

The Tortoise and the Hare Algorithm

Peter Gammie

May 26, 2024

Abstract

We formalize the [Tortoise and Hare cycle-finding algorithm](#) ascribed to Floyd by [Knuth \(1981, p7, exercise 6\)](#), and an improved version due to [Brent \(1980\)](#).

Contents

1	Introduction	1
2	Point-free notation	2
3	“Monoidal” Hoare logic	3
4	Properties of iterated functions on finite sets	3
5	The Tortoise and the Hare	5
5.1	Finding <i>nu</i>	5
5.1.1	Side observations	6
5.2	Finding <i>mu</i>	6
5.3	Finding <i>lambda</i>	7
5.4	Top level	7
6	Brent’s algorithm	7
6.1	Finding <i>lambda</i>	8
6.2	Finding <i>mu</i>	9
6.3	Top level	9
7	Concluding remarks	9
	References	10

1 Introduction

[Knuth \(1981, p7, exercise 6\)](#) frames the problem like so: given a finite set X , an initial value $x_0 \in X$, and a function $f : X \rightarrow X$, define the infinite sequence x by recursion: $x_{i+1} = f(x_i)$. Show that the sequence is ultimately periodic, i.e., that there exist λ and μ where

$$x_0, x_1, \dots, x_\mu, \dots, x_{\mu+\lambda-1}$$

are distinct, but $x_{n+\lambda} = x_n$ when $n \geq \mu$.

Secondly (and he ascribes this to Robert W. Floyd), show that there is an $\nu > 0$ such that $x_\nu = x_{2\nu}$.

These facts are supposed to yield the insight required to develop the Tortoise and Hare algorithm, which calculates λ and μ for any f and x_0 using only $O(\lambda + \mu)$ steps and a bounded number of memory locations. We fill in the details in §5.

We also show the correctness of [Brent \(1980\)](#)’s algorithm in §6, which satisfies the same resource bounds and is more efficient in practice.

These algorithms have been used to analyze random number generators (Knuth 1981, op. cit.) and factor large numbers (Brent 1980). See Nivasch (2004) for further discussion, and an algorithm that is not constant-space but is more efficient in some situations. Wang and Zhang (2012) also survey these algorithms and present a new one.

2 Point-free notation

We adopt point-free notation for our assertions over program states.

abbreviation (*input*)

$pred_K :: 'b \Rightarrow 'a \Rightarrow 'b (\langle _ \rangle)$ **where**
 $\langle f \rangle \equiv \lambda s. f$

abbreviation (*input*)

$pred_not :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool (\neg)$ **where**
 $\neg a \equiv \lambda s. \neg a s$

abbreviation (*input*)

$pred_conj :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** \wedge 35) **where**
 $a \wedge b \equiv \lambda s. a s \wedge b s$

abbreviation (*input*)

$pred_implies :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** \longrightarrow 25) **where**
 $a \longrightarrow b \equiv \lambda s. a s \longrightarrow b s$

abbreviation (*input*)

$pred_eq :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** $=$ 40) **where**
 $a = b \equiv \lambda s. a s = b s$

abbreviation (*input*)

$pred_member :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b \text{ set}) \Rightarrow 'a \Rightarrow bool$ (**infix** \in 40) **where**
 $a \in b \equiv \lambda s. a s \in b s$

abbreviation (*input*)

$pred_neq :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** \neq 40) **where**
 $a \neq b \equiv \lambda s. a s \neq b s$

abbreviation (*input*)

$pred_If :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**if** $(_)$ / **then** $(_)$ / **else** $(_)$) [0, 0, 10] 10) **where**
if P **then** x **else** $y \equiv \lambda s. \text{if } P s \text{ then } x s \text{ else } y s$

abbreviation (*input*)

$pred_less :: ('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** $<$ 40) **where**
 $a < b \equiv \lambda s. a s < b s$

abbreviation (*input*)

$pred_le :: ('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** \leq 40) **where**
 $a \leq b \equiv \lambda s. a s \leq b s$

abbreviation (*input*)

$pred_plus :: ('a \Rightarrow 'b::plus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $+$ 65) **where**
 $a + b \equiv \lambda s. a s + b s$

abbreviation (*input*)

