Walnut: A Tool for Doing Combinatorics on Words

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

What is Walnut?

Walnut is a free software program written (in Java) by Hamoon Mousavi that, in many cases, can 'automatically'

- prove or disprove conjectures about automatic sequences;
- give simple, explicit characterizations of the factors of automatic sequences having certain properties;
- provide explicit formulas for counting aspects of automatic sequences.

What is Walnut?

In conjunction with other software it can be used to

- heuristically search for infinite sequences having certain properties; and then
- prove that the candidate you found really does have the property you want.

Restrictions:

- It is *not* a general-purpose tool. Its use is restricted to the small domain of automatic sequences only.
- The statements it works with must be phrased in first-order logic (an extension of Presburger arithmetic).

What can be done with Walnut

Walnut can

- Re-prove dozens of existing results published in the literature, often with trivial proofs
- Prove new results (about 35 papers published so far)
- Correct or strengthen already-published results.

See https://cs.uwaterloo.ca/~shallit/walnut.html for some of these papers and to download Walnut or the new Ostrowski version.

Example 1: the Thue-Morse sequence is overlap-free

One of the most famous automatic sequences is the *Thue-Morse sequence*

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

a fixed point of the morphism $\mu : 0 \to 01,\ 1 \to 10$.

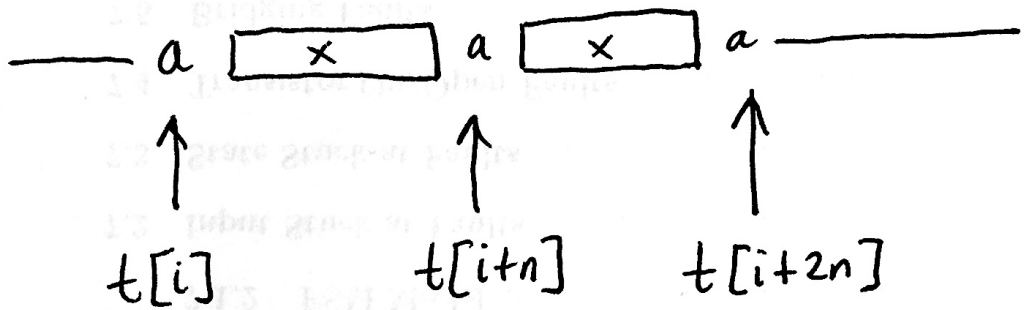
Let us see how to use `Walnut` to prove one of the oldest and most famous results in combinatorics on words: Thue's 1912 result that **t** is overlap-free.

An *overlap* is a word of the form *axaxa*, where *a* is a single letter and *x* is a (possibly empty) word, like the French word `entente`.

"Overlap-free" means **t** has no factor that is an overlap.

Example 1: the Thue-Morse sequence is overlap-free

If $\mathbf{t}$ has an overlap $axaxa$, then it must begin at some position i and we must have $|ax| = n$ for some $n \geq 1$:



So an overlap in $\mathbf{t}$ means there are i, n such that

$$(n \geq 1) \text{ and } \mathbf{t}[i..i+n] = \mathbf{t}[i+n..i+2n]$$

or in other words

$$\exists i, n\ (n \geq 1)\ \wedge\ \forall s\ (0 \leq s \leq n) \implies \mathbf{t}[i+s] = \mathbf{t}[i+s+n].$$

This can be translated into `Walnut` as follows:

```
eval hasolap "Ei,n (n>=1) & As (s<=n) => T[i+s]=T[i+s+n]":
```

and this returns FALSE.

Automatic sequences

`Walnut` can determine the truth of well-formed first-order formulas about *automatic* sequences $\mathbf{a} = (a_n)_{n \geq 0}$.

a is automatic if

- there is a regular numeration system S and a
- deterministic finite automaton with output (DFAO) M such that on input the S-representation of n, the DFAO M reaches a state with output a_n.

Examples include the Thue-Morse sequence, the Rudin-Shapiro sequence, the Fibonacci infinite word, the Tribonacci infinite word, etc.

Regular numeration systems

A *numeration system* is a way of representing natural numbers n by strings over a finite alphabet Σ.

A numeration system S is *regular* if

(a) Every natural number has exactly one representation (modulo leading zeros);

(b) The set of valid representations is a regular language; and

(c) The set of representations of triples (x, y, z) such that $x + y = z$ is recognized by a deterministic finite automaton (DFA).

(Shorter representations are padded with leading zeros, if necessary, so that (x, y, z) is read digit-by-digit in parallel.)

Examples of regular numeration systems

Examples of regular numeration systems:

- Base-k representation, $k \geq 2$ (built-in to `Walnut`)
- Fibonacci representation (built-in)
- Tribonacci representation (built-in)
- Ostrowski representation for quadratic irrationals (built-in to the Ostrowski version written by Baranwal)

In the case of base-k representation, k-automatic sequences coincide with images (under a coding) of the fixed points of k-uniform morphisms (Cobham).

The theory behind `Walnut`

Very little is original with me. The theory behind how it works is due to these people and more:

- J. Richard Büchi
- Bernard R. Hodgson
- Veronique Bruyère, Georges Hansel, Christian Michaux, Roger Villemaire
- Christiane Frougny
- Hamoon Mousavi
- Luke Schaeffer, Aseem Raj Baranwal

But I'm not going to go into the theory today.

What Walnut needs

(a) Choice of a regular numeration system (the default is base 2);

(b) A DFAO computing a sequence $\mathbf{s} = (s_n)_{n \geq 0}$ (a few, like the Thue-Morse sequence, are automatically supplied with Walnut—others you can define yourself);

(c) A first-order formula φ involving variables, quantifiers, logical operations, addition & subtraction of natural numbers, and indexing into $\mathbf{s}$.

