y.)

Balanced words

- This definition appears (much like Parity or Abelian Squares) to not be expressible, since it involves counting occurrences of letters.
- However, Coven and Hedlund gave a characterization in the case of the binary alphabet that is expressible.
- A binary word w is unbalanced if and only if there exists a word v such that both $0v0$ and $1v1$ are factors of w.
- In the exercises you will show that every factor of the Fibonacci word is balanced.

Enumeration

- Walnut can also be used to solve certain enumeration problems.
- A typical example is finding the subword complexity of an automatic sequence: i.e., the number of distinct subwords (factors) of length n.
- The enumeration is given in terms of a linear representation.

Linear representations

Let $f : \mathbb{N} \to \mathbb{N}$. A (binary) linear representation for $f(i)$ is:

- an integer row vector v,
- an integer column vector w, and
- a pair of integer matrices M_0 and M_1, such that

$$f(i) = vM_{i_{\ell-1}}M_{i_{\ell-2}} \cdots M_{i_0}w,$$

where $i_{\ell-1}i_{\ell-2} \cdots i_0$ is the binary representation of i.

For example,

$$v = [1, 0, 0, 0], \quad w^T = [0, 0, 1, 1]$$

and

$$M_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

is a linear representation for the function $f(n) = n(n+1)/2$. We can compute, for example, that $vM_1M_0M_0w = 10$ and $f(4) = 4(5)/2 = 10$.

Linear representations in Walnut

- Walnut can compute linear representations to count certain things.
- Given a formula with free variables $i_1, i_2, \ldots, i_r$ and n, Walnut can compute a linear representation for the function of n that counts the number of values of $(i_1, i_2, \ldots, i_r)$ that satisfy the formula.

Linear representations in Walnut

For example, it is easy to see that the number of pairs (i, j) that satisfy $i < j$ and $j \leq n$ is $n(n+1)/2$. The Walnut command

```
eval pairmat n "i<j & j<=n":
```

therefore produces the linear representation (in MAPLE format) we previously gave for $f(n) = n(n+1)/2$.

Counting 1's in a prefix of length n

Walnut has the characteristic sequence of powers of 2

$$01101000100000001000000000000000100\cdots$$

built in as word `P2`. Let's count the number of 1's in a prefix of length n. The Walnut command

```
eval P2ones n "i<n & P2[i]=@1":
```

produces a linear representation for this number as a function of n.

(For technical reasons, the v vector returned by Walnut has to be replaced with vM_0^ℓ, where ℓ is the dimension of v.)

The linear representation

The linear representation we get is

$$v = [1, 0, 0, 0], \quad w^T = [0, 0, 0, 1]$$

and

$$M_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad M_1 \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can compute, for example, that $vM_1M_0M_0M_1w = 4$ and there are 4 1's in the prefix of P2 of length 9.

k-synchronized functions

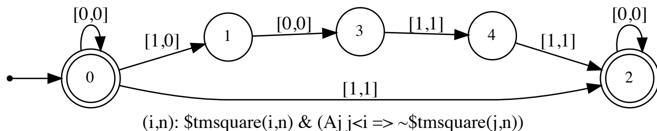
- A function $f(n)$ is k-synchronized if the set of base-k representations of the pairs $(n, f(n))$ is accepted by a finite automaton.
- As always, we read the pairs of base-k digits of n and $f(n)$ in parallel.

Squares of period n in **t**

As an example, let $f(n)$ be the position of the first occurrence of a square of period n in the Thue–Morse word **t**. An automaton accepting the pairs $(f(n), n)$ can easily be computed with the following Walnut code:

```
def tmfactoreq "At t<n => T[i+t]=T[j+t]":
def tmsquare "$tmfactoreq(i,i+n,n)":
eval tmfirstsquare "$tmsquare(i,n) & (Aj j<i =>
  ~$tmsquare(j,n))":
```

The result is the automaton



This automaton accepts (ignoring leading $[0,0]$'s)

$$[1,1]\,[0,0]^* \text{ and } [1,0]\,[0,0]\,[1,1]\,[1,1]\,[0,0]^*$$

From this we deduce that the first occurrence of a square of period 2^k is at position 2^k and the first occurrence of a square of period $3 \cdot 2^k$ is at position $11 \cdot 2^k$.

Abelian properties

- Abelian properties of words are based on the abelian equivalence relations: $x \sim y$ if x and y are anagrams of each other.
- Equivalently, for every alphabet symbol a, the words x and y contains exactly the same number of a's.
- An abelian square, for example, is a word xy where $x \sim y$.
- Abelian properties of words are generally not testable with Walnut, but in some cases this can done.

Abelian complexity

The abelian complexity of an infinite sequence $\mathbf{x}$ counts the number of distinct subwords, up to abelian equivalence of factors.