$pred_minus :: ('a \Rightarrow 'b::minus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $-$ 65) **where**
 $a - b \equiv \lambda s. a s - b s$

abbreviation (*input*)

$fun_fanout :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infix** \bowtie 35) **where**
 $f \bowtie g \equiv \lambda x. (f x, g x)$

abbreviation (*input*)

$pred_all :: ('b \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**binder** \forall 10) **where**
 $\forall x. P x \equiv \lambda s. \forall x. P x s$

abbreviation (*input*)

$pred_ex :: ('b \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**binder** \exists 10) **where**
 $\exists x. P x \equiv \lambda s. \exists x. P x s$

3 “Monoidal” Hoare logic

In the absence of a general-purpose development of Hoare Logic for total correctness in Isabelle/HOL¹, we adopt the following syntactic contrivance that eases making multiple assertions about function results. “Programs” consist of the state-transformer semantics of statements.

definition $valid :: ('s \Rightarrow bool) \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow bool) \Rightarrow bool$ ($\{_\}/ _ / \{_\}$) **where**
 $\{P\} c \{Q\} \equiv \forall s. P s \longrightarrow Q (c s)$

notation (*input*) *id* (*SKIP*)

notation *fcomp* (**infixl** ;; 60)

named_theorems *wp_intro* *weakest precondition intro rules*

lemma $seqI[wp_intro]$:

assumes $\{Q\} d \{R\}$

assumes $\{P\} c \{Q\}$

shows $\{P\} c ;; d \{R\}$

<proof>

lemma $iteI[wp_intro]$:

assumes $\{P'\} x \{Q\}$

assumes $\{P''\} y \{Q\}$

shows $\{\text{if } b \text{ then } P' \text{ else } P''\} \text{if } b \text{ then } x \text{ else } y \{Q\}$

<proof>

lemma $assignI[wp_intro]$:

shows $\{Q \circ f\} f \{Q\}$

<proof>

lemma $whileI$:

assumes $\{I'\} c \{I\}$

assumes $\bigwedge s. I s \Longrightarrow \text{if } b \text{ s then } I' \text{ s else } Q \text{ s}$

assumes $wf \ r$

assumes $\bigwedge s. \llbracket I \text{ s}; b \text{ s} \rrbracket \Longrightarrow (c \text{ s}, s) \in r$

shows $\{I\} \text{while } b \text{ c } \{Q\}$

<proof>

lemma $hoare_pre$:

assumes $\{R\} f \{Q\}$

assumes $\bigwedge s. P s \Longrightarrow R s$

shows $\{P\} f \{Q\}$

<proof>

lemma $hoare_post_imp$:

assumes $\{P\} a \{Q\}$

assumes $\bigwedge s. Q s \Longrightarrow R s$

shows $\{P\} a \{R\}$

<proof>

Note that the $assignI$ rule applies to all state transformers, and therefore the order in which we attempt to use the wp_intro rules matters.

4 Properties of iterated functions on finite sets

We begin by fixing the f and $x0$ under consideration in a locale, and establishing Knuth’s properties.

The sequence is modelled as a function $seq :: nat \Rightarrow 'a$ in the obvious way.

¹At the time of writing the distribution contains several for partial correctness, and one for total correctness over a language with restricted expressions. SIMPL (Schirmer (2008)) is overkill for our present purposes.

```

locale fx0 =
  fixes f :: 'a::finite  $\Rightarrow$  'a
  fixes x0 :: 'a
begin

```

```

definition seq' :: 'a  $\Rightarrow$  nat  $\Rightarrow$  'a where
  seq' x i  $\equiv$  (f  $\widehat{\sim}$  i) x

```

```

abbreviation seq  $\equiv$  seq' x0 <proof> <proof>

```

The parameters *lambda* and *mu* must exist by the pigeonhole principle.

```

lemma seq'_not_inj_on_card_UNIV:
  shows  $\neg$ inj_on (seq' x) {0 .. card (UNIV::'a set)}
<proof>

```

```

definition properties :: nat  $\Rightarrow$  nat  $\Rightarrow$  bool where
  properties lambda mu  $\equiv$ 
    0 < lambda
     $\wedge$  inj_on seq {0 ..< mu + lambda}
     $\wedge$  ( $\forall i \geq mu. \forall j. seq (i + j * lambda) = seq i$ )