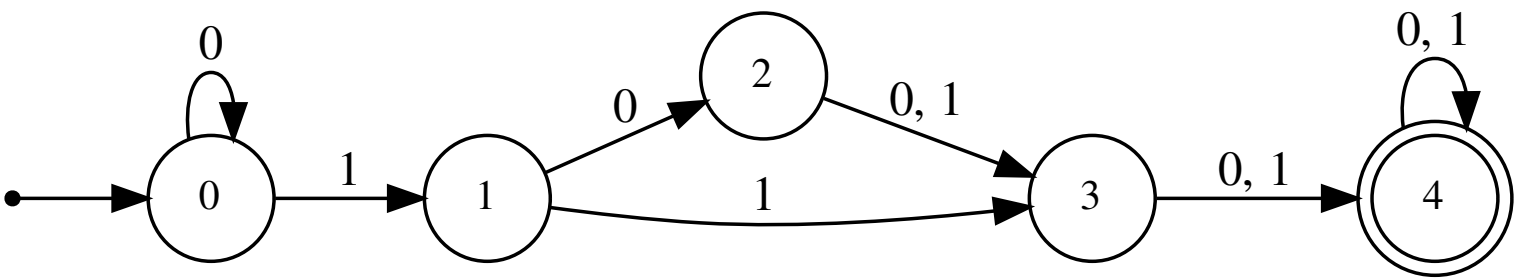
- No multiplication or division allowed (but can multiply or do integer division with a natural number constant; e.g., `2*x` is understood as `x+x`)
- Subtraction must not result in a negative number (may be relaxed in a future version of Walnut)
- No arithmetic with sequence values (only with indices), but can compare with $<$, $=$, etc.

What Walnut produces

If the logical formula φ has no free variables ('free' = not bound to a quantifier), Walnut produces either the answer TRUE or FALSE.

If the logical formula φ has one or more free variables, then Walnut produces a DFA recognizing the values of the free variables that make φ true.

For example, the formula $n \geq 6$ corresponds to the following automaton (in base-2 representation):



The domain of all variables is assumed to be $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Quick guide to Walnut syntax

- `A` means $\forall$, "for all"; `E` means $\exists$, "there exists"
- `&` means $\wedge$, "and"; `|` means $\vee$, "or"; `~` means $\neg$, "logical not"; `=>` means $\Longrightarrow$, "logical implication"; `<=>` means $\Longleftrightarrow$, "iff"
- Arithmetic operations are `+, -, *, /`
- `def`: defines a macro (automaton) that can be used later, with multiple arguments
- `eval`: evaluates a statement and returns TRUE/FALSE
- `reg`: defines a regular expression for matching representation of integers
- `?msd_3`: says to evaluate the formula using base-3 representation (can also say `?msd_fib`, `?lsd_2`, etc.)
- `@0` represents the sequence value 0,
  `@1` represents the value 1, etc.
- When calling multi-parameter macro, order of parameters is alphabetical in terms of original definition

Expressing bounded quantification

To say $\forall n \geq t\ p(n)$ in `Walnut`, say instead

```
An (n>=t) => $p(n)
```

To say $\exists n \geq t\ p(n)$ in `Walnut`, say instead

```
En (n>=t) & $p(n)
```

To say $p(n)$ holds for infinitely many n in `Walnut`, say instead

```
Am En (n>=m) & $p(n)
```

Example 2: analyzing antipowers with `Walnut`

An r-antipower in a word consists of r consecutive **distinct** blocks, all of the same length (Fici-Restivo-Silva-Zamboni, 2018).

For example, `entanglement` is a 3-antipower, but not a 4-antipower:

$$\texttt{enta} \cdot \texttt{ngle} \cdot \texttt{ment}$$

$$\texttt{ent} \cdot \texttt{ang} \cdot \texttt{lem} \cdot \texttt{ent}$$

The Cantor word **ca** $= 101000101\cdots$ is the fixed point of the morphism $1 \to 101$, $0 \to 000$ starting with 1.

Fici, Postic, and Silva (2019) proved that **ca** avoids 11-antipowers. We can improve this result optimally to:

Theorem. (Riasat) The Cantor word **ca** contains no 10-antipowers, but does contain a 9-antipower.

Proof of the result

Proof. It is easy to verify that

$$\mathbf{ca}[157..246] = (0000010100)(0101000000)(0001010001)(0100000000)(0000000000)$$
$$(0000000001)(0100010100)(0000000101)(0001010000)$$

is a 9-antipower.

To show that **ca** has no 10-antipowers, it suffices to show that every block of size $10n$ in it can be split into 10 consecutive blocks of size n, with at least two blocks being identical.

At first glance, proving this would seem to require comparison of $10 \cdot 9/2 = 45$ different pairs of blocks: there are no 10-antipowers provided

$$\forall i, n\,(n \geq 1) \implies \exists k, l\,(0 \leq k < \ell < 10) \wedge \text{factoreq}(i+kn, i+ln, n).$$

But this uses a disallowed operation (multiplication).

Finishing the proof

However, we can make it expressible by "unrolling" the $0 \leq k < \ell < 10$ part to make it 45 individual statements with multiplication by constants.

In fact, for the Cantor word **ca** we can get by with comparison of only 14 of the 45 blocks, as follows:

```
def cfactoreq "?msd_3 At (t<n) => CA[i+t]=CA[j+t]":

eval cno10 "?msd_3 Ai,n (n>=1) => (
     $cfactoreq(i+0*n,i+1*n,n) | $cfactoreq(i+0*n,i+5*n,n) |
     $cfactoreq(i+2*n,i+3*n,n) | $cfactoreq(i+2*n,i+8*n,n) |
     $cfactoreq(i+3*n,i+4*n,n) | $cfactoreq(i+3*n,i+7*n,n) |
     $cfactoreq(i+3*n,i+9*n,n) | $cfactoreq(i+4*n,i+5*n,n) |
     $cfactoreq(i+4*n,i+9*n,n) | $cfactoreq(i+5*n,i+6*n,n) |
     $cfactoreq(i+5*n,i+8*n,n) | $cfactoreq(i+6*n,i+7*n,n) |
     $cfactoreq(i+7*n,i+8*n,n) | $cfactoreq(i+8*n,i+9*n,n))":
```

which evaluates to TRUE.

Example 3: unbordered factors of Thue-Morse

Recall that a word w is *bordered* if it begins and ends with a nonempty word different from w.

For example, `meantime` is bordered with border `me`.

Currie and Saari (2009) studied the *unbordered* factors of the Thue-Morse sequence $\mathbf{t}$.