For example, the abelian complexity of the Thue–Morse sequence $\mathbf{t}$ at $n = 2$ is 3, because among the four factors 00, 01, 10, and 11, the factors 01 and 10 are (abelian) equivalent.

Implement a procedure to check whether two factors of the Thue–Morse sequence $\mathbf{t}$ are abelian equivalent.

Step 1

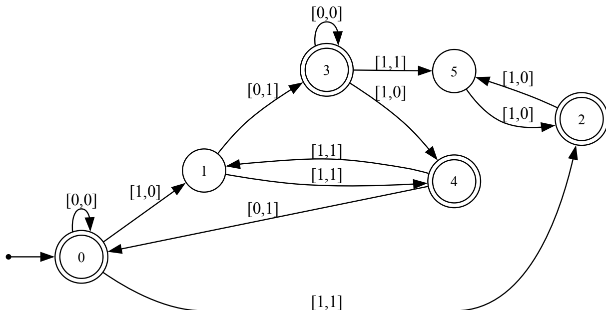
Show that the function $n \mapsto |\mathbf{t}[0..n-1]|_0$ counting the number of 0's in a length-n prefix of $\mathbf{t}$ is synchronized.

To do so, observe that if n is even, then the length-n prefix of $\mathbf{t}$ has exactly $\frac{n}{2}$ letters 0. If n is odd, then this prefix has $\frac{n-1}{2}$ letters 0, plus one more if $\mathbf{t}_{n-1} = 0$.

Walnut:

```
def tmpref0 "Er,t n=2*t+r & r<2 & (r=0 => s=t)
    & ((r=1 & T[n-1]=@1) => s=t)
    & ((r=1 & T[n-1]=@0) => s=t+1)":
```

Output:



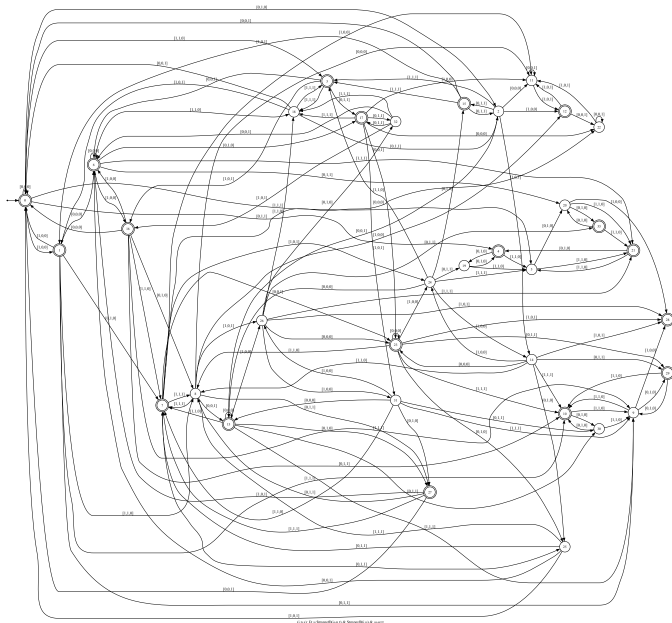
Step 2

From this synchronized function, obtain a similar one for arbitrary factors: $(i, n) \mapsto |\mathbf{t}[i..i+n-1]|_0$, and also one that counts letters 1: $(i, n) \mapsto |\mathbf{t}[i..i+n-1]|_1$.

Walnut:

```
def tmfac0 "Et,u $tmpref0(i+n,t)
    & $tmpref0(i,u) & s+u=t":
def tmfac1 "Er $tmfac0(i,n,r) & n=r+s":
```

Output:



Step 3

From the DFA `tmfac0`, construct a DFA that on input i, j, n tests whether $\mathbf{t}[i..i+n-1]$ and $\mathbf{t}[j..j+n-1]$ are abelian equivalent.

Note that, since $\mathbf{t}$ is a binary word, it suffices to check that $|\mathbf{t}[i..i+n-1]|_0 = |\mathbf{t}[j..j+n-1]|_0$.

Walnut:

```
def tmabelfaceq "Es $tmfac0(i,n,s)
    & $tmfac0(j,n,s)":
```

Output is too crazy to show!

Fibonacci-synchronized functions

We can define synchronized functions for the Fibonacci numeration system in the same way we defined them for integer bases. Two useful Fibonacci-synchronized functions are

$$n \rightarrow \lfloor \varphi n \rfloor \text{ and } n \rightarrow \lfloor \varphi^2 n \rfloor,$$

where $\varphi = (1 + \sqrt{(5)})/2$.