```

```

lemma properties_existence:
  obtains lambda mu
  where properties lambda mu
<proof>

```

end

To ease further reasoning, we define a new locale that fixes *lambda* and *mu*, and assume these properties hold. We then derive further rules that are easy to apply.

```

locale properties = fx0 +
  fixes lambda mu :: nat
  assumes P: properties lambda mu
begin

```

```

lemma properties_lambda_gt_0:
  shows 0 < lambda
<proof>

```

```

lemma properties_loop:
  assumes mu  $\leq$  i
  shows seq (i + j * lambda) = seq i
<proof>

```

```

lemma properties_mod_lambda:
  assumes mu  $\leq$  i
  shows seq i = seq (mu + (i - mu) mod lambda)
<proof>

```

```

lemma properties_distinct:
  assumes j  $\in$  {0 <..lambda}
  shows seq (i + j)  $\neq$  seq i
<proof>

```

```

lemma properties_distinct_contrapos:
  assumes seq (i + j) = seq i
  shows j  $\notin$  {0 <..lambda}
<proof>

```

```

lemma properties_loops_ge_mu:
  assumes seq (i + j) = seq i
  assumes 0 < j
  shows mu  $\leq$  i
<proof>

```

end

5 The Tortoise and the Hare

The key to the Tortoise and Hare algorithm is that any nu such that $seq (nu + nu) = seq nu$ must be divisible by $lambda$. Intuitively the first nu steps get us into the loop. If the second nu steps return us to the same value of the sequence, then we must have gone around the loop one or more times.

lemma (in *properties*) *lambda_dvd_nu*:
 assumes $seq (i + i) = seq i$
 shows $lambda \text{ dvd } i$
(*proof*)

The program is split into three loops; we find nu , mu and $lambda$ in that order.

5.1 Finding nu

The state space of the program tracks each of the variables we wish to discover, and the current positions of the Tortoise and Hare.

record 'a state =
 nu :: nat — ν
 m :: nat — μ
 l :: nat — λ
 hare :: 'a
 tortoise :: 'a

context *properties*
begin

The Hare proceeds at twice the speed of the Tortoise. The program tracks how many steps the Tortoise has taken in nu .

definition (in *fx0*) *find_nu* :: 'a state \Rightarrow 'a state **where**
 find_nu \equiv
 ($\lambda s. s(| nu := 1, tortoise := f(x0), hare := f(f(x0)) |) |$);;
 while ($hare \neq tortoise$)
 ($\lambda s. s(| nu := nu s + 1, tortoise := f(tortoise s), hare := f(f(hare s)) |)$)

If this program terminates, we expect $seq \circ (nu + nu) = seq \circ nu$ to hold in the final state.

The simplest approach to showing termination is to define a suitable nu in terms of $lambda$ and mu , which also gives us an upper bound on the number of calls to f .

definition *nu_witness* :: nat **where**
 nu_witness $\equiv mu + lambda - mu \text{ mod } lambda$

This constant has the following useful properties:

lemma *nu_witness_properties*:
 $mu < nu_witness$
 $nu_witness \leq lambda + mu$
 $lambda \text{ dvd } nu_witness$
 $mu = 0 \implies nu_witness = lambda$
(*proof*)

These demonstrate that *nu_witness* has the key property:

lemma *nu_witness*:
 shows $seq (nu_witness + nu_witness) = seq nu_witness$
(*proof*)

Termination amounts to showing that the Tortoise gets closer to *nu_witness* on each iteration of the loop.

definition *find_nu_measure* :: (nat \times nat) set **where**
 find_nu_measure $\equiv measure (\lambda \nu. nu_witness - \nu)$

lemma *find_nu_measure_wellfounded*:
 wf *find_nu_measure*

$\langle proof \rangle$

lemma *find_nu_measure_decreases*:

assumes $seq (\nu + \nu) \neq seq \nu$

assumes $\nu \leq nu_witness$

shows $(Suc \nu, \nu) \in find_nu_measure$

$\langle proof \rangle$

The remainder of the Hoare proof is straightforward.