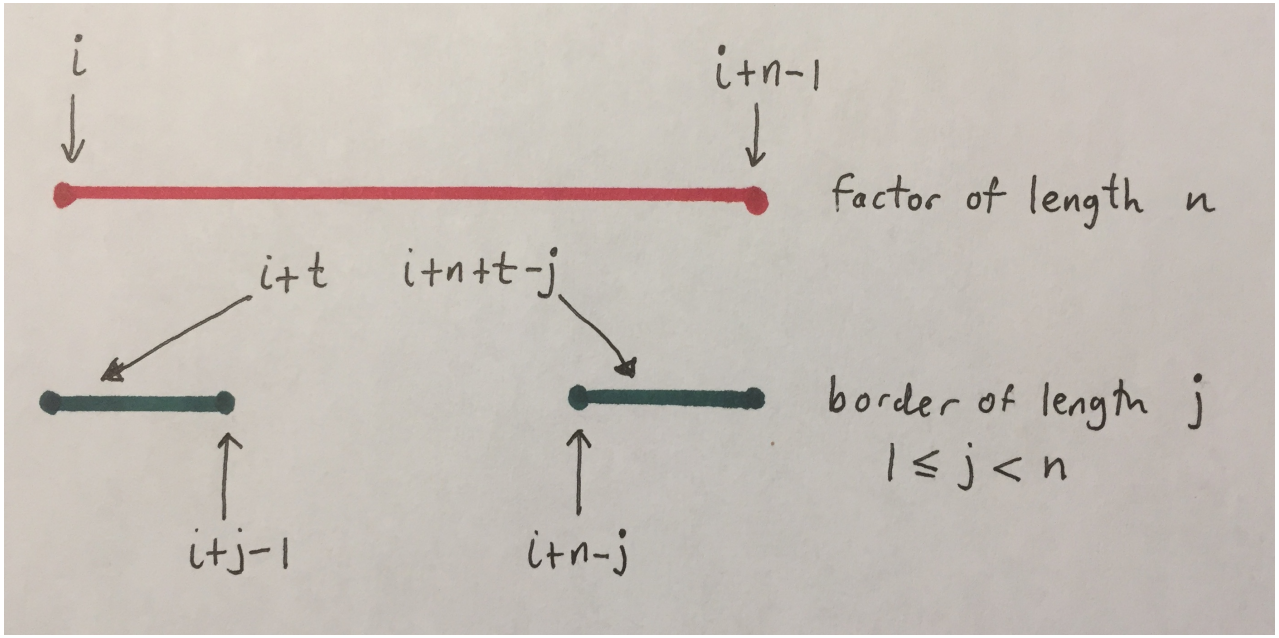
They proved the implication: if $n \not\equiv 1 \pmod 6$, then $\mathbf{t}$ has an unbordered factor of length n.

But $\mathbf{t}$ also has an unbordered factor of length 31, so their criterion is sufficient, but not necessary.

Also their proof was rather long and case-based.

Unbordered factors of Thue-Morse

We can get a full characterization of the lengths n for which there's an unbordered factor by writing a formula for a factor $\mathbf{t}[i..i+n-1]$ to be unbordered.

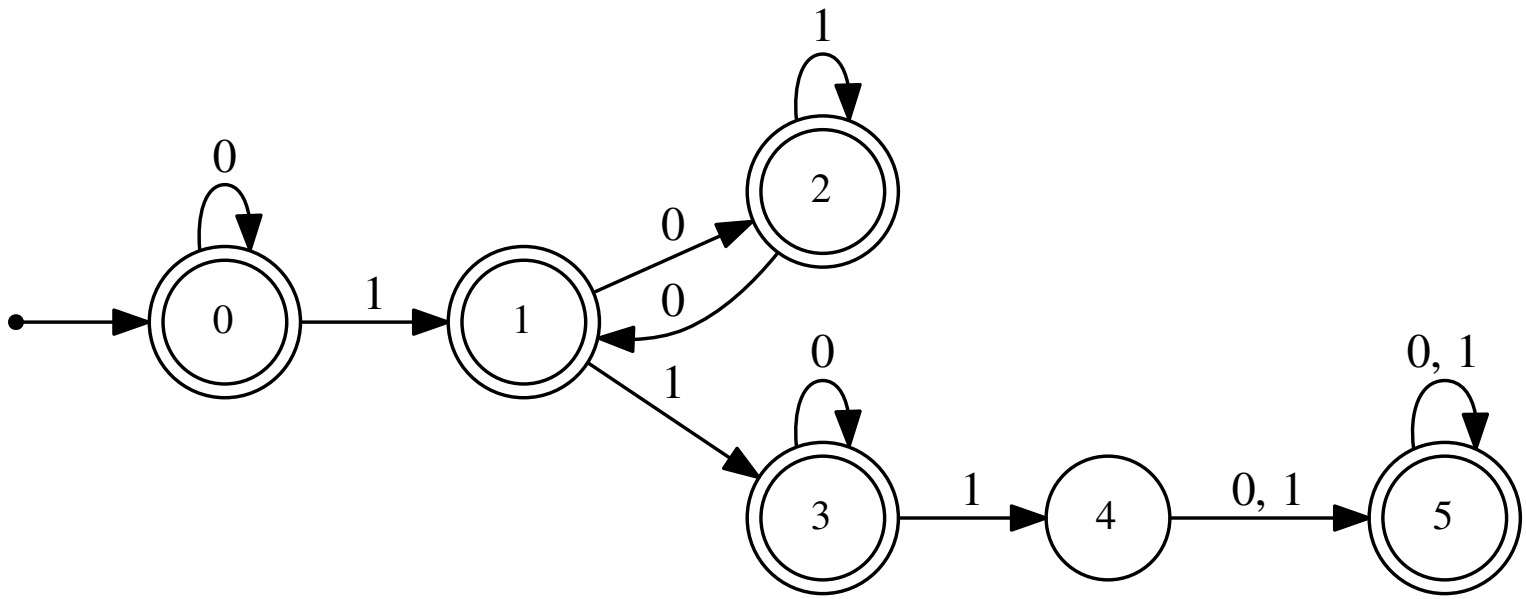


In `Walnut` this is (n is unbound variable):

```
def tmhasbord "Ej (j>=1) & (j<n) & At (t<j) =>
T[i+t]=T[(i+n+t)-j]": # T[i..i+n-1] is bordered
def unbordlen "Ei ~$tmhasbord(i,n)":
```

Unbordered factors of Thue-Morse

`Walnut` computes an automaton with 6 states recognizing exactly these n:



By considering the paths that go from state 0 to state 4, we get:

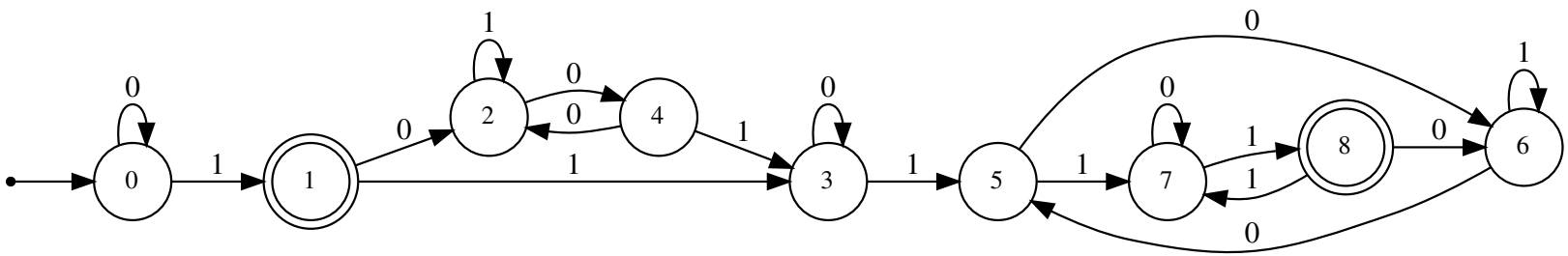
Theorem. **t** has an unbordered factor of length n iff $(n)_2$ is *not* of the form $1(01^*0)^*10^*1$.

Unbordered factors of Thue-Morse

Which n did the Currie-Saari characterization miss?

```
def missing "$unbordlen(n) & n=1+6*(n/6)":
```

This gives the following DFA:



So there are infinitely many (for example $2^{2k+1} - 1$ for $k \geq 2$).

Example 4: Automated heuristic search for sequences with certain properties

Example: construct an infinite binary sequence with only three distinct squares. (Original result due to Fraenkel and Simpson).

Guess: an example exists that is k-automatic.

Algorithm: for all k, s with $k \geq 2$ and $ks \leq B$, do breadth-first search on the space of all finite binary sequences having at most three distinct squares, and recognizable by an automaton in base k with at most s states (use Myhill-Nerode theorem).

For each automaton determined, compute $100, 200, 400, 800$, etc. terms from the automaton and see if the result still has at most three squares.

If these tests are passed, we have a candidate that can be checked with `Walnut`. (There is a first-order formula for "having at most three squares".)

Result for three squares

Theorem. (Gabric and Shallit, 2020). The following 2-uniform morphism q on 22 letters, and coding γ are such that $\gamma(q^\omega(0))$ has only three squares.

$$
\begin{array}{llll}
a, p \to ab & b \to cd & c \to ef & d, q \to gh \\
e, h \to ij & f, r \to kl & g \to mh & i, u \to no \\
j \to pb & k, n \to cq & l \to hr & m \to ge \\
o, v \to st & s \to uj & t \to kv &
\end{array}
$$

$$\gamma(abcdefghijklmnopqrstuv) = 11010011000111110010110.$$

This is the "simplest" infinite binary word containing at most three distinct squares.

Example 5: Dean words

A recent preprint of Harju discusses *Dean words*: squarefree words over $\{x, y, x^{-1}, y^{-1}\}$ that are not reducible (no occurrences of $xx^{-1}, x^{-1}x, yy^{-1}, y^{-1}y$).

Use the coding $0 \leftrightarrow x$, $1 \leftrightarrow y$, $2 \leftrightarrow x^{-1}$, $3 \leftrightarrow y^{-1}$.

Again, we can use breadth-first search to look for a candidate automatic sequence.

It quickly converges on the sequence

$$0121032101230321\cdots,$$

the fixed point of the morphism

$$0 \to 01,\ 1 \to 21,\ 2 \to 03,\ 3 \to 23.$$

Example 5: Dean words

We make a DFAO and store it under the name `DE.txt` in the `Word Automata` library of `Walnut`.

Then we carry out the following commands:

```
eval dean1 "Ei,n (n>=1) & At (t<n) => DE[i+t]=DE[i+n+t]":
# check if there's a square

eval dean02 "Ei DE[i]=@0 & DE[i+1]=@2":
eval dean20 "Ei DE[i]=@2 & DE[i+1]=@0":
eval dean13 "Ei DE[i]=@1 & DE[i+1]=@3":
eval dean31 "Ei DE[i]=@3 & DE[i+1]=@1":
# check for existence of factors 02, 20, 13, 31
```

All of these return `FALSE`, so this word is a Dean word.

Example 6: Comparing two different automatic sequences

Recall the automatic sequence $0121032101230321\cdots$ from the previous example.

If we look at the even-indexed terms, they look like $020202\cdots = (02)^\omega$.

If we look at the odd-indexed terms, they look like $1131133\cdots$, which looks a lot like the regular paperfolding sequence.

We can check both of these claims as follows:

```
eval checkde1 "An DE[4*n]=@0 & DE[4*n+2]=@2":
eval checkde2 "An (DE[2*n+1]=@1 => P[n+1]=@0) &
     (DE[2*n+1]=@3 => P[n+1]=@1)":
```

Both of these return TRUE.

Example 7: Quasiperiodicity

We say an infinite word $\mathbf{x}$ is *quasiperiodic* if there exists a finite prefix of $\mathbf{x}$ (call it z) that covers $\mathbf{x}$ by shifts (allowing overlaps). Such a z is called a *quasiperiod*.

For example, `aba` covers `abaababa` by shifts.

We can write a first-order logical formula for quasiperiodicity of the infinite word $\mathbf{x}$ as follows:

$$(n > 0) \wedge \forall i \, \exists j \, (j \leq i) \wedge (i < n+j) \wedge (\forall \ell \, (\ell < n) \implies \mathbf{x}[\ell] = \mathbf{x}[j+\ell]).$$

Let us use `Walnut` to prove the following result of Christou, Crochemore, and Iliopoulos (2016):

Theorem. A length-n prefix of the Fibonacci word $\mathbf{f}$ (fixed point of $0 \to 01$, $1 \to 0$) is a quasiperiod of $\mathbf{f}$ if and only if n is not of the form $F_k - 1$ for $k \geq 1$.