We obtain automata for the pairs $(n, \lfloor \varphi n \rfloor)$ and $(n, \lfloor \varphi^2 n \rfloor)$ using the following facts: if $(n)_F$ is the Fibonacci-base representation of n, then

- $(n)_F0$ is the Fibonacci-base representation of $\lfloor \varphi(n+1) \rfloor - 1$, and
- $(n)_F00$ is the Fibonacci-base representation of $\lfloor \varphi^2(n+1) \rfloor - 2$.

Automata for $(n, \lfloor \varphi n \rfloor)$ and $(n, \lfloor \varphi^2 n \rfloor)$

Using these identities, we compute automata for the two functions with the Walnut commands

```
reg shift {0,1} {0,1} "([0,0]|[0,1][1,1]*[1,0])*":
def phin "?msd_fib (s=0 & n=0) |
   Ex $shift(n-1,x) & s=x+1":
def phi2n "?msd_fib (s=0 & n=0) |
   Ex,y $shift(n-1,x) & $shift(x,y) &  s=y+2":
```

Wythoff sequences

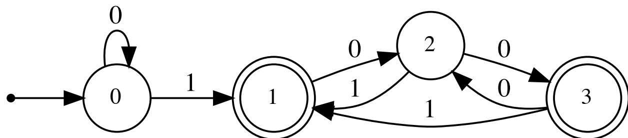
We can also obtain automata for the lower Wythoff set

$$L = \{\lfloor \varphi n \rfloor : n \geq 1\}$$
$$= \{1, 3, 4, 6, 8, 9, 11, 12, 14, 16, 17, 19, 21, \ldots\}$$

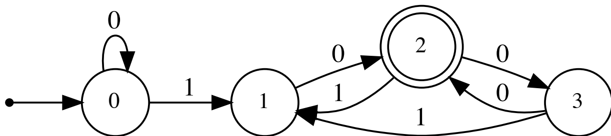
and the upper Wythoff set

$$U = \{\lfloor \varphi^2 n \rfloor : n \geq 1\}$$
$$= \{2, 5, 7, 10, 13, 15, 18, 20, 23, 26, 28, \ldots\}.$$

```
def lower "?msd_fib En n>=1 & $phin(n,s)":
def upper "?msd_fib En n>=1 & $phi2n(n,s)":
```



(s): ?msd_fib En n>=1 & $phin(n,s)



(s): ?msd_fib En n>=1 & $phi2n(n,s)

We can then easily prove the following recent result of Kawsumarng, Khemaratchatakumthorn, Noppakaew, and Pongsriiam (2021):

Theorem

Every non-negative integer, except $0, 1, 3$, can be written as a sum of two lower Wythoff numbers.

```
def lplusl "?msd_fib Ea,b n=a+b & $lower(a) &
   $lower(b)":
def lpluslcheck "?msd_fib An $lplusl(n) <=>
   (n>=4 | n=2)":
```

Related work

Pecan is another software package that also implements a similar decision procedure for logical predicates but extends what Walnut can do by including the theory of Sturmian words. It was implemented by Reed Oei, who, sadly, passed away in 2022 at the age of 23.

Reed Oei   Home Blog Publications CV

Pecan: An Automated Theorem Prover

Pecan is an automated theorem prover. You can view the manual (which is currently rather incomplete) here. Below, you can enter a program and click "Run" to try out Pecan; there are several example programs you can try out. There are some limitations: some features of Pecan are not available, your programs must be under 10^5 characters, and your programs will time out after 5 minutes. If that's too limiting, you should look into installing Pecan on your own computer. Instructions are available at the repository: https://github.com/ReedOei/Pecan. If you use Pecan in your research, please cite this paper

Enter a Pecan program below:

Basic Arithmetic | Chicken McNuggets | Plotting | Thue-Morse Word | Sturmian Words

```
#import("SturmianWords/ostrowski_defs.pn")
#load("SturmianWords/automata/antisquare.aut", "hoa", antisquare(a,i,n))
#load("SturmianWords/automata/eventually_periodic.aut", "hoa", eventually_periodic(a, p))
#load("SturmianWords/automata/palindrome.aut", "hoa", palindrome(a, i, n))

// See the GitHub repo for more: https://github.com/ReedOei/SturmianWords

Let a be bco_standard.
Let i,n,m,p be ostrowski(a).

Theorem ("Sturmian words are not eventually periodic", {
    forall a, p. !@no_simplify[eventually_periodic(a, p)]
}).

Theorem ("Sturmian words contain palindromes of every length.", {
    forall a, n. if n > 0 then exists i. palindrome(a, i, n)
}).

Theorem ("There are finitely many antisquares", {
    forall a. exists m. forall i,n. if antisquare(a,i,n) then n <= m
}).
```

Run Debug Mode

Output:

This website is written in Prolog

The End