lemma *find_nu*:

$\{\langle True \rangle\} find_nu \{ \nu \in \{0 <.. lambda + mu\} \wedge seq \circ (nu + nu) = seq \circ nu \wedge hare = seq \circ nu \}$

$\langle proof \rangle$

5.1.1 Side observations

We can also show termination ala [Filliâtre \(2007\)](#).

definition *find_nu_measures* :: $(nat \times nat)$ set **where**

find_nu_measures \equiv

measures $[\lambda \nu. mu - \nu, \lambda \nu. LEAST i. seq (\nu + \nu + i) = seq \nu]$

lemma *find_nu_measures_wellfounded*:

wf find_nu_measures

$\langle proof \rangle$

lemma *find_nu_measures_existence*:

assumes $\nu: mu \leq \nu$

shows $\exists i. seq (\nu + \nu + i) = seq \nu$

$\langle proof \rangle$

lemma *find_nu_measures_decreases*:

assumes $\nu: seq (\nu + \nu) \neq seq \nu$

shows $(Suc \nu, \nu) \in find_nu_measures$

$\langle proof \rangle$

lemma *find_nu_Filliâtre*:

$\{\langle True \rangle\} find_nu \{ \langle 0 \rangle < nu \wedge seq \circ (nu + nu) = seq \circ nu \wedge hare = seq \circ nu \}$

$\langle proof \rangle$

This approach does not provide an upper bound on nu however.

[Harper \(2011\)](#) observes (in his §13.5.2) that if mu is zero then $nu = lambda$.

lemma *Harper*:

assumes $mu = 0$

shows $\{\langle True \rangle\} find_nu \{ nu = \langle lambda \rangle \}$

$\langle proof \rangle$

5.2 Finding mu

We recover mu from nu by exploiting the fact that $lambda$ divides nu : the Tortoise, reset to $x0$ and the Hare, both now moving at the same speed, will meet at mu .

lemma *mu_nu*:

assumes $si: seq (i + i) = seq i$

assumes $j: mu \leq j$

shows $seq (j + i) = seq j$

$\langle proof \rangle$

definition (in *fx0*) *find_mu* :: 'a state \Rightarrow 'a state **where**

find_mu \equiv

$(\lambda s. s(\ m := 0, tortoise := x0 \))$;;

while ($hare \neq tortoise$)

$(\lambda s. s(\ tortoise := f (tortoise s), hare := f (hare s), m := m s + 1 \))$

lemma *find_mu*:

$\{ nu \in \{0 <.. lambda + mu\} \wedge seq \circ (nu + nu) = seq \circ nu \wedge hare = seq \circ nu \}$

```

    find_mu
    {nu ∈ {0<..lambda + mu}} ∧ tortoise = ⟨seq mu⟩ ∧ m = ⟨mu⟩
  ⟨proof⟩

```

5.3 Finding lambda

With the Tortoise parked at mu , we find $lambda$ by walking the Hare around the loop.

definition (in $fx0$) $find_lambda :: 'a\ state \Rightarrow 'a\ state$ **where**

```

  find_lambda ≡
    (λs. s( l := 1, hare := f (tortoise s) )) ;;
    while (hare ≠ tortoise)
      (λs. s( hare := f (hare s), l := l s + 1 ))

```

lemma $find_lambda$:

```

  {nu ∈ {0<..lambda + mu}} ∧ tortoise = ⟨seq mu⟩ ∧ m = ⟨mu⟩
  find_lambda
  {nu ∈ {0<..lambda + mu}} ∧ l = ⟨lambda⟩ ∧ m = ⟨mu⟩
  ⟨proof⟩

```

5.4 Top level

The complete program is simply the steps composed in order.

definition (in $fx0$) $tortoise_hare :: 'a\ state \Rightarrow 'a\ state$ **where**

```

  tortoise_hare ≡ find_nu ;; find_mu ;; find_lambda

```

theorem $tortoise_hare$:

```

  {⟨True⟩} tortoise_hare {nu ∈ {0<..lambda + mu}} ∧ l = ⟨lambda⟩ ∧ m = ⟨mu⟩
  ⟨proof⟩