Quasiperiodicity

To do so we write Walnut code as follows:

```
def fibfactoreq "?msd_fib At (t<n) => F[i+t]=F[j+t]":
def fibquasi "?msd_fib Ai Ej j<=i & i<n+j &
     $fibfactoreq(0,j,n)":  # F quasiperiodic with per n
reg isfib msd_fib "0*10*":
eval fibquasichk "?msd_fib An $fibquasi(n) <=> ~$isfib(n+1)":
```

In contrast it is easy to prove that the Thue-Morse word is not quasiperiodic.

```
def tmfactoreq "At (t<n) => T[i+t]=T[j+t]":
def tmquasi "Ai (Ej j<=i & i<n+j & $tmfactoreq(0,j,n))":
eval tmquasicheck "An ~$tmquasi(n)":
```

Both of these return TRUE.

Quasiperiodicity

However, the Thue-Morse word *can* be covered by a *pair* of words: it is *2-quasiperiodic*.

We can use `Walnut` to prove the following new result:

Theorem. The word **t** can be covered by the pairs

- $\mu^n(0), \mu^n(1)$ for $n \geq 0$, and
- $\mu^n(010), \mu^n(0110)$ for $n \geq 0$,

and no other pairs of words.

Example 8: Balanced words

Let $|x|_a$ denote the number of occurrences of the letter a in x.

We say a word x is *balanced* if $||y|_a - |z|_a| \leq 1$ for all equal-length factors y, z of x and all letters a.

For example, the word `banana` is balanced, but `apple` is unbalanced.

A priori there is no obvious way to state the balanced definition in first-order logic, since it seems to require counting a finite subset having a certain property.

However, there is an alternate characterization of balance for binary words: a (finite or infinite) word x is balanced if it contains no pairs of factors of the form $0v0$ and $1v1$.

Balanced words

We can create a formula asserting that $\mathbf{x}[i..i+n-1]$ is unbalanced, as follows: there exist indices ℓ_1, r_1, ℓ_2, r_2 such that $(\mathbf{x}[\ell_1..r_1] = 0v0)$ and $(\mathbf{x}[\ell_2..r_2] = 1v1)$. That is,

- $i \le \ell_1 < r_1 < i+n, \quad i \le \ell_2 < r_2 < i+n$
- $r_1 - \ell_1 = r_2 - \ell_2$
- $\mathbf{x}[\ell_1] = \mathbf{x}[r_1] = 0, \quad \mathbf{x}[\ell_2] = \mathbf{x}[r_2] = 1$
- $\mathbf{x}[\ell_1+1..r_1-1] = \mathbf{x}[\ell_2+1..r_2-1]$

For the Fibonacci word **f** this is

```
def unbalfib "?msd_fib E l1,l2,r1,r2 i<=l1 & l1<r1 & r1<i+n
& i<=l2 & l2<r2 & r2<i+n & r1+l2=r2+l1 & F[l1]=@0 & F[r1]=@0
& F[l2]=@1 & F[r2]=@1 & $fibfactoreq(l1+1,l2+1,r1-(l1+1))":
     # F[i..i+n-1] is unbalanced
eval balfib "?msd_fib Ai An ~$unbalfib(i,n)":
```

This returns TRUE, so every factor of **f** is balanced.

Balanced words

For Thue-Morse we can write

```
def unbaltm "E l1,l2,r1,r2 i<=l1 & l1<r1 & r1<i+n & i<=l2
& l2<r2 & r2<i+n & r1+l2=r2+l1 & T[l1]=@0 & T[r1]=@0 &
T[l2]=@1 & T[r2]=@1 & $tmfactoreq(l1+1,l2+1,r1-(l1+1))":
    # T[i..i+n-1] is unbalanced

eval baltm "Ei ~$unbaltm(i,n)":
```

Inspecting the result shows that **t** has balanced factors of length ≤ 8 only.

Example 9: Linear recurrence

Recall that a sequence $\mathbf{x}$ is *linearly recurrent* if there is a function $g(n) = O(n)$ such that every occurrence of a factor of length n in $\mathbf{x}$ is followed by another occurrence of the same factor at distance at most $g(n)$. (Distance = difference in indices of starting points of factor.)

In other words:

$$\exists c\ \forall i, n\ (n \geq 1) \implies \exists j\ (j \geq 1\ \wedge\ j \leq cn)\ \wedge\ \text{factoreq}(i, i+j, n).$$

As stated, this is not expressible (because of the multiplication cn).

However, if we have a good guess for what g might be, we can easily verify it (and verify it is best possible).

Example 9: Linear recurrence

Theorem. For the Thue-Morse sequence **t** we may take $g(n) = 9n - 18$ for $n \geq 3$, and this is optimal.

For the proof, we use `Walnut`:

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

eval tmlinrec "Ai, n (n>=3) => Ej (j>=1 & j+18<=9*n) &
     $tmfactoreq(i,i+j,n)": # for all i, n the next
     occurrence of T[i..i+n-1] is at distance <= 9n-18

eval tmopt "Am Ei,n (n>=m) & Aj
     (j>=1 & $tmfactoreq(i,i+j,n)) => j+18>=9*n":
     # there are arbitrarily large factors where next
     # occurrence is at distance >= 9n-18.
```

Example 10: Additive number theory

Let's do some number theory with `Walnut`!

Let $S, T \subseteq \mathbb{N}$ be subsets of natural numbers. Define the *sumset*

$$S + T = \{s + t : s \in S, t \in T\}.$$

Let $\varphi = (1 + \sqrt{5})/2$ be the golden ratio. The lower Wythoff sequence is

$$L = (\lfloor \varphi n \rfloor)_{n \geq 1} = (1, 3, 4, 6, 8, 9, \ldots)$$

and the upper Wythoff sequence is

$$U = (\lfloor \varphi^2 n \rfloor)_{n \geq 1} = (2, 5, 7, 10, 13, 15, \ldots).$$

Recently Kawsumarng et al. looked at sumsets of the lower and upper Wythoff sequences.

Additive number theory

They proved theorems like

$$L + U + U = \mathbb{N} - \{0, 1, 2, 3, 4, 6, 9\}.$$

But their proofs were long, case-based, and complicated.
We can re-prove their results with `Walnut`, using Fibonacci representation and a theorem of Silber:

$$n \in L \Longleftrightarrow (n-1)_F \text{ ends in } 0;$$
$$n \in U \Longleftrightarrow (n-1)_F \text{ ends in } 1.$$

We can implement this in `Walnut` as follows:

```
reg end0 msd_fib "(0|1)*0":
reg end1 msd_fib "(0|1)*1":
def lower "?msd_fib $end0(n-1)":
def upper "?msd_fib $end1(n-1)":
```

Additive number theory

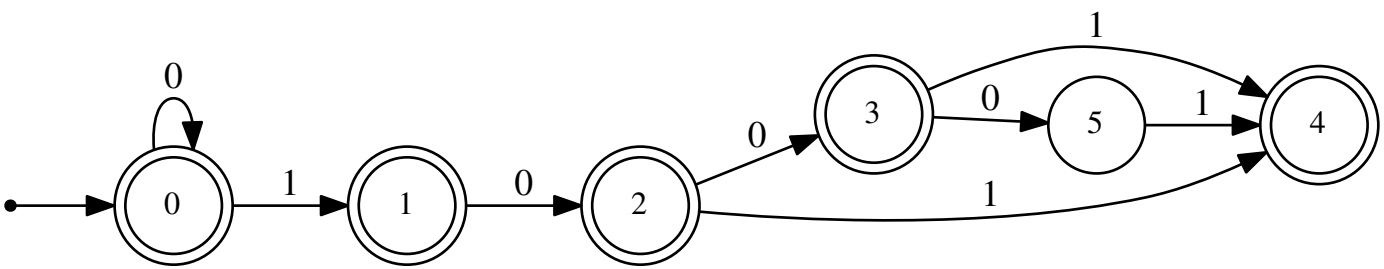
And we can prove their theorem by just evaluating

```
eval kaw "?msd_fib ~Ea,b,c (n=a+b+c) & $lower(a) &
      $upper(b) & $upper(c)":
```

which gives the Fibonacci representation of all numbers **not** in the sumset $L + U + U$.

The result is provided as an automaton recognizing the Fibonacci representation of those $n \notin L + U + U$.

Here's the automaton:



By inspection we see this DFA recognizes the Fibonacci representations of $\{0, 1, 2, 3, 4, 6, 9\}$.

Using `Walnut` for enumeration

`Walnut` can also be used to find a representation for various kinds of *integer-valued functions* of automatic sequences.

The simplest is a so-called *linear representation* for a function $f(n)$. It consists of a row vector u, a column vector v, and a matrix-valued morphism $\gamma : \Sigma^* \to \mathbb{N}$ such that

$$f(n) = u \cdot \gamma(x) \cdot v$$

where x is the representation of n.

Let us do this for *subword complexity* of the Thue-Morse sequence (previously computed by Brlek, de Luca & Varricchio, and Avgustinovich).
Here $f(n) =$ number of distinct length-n factors of $\mathbf{t}$.

Subword complexity of the Thue-Morse sequence I

To compute f, we count the number of length-n *novel* factors: those beginning at some position i, but not occurring at any earlier position. We can do this with the following `Walnut` commands:

```
def tmfactoreq "At (t<n) => T[i+t]=T[j+t]":
eval tmsw n "Aj (j<i) => ~$tmfactoreq(i,j,n)":
     # T[i..i+n-1] is a novel factor
```

The output is a `Maple` file with the definitions of $u, \gamma(0), \gamma(1), v$.

For technical reasons I won't go in to here, one needs to replace u by $u' := u\gamma(0)$ until the value stabilizes. This gives a linear representation for $f(n)$, of rank 8.

The linear representation

We have

$$\begin{aligned} f(n) &= \text{the number of distinct length-}n\text{ factors of } \mathbf{t} \\ &= u' \cdot \gamma(\,(n)_2\,) \cdot v, \end{aligned}$$

where

$$u' = \begin{bmatrix} 1\\1\\0\\0\\1\\0\\0\\0 \end{bmatrix}^T; \quad \gamma(0) = \begin{bmatrix} 11000000\\00001000\\00200000\\00100100\\00000000\\00100100\\00000010\\00000020 \end{bmatrix}; \quad \gamma(1) = \begin{bmatrix} 00110000\\00000110\\00200000\\00200000\\00000011\\00200000\\00000110\\00000200 \end{bmatrix}; \quad v = \begin{bmatrix} 1\\0\\1\\1\\0\\0\\0\\0 \end{bmatrix}$$

This gives

- An explicit formula for $f(n)$ that we can evaluate in $O((\log n)^3)$ time
- An exact formula for $f(2^n)$ and similar subsequences in terms of the eigenvalues of $\gamma(0)$, etc.

Getting an exact formula for some cases

We have

$$f(2^n) = u' \cdot \gamma(1) \cdot \gamma(0)^n \cdot v.$$

Each entry of $\gamma(0)^n$ satisfies a linear recurrence corresponding to the minimal polynomial of $\gamma(0)$, which is $X^2(X-1)(X-2)$.

Since $f(2^n)$ is a linear combination of these entries, it also satisfies the same linear recurrence.

By the fundamental theorem of linear recurrences, we have $f(2^n) = a \cdot 2^n + b$ for some constants a, b and $n \geq 2$.

We can now solve for a and b by substituting two values for n.

This gives $a = 3$, $b = -2$, so $f(2^n) = 3 \cdot 2^n - 2$ for $n \geq 2$.

Subword complexity II: Via right special factors

We know from a theorem of Cassaigne that the first difference of the subword complexity function of an automatic sequence is bounded above by a constant.

It is easy to see that $f(n+1) - f(n)$ is the number of length-n factors x that are *right special*, that is, both $x0$ and $x1$ occur in $\mathbf{t}$.

So we can count these as follows in `Walnut`:

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

This gives us a linear representation of rank 6 for $f(n+1) - f(n)$:

$$u = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}^T ; \quad \gamma(0) = \begin{bmatrix} 110000 \\ 000000 \\ 000011 \\ 001010 \\ 000010 \\ 000001 \end{bmatrix} ; \quad \gamma(1) = \begin{bmatrix} 001100 \\ 000010 \\ 000011 \\ 000002 \\ 000011 \\ 000000 \end{bmatrix} ; \quad v = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Subword complexity II: Via right special factors

We can now minimize this using an algorithm of Schützenberger (see the book of Berstel and Reutenauer) to get the following linear representation:

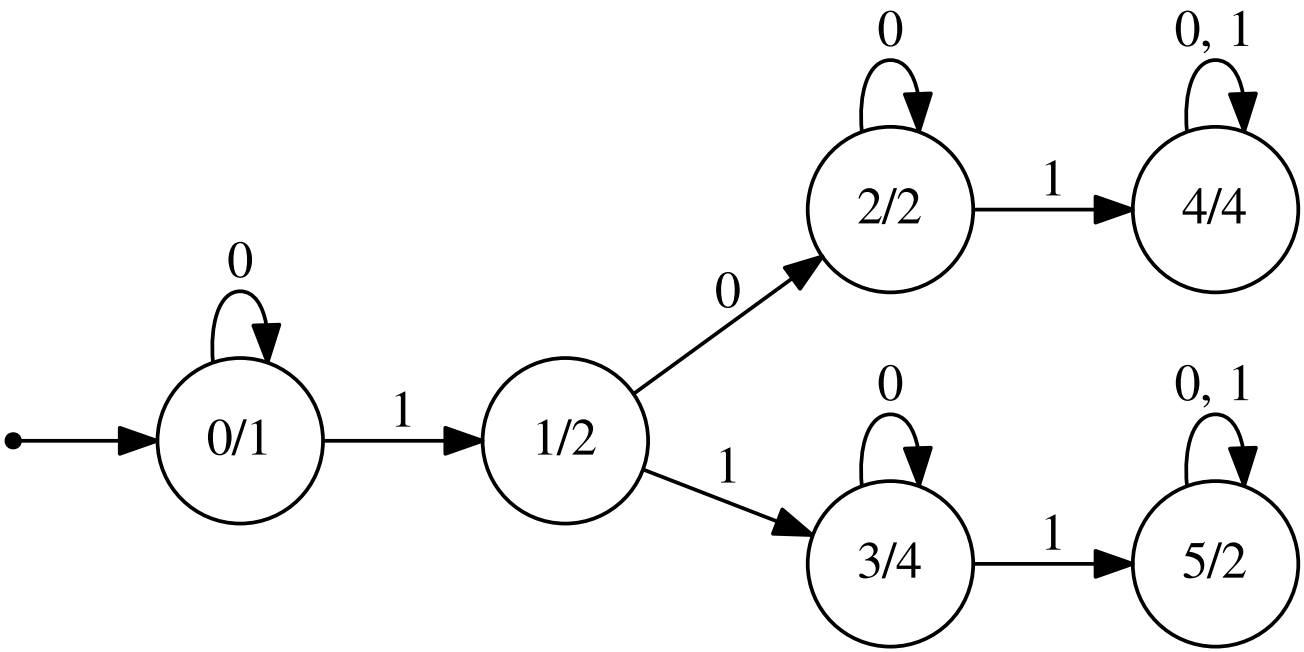
$$u = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}^T; \quad \gamma(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad \gamma(1) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{2}{3} & \frac{2}{3} \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} \end{bmatrix}; \quad v = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 4 \end{bmatrix}$$

We can now turn this into a DFAO as follows:

- states are 4-element vectors
- the initial state is u
- transition function $\delta(s, a) := s \cdot \gamma(a)$
- output function of state s is $s \cdot v$

Computing the DFAO

This gives the following DFAO computing the first difference of the subword complexity function for Thue-Morse:



For example, the right-special factors of length 5 are $00110, 01001, 10110, 11001$.

Subword complexity III: Via synchronized sequences

As we've seen, subword complexity is the number of length-n novel factors.

I observed and Luke Schaeffer proved that for automatic sequences, the starting positions of length-n novel factors lie in a finite number of *clumps* (consecutive positions).

So we can compute subword complexity by adding up the clump lengths.

We just "guess" (with $\exists$) the left and right endpoints of the clumps, and verify that

- inside a clump, all the length-n factors beginning there are novel, and
- outside the guessed clumps, the factors are not novel.

Subword complexity III: Via synchronized sequences

For Thue-Morse there are at most 5 clumps for each n.

This gives the following `Walnut` code:

```
def nf "Aj (j<i) => Et (t<n) & T[i+t] != T[j+t]": # novel factor
def allnovel "Ai (x<=i & i<y) => $nf(i,z)":
def allnotnov "Ai (x<=i & i<y) => ~$nf(i,z)":

def tmsub "E a2,a3,a4,a5,b1,b2,b3,b4,b5 (b1<=a2 &
a2<=b2 & b2<=a3 & a3<=b3 & b3<=a4 & a4<=b4 & b4<=a5 & a5<=b5)
& $allnovel(0,b1,n) & $allnovel(a2,b2,n) & $allnovel(a3,b3,n) &
$allnovel(a4,b4,n) & $allnovel(a5,b5,n) & $allnotnov(b1,a2,n) &
$allnotnov(b2,a3,n) & $allnotnov(b3,a4,n) & $allnotnov(b4,a5,n) &
(Ai (i>=b5) => ~$nf(i,n)) & s = b1+(b2-a2)+(b3-a3)+(b4-a4)+(b5-a5)":
```

This gives us a 14-state "synchronized" automaton recognizing, in parallel, the base-2 representations of n and $s = f(n)$.

Subword complexity via synchronized sequences

With such an automaton, one can easily use `Walnut` to verify the following formula for the subword complexity of Thue-Morse:

$$\rho_{\mathbf{t}}(n) = \begin{cases} 3 \cdot 2^r + 4(i-1), & \text{if } n = 2^r + i,\ 1 \le i \le 2^{r-1}; \\ 5 \cdot 2^r + 2(i-1), & \text{if } n = 3 \cdot 2^{r-1} + i,\ 1 \le i \le 2^{r-1}. \end{cases}$$