```

end

corollary $tortoise_hare_correct$:

```

  assumes s': s' = fx0.tortoise_hare f x arbitrary
  shows fx0.properties f x (l s') (m s')
  ⟨proof⟩

```

Isabelle can generate code from these definitions.

schematic_goal $tortoise_hare_code[code]$:

```

  fx0.tortoise_hare f x = ?code
  ⟨proof⟩

```

export_code $fx0.tortoise_hare$ **in** SML

6 Brent's algorithm

[Brent \(1980\)](#) improved on the Tortoise and Hare algorithm and used it to factor large primes. In practice it makes significantly fewer calls to the function f before detecting a loop.

We begin by defining the base-2 logarithm.

fun $lg :: nat \Rightarrow nat$ **where**

```

  [simp del]: lg x = (if x ≤ 1 then 0 else 1 + lg (x div 2))

```

lemma lg_safe :

```

  lg 0 = 0
  lg (Suc 0) = 0
  lg (Suc (Suc 0)) = 1
  0 < x ⇒ lg (x + x) = 1 + lg x
  ⟨proof⟩

```

lemma lg_inv :

```

  0 < x ⇒ lg (2 ^ x) = x
  ⟨proof⟩

```

lemma *lg_inv2*:

$\langle 2 \wedge \text{lg } x = x \rangle$ **if** $\langle 2 \wedge i = x \rangle$ **for** x
<proof>

lemmas *lg_simps* = *lg_safe lg_inv lg_inv2*

6.1 Finding *lambda*

Imagine now that the Tortoise carries an unbounded number of carrots, which he passes to the Hare when they meet, and the Hare has a teleporter. The Hare eats a carrot each time she waits for the function f to execute, and initially has just one. If she runs out of carrots before meeting the Tortoise again, she teleports him to her position, and he gives her twice as many carrots as the last time they met (tracked by the variable *carrots*). By counting how many carrots she has eaten from when she last teleported the Tortoise (recorded in l) until she finally has surplus carrots when she meets him again, the Hare directly discovers *lambda*.

record 'a *state* =

$m :: \text{nat} \text{ — } \mu$
 $l :: \text{nat} \text{ — } \lambda$
 $\text{carrots} :: \text{nat}$
 $\text{hare} :: 'a$
 $\text{tortoise} :: 'a$

context *properties*
begin

definition (in *fx0*) *find_lambda* :: 'a *state* \Rightarrow 'a *state* **where**

find_lambda \equiv
 $(\lambda s. s(| \text{carrots} := 1, l := 1, \text{tortoise} := x0, \text{hare} := f\ x0 \ |)) ;;$
 $\text{while } (\text{hare} \neq \text{tortoise})$
 $((\text{if } \text{carrots} = l \text{ then } (\lambda s. s(| \text{tortoise} := \text{hare } s, \text{carrots} := 2 * \text{carrots } s, l := 0 \ |))$
 $\quad \text{else } \text{SKIP}) ;;$
 $(\lambda s. s(| \text{hare} := f (\text{hare } s), l := l\ s + 1 \ |)))$

The termination argument goes intuitively as follows. The Hare eats as many carrots as it takes to teleport the Tortoise into the loop. Afterwards she continues the teleportation dance until the Tortoise has given her enough carrots to make it all the way around the loop and back to him.

We can calculate the Tortoise's position as a function of *carrots*.

definition *carrots_total* :: *nat* \Rightarrow *nat* **where**

carrots_total $c \equiv \sum_{i < \text{lg } c} 2 \wedge i$

lemma *carrots_total_simps*:

$\text{carrots_total } (\text{Suc } 0) = 0$
 $\text{carrots_total } (\text{Suc } (\text{Suc } 0)) = 1$
 $2 \wedge i = c \implies \text{carrots_total } (c + c) = c + \text{carrots_total } c$
<proof>

definition *find_lambda_measures* :: $((\text{nat} \times \text{nat}) \times (\text{nat} \times \text{nat}))$ *set* **where**

find_lambda_measures \equiv
 $\text{measures } [\lambda(l, c). \mu - \text{carrots_total } c,$
 $\quad \lambda(l, c). \text{LEAST } i. \text{lambda} \leq c * 2 \wedge i,$
 $\quad \lambda(l, c). c - l]$

lemma *find_lambda_measures_wellfounded*:

$\text{wf } \text{find_lambda_measures}$
<proof>

lemma *find_lambda_measures_decreases1*:

assumes $c = 2 \wedge i$
assumes $\mu \leq \text{carrots_total } c \implies c \leq \text{lambda}$
assumes $\text{seq } (\text{carrots_total } c) \neq \text{seq } (\text{carrots_total } c + c)$
shows $((c', 2 * c), (c, c)) \in \text{find_lambda_measures}$
<proof>

lemma *find_lambda_measures_decreases2*:

assumes $ls < c$
shows $((Suc\ ls, c), (ls, c)) \in find_lambda_measures$
 $\langle proof \rangle$

lemma *find_lambda*:
 $\{\langle True \rangle\} find_lambda \{l = \langle lambda \rangle\}$
 $\langle proof \rangle$

6.2 Finding mu

With $lambda$ in hand, we can find mu using the same approach as for the Tortoise and Hare (§5.2), after we first move the Hare to $lambda$.

definition (in $fx0$) *find_mu* :: 'a state \Rightarrow 'a state **where**
 $find_mu \equiv$
 $(\lambda s. s \{ m := 0, tortoise := x0, hare := seq (l\ s) \}) ;;$
 $while (hare \neq tortoise)$
 $(\lambda s. s \{ tortoise := f (tortoise\ s), hare := f (hare\ s), m := m\ s + 1 \})$

lemma *find_mu*:
 $\{l = \langle lambda \rangle\} find_mu \{l = \langle lambda \rangle \wedge m = \langle mu \rangle\}$
 $\langle proof \rangle$

6.3 Top level

definition (in $fx0$) *brent* :: 'a state \Rightarrow 'a state **where**
 $brent \equiv find_lambda ;; find_mu$

theorem *brent*:
 $\{\langle True \rangle\} brent \{l = \langle lambda \rangle \wedge m = \langle mu \rangle\}$
 $\langle proof \rangle$

end

corollary *brent_correct*:
assumes $s': s' = fx0.brent\ f\ x$ *arbitrary*
shows $fx0.properties\ f\ x (l\ s') (m\ s')$
 $\langle proof \rangle$

schematic_goal *brent_code*[$code$]:
 $fx0.brent\ f\ x = ?code$
 $\langle proof \rangle$

export_code $fx0.brent$ **in** *SML*

7 Concluding remarks

Leino (2012) uses an SMT solver to verify a Tortoise-and-Hare cycle-finder. He finds the parameters $lambda$ and mu initially by using a “ghost” depth-first search, while we use more economical non-constructive methods.

I thank Christian Griset for patiently discussing the finer details of the proofs, and Makarius for many helpful suggestions.

References

- Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980. URL <https://doi.org/10.1007/BF01933190>.
- Jean-Christophe Filliâtre. Tortoise and hare algorithm, 2007. URL <http://www.lix.polytechnique.fr/coq/pylons/contribs/files/TortoiseHareAlgorithm/v8.4/TortoiseHareAlgorithm.TortoiseHareAlgorithm.html>.
- Robert Harper. *Programming in Standard ML*. Unpublished, 2011. URL <http://www.cs.cmu.edu/~rwh/smlbook/>.

- Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.
- K. Rustan M. Leino. Automating induction with an SMT solver. In *VMCAI 2012*, pages 315–331, 2012. URL https://doi.org/10.1007/978-3-642-27940-9_21.
- Gabriel Nivasch. Cycle detection using a stack. *Inf. Process. Lett.*, 90(3):135–140, 2004. URL <http://www.gabrielnivasch.org/fun/cycle-detection>.
- Norbert Schirmer. A sequential imperative programming language: Syntax, semantics, Hoare logics and verification environment. *Archive of Formal Proofs*, 2008. URL <http://isa-afp.org/entries/Simpl.shtml>.
- Ping Wang and Fangguo Zhang. An efficient collision detection method for computing discrete logarithms with Pollard’s rho. *J. Applied Mathematics*, 2012, 2012. URL <https://doi.org/10.1155/2012/635909>.