```
reg power2 msd_2 "0*10*":
eval tmsubcheck "Ax,i,s,n (((n>=3 & $power2(x) & n=x+i & i>=1
& 2*i<=x & s+4=3*x+4*i) => $tmsub(n,s)) & ((n>=3 & $power2(x) &
2*n=3*x+2*i & i>=1 & 2*i<=x & s+2=5*x+2*i) => $tmsub(n,s)))":
```

In general, synchronized representations for functions are the most useful, because we can use them to prove guessed exact formulas like the one above.

Unfortunately it's not always possible to find a synchronized representation for a function given by a linear representation.

Paperfolding sequences

The paperfolding sequences are an uncountable set of infinite sequences F, each one determined by a specific infinite sequence of unfolding instructions $\mathbf{u} = u_0, u_1, \ldots \in \{-1, +1\}$, as follows:

$$P_0 = \epsilon$$
$$P_{i+1} = P_i,\ u_i,\ -P_i^R.$$

and $P = \lim_{i \to \infty} P_i$.

There is a single finite automaton that specifies all these sequences: it takes as input n in base 2 *in parallel* with the unfolding instructions, and returns the ith bit of the appropriate paperfolding sequence.

Paperfolding sequences

So we can prove assertions about some or all paperfolding sequences using this one automaton and `Walnut`, such as:

Theorem. (Allouche & Bousquet-Mélou) No paperfolding sequence contains a repetition larger than a cube (i.e., a 3^+-power).

```
eval pfhaslcube "?lsd_2 En (n>0) & Ef Ei i>=1 &
Ak (k<=2*n) => PF[f][i+k] = PF[f][i+k+n]":
```

which returns FALSE.

Alternate formulations of queries can save time

Consider comparing two factors of the Tribonacci word:
TR$[i..i+n-1]$ and **TR**$[j..j+n-1]$:

```
def tribfaceq "?msd_trib At (t<n) => TR[i+t]=TR[j+t]":
 # Does TR[i..i+n-1]=TR[j..j+n-1] ?
```

This didn't run to completion, even after allocating 120 gigs of storage (by invoking `java -Xmx120000M Main.Prover`).

But a simple reformulation runs very quickly.

```
def tribfaceq2 "?msd_trib Ax Ay (x>=i & x<i+n & x+j=y+i)
=> TR[x]=TR[y]":  # Does TR[i..i+n-1]=TR[j..j+n-1] ?
```

This takes only 35 secs of CPU time and uses 7.2 gigs of storage to produce the required 26-state automaton.

Limitations of Walnut

- Only works with automatic sequences, not with arbitrary morphic sequences
  - Cannot be remedied in general, because the characteristic sequence of the squares is morphic, and $\mathrm{FO}(\mathbb{N}, +, x \to x^2)$ is not decidable (Tarski)
- Some queries might require ridiculously large amounts of time and space
  - Cannot be remedied in general, because there is a double exponential lower bound just for Presburger arithmetic
  - In practice, most queries run to completion, or can be reformulated to do so
- Can work with two or more sequences at once, but only if they are defined over the same numeration system (e.g., Thue-Morse and Rudin-Shapiro, but not Thue-Morse and Fibonacci)

Things `Walnut` can't handle because they're not expressible

- Arbitrary words (can only handle factors of automatic sequences and simple variations on them)
- Arbitrary-length (scattered) subsequences of automatic sequences (fixed-length ok)
- Abelian properties of factors (except in special cases, like the Thue-Morse sequence and Fibonacci word)
- Comparing the number of 1's in two different factors (except in special cases)
- Adding up the elements of a factor (except in special cases)
- Checking if terms of one sequence are arbitrary linear combinations of another (except if coefficients and number of terms are bounded)

Common `Walnut` pitfalls for beginners

- forgetting to specify the numeration system (if different from base 2)
- forgetting about edge conditions (empty factors, etc.)
  - Example: all conjugates of $\mathbf{f}[i..i+n-1]$ appear in $\mathbf{f}$. Wrong:

```
# asserts that F[j..j+n-1] is F[i..i+n-1], shifted by t
def fibshift "?msd_fib $fibfactoreq(j,i+t,n-t) &
     $fib(i,(j+n)-t,t)":

# asserts that F[j..j+n-1] is a conjugate of F[i..i+n-1]
def fibconj "?msd_fib Et (t<n) & $fibshift(i,j,n,t)":
```

    It fails for the empty string (so need `t<=n` instead).
- using parameters in the wrong order
- subexpression with subtraction resulting in negative number
- self-defined DFAO's must be placed in the `Word Automata` library and name must consist of capital letters, not starting with A or E

Conclusions

- First-order properties of automatic sequences are now "trivially decidable" (usually)
- Some properties (like balance for binary sequences) at first glance don't seem to be expressible, but can be reformulated to satisfy this requirement
- Conjectures about enumeration can similarly be "easily verified" if they involve synchronized automatic sequences
- You can replace long case-based arguments, prone to error, with a simple computation
- Rephrasing properties in first-order logic helps you make your definitions precise (handle edge cases like empty word correctly, etc.)
- Your mind is now free to work on harder properties, such as abelian and additive properties of sequences!

For further research

- Is there a first-order formula for the balanced property for words over a larger alphabet than size two? Or for generalizations of balance?
- Can one produce "automatic reformulation" of queries to make them more efficient? (compare `tribfaceq` discussion previously)
- implement a multicore version of `Walnut`

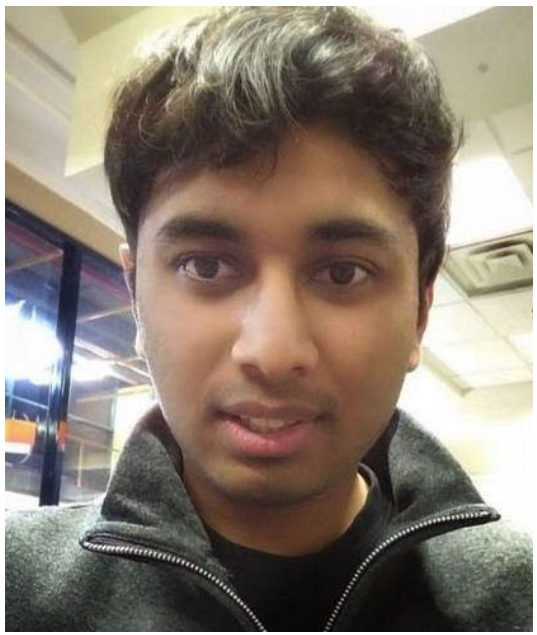
Walnut

To download Walnut, visit

https://cs.uwaterloo.ca/~shallit/walnut.html

Thanks to Hamoon Mousavi and Aseem Baranwal for their wonderful software!

Hamoon Mousavi



Aseem Baranwal