The Tortoise and the Hare Algorithm

Peter Gammie

March 17, 2025

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

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 $f : X \to X$, 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

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

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*)
$pred\_K :: 'b \Rightarrow 'a \Rightarrow 'b$ (‹⟨_⟩›) **where**
$\langle f \rangle \equiv \lambda s.\ f$

abbreviation (*input*)
$pred\_not :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (‹¬›) **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** ‹∧› *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** ‹⟶› *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*)
$pred\_member :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow bool$ (**infix** ‹∈› *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** ‹≠› *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** (_)/ **t**hen (_)/ **e**lse (_))› [*0*, *0*, *10*] *10*) **where**
if P **t**hen x **e**lse $y \equiv \lambda s.$ *if* $P\ s$ *then* $x\ s$ *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** ‹≤› *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** ‹⋈› *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** ‹∀ › *10*) **where**
$\forall x.\ P\ x \equiv \lambda s.\ \forall x.\ P\ x\ s$

abbreviation (*input*)

```
  pred_ex :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (binder ‹∃ › 10) where
  ∃ x. P x ≡ λs. ∃ x. P x s
```

3 “Monoidal” Hoare logic

In the absence of a general-purpose development of Hoare Logic for total correctness in Isabelle/HOL[1], 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 ⇒ bool) ⇒ ('s ⇒ 's) ⇒ ('s ⇒ bool) ⇒ bool (‹⦃_⦄/ _/ ⦃_⦄›) where
  ⦃P⦄ c ⦃Q⦄ ≡ ∀ s. P s ⟶ 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⦄
using assms by (simp add: valid_def)

lemma iteI[wp_intro]:
  assumes ⦃P'⦄ x ⦃Q⦄
  assumes ⦃P''⦄ y ⦃Q⦄
  shows ⦃if b then P' else P''⦄ if b then x else y ⦃Q⦄
using assms by (simp add: valid_def)

lemma assignI[wp_intro]:
  shows ⦃Q ∘ f⦄ f ⦃Q⦄
by (simp add: valid_def)

lemma whileI:
  assumes ⦃I'⦄ c ⦃I⦄
  assumes ⋀s. I s ⟹ if b s then I' s else Q s
  assumes wf r
  assumes ⋀s. ⟦ I s; b s ⟧ ⟹ (c s, s) ∈ r
  shows ⦃I⦄ while b c ⦃Q⦄
using assms by (simp add: while_rule valid_def)

lemma hoare_pre:
  assumes ⦃R⦄ f ⦃Q⦄
  assumes ⋀s. P s ⟹ R s
  shows ⦃P⦄ f ⦃Q⦄
using assms by (simp add: valid_def)

lemma hoare_post_imp:
  assumes ⦃P⦄ a ⦃Q⦄
  assumes ⋀s. Q s ⟹ R s
  shows ⦃P⦄ a ⦃R⦄
using assms by (simp add: valid_def)
```

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* ⇒ *'a* in the obvious way.

[1] At 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 ⇒ 'a
  fixes x0 :: 'a
begin

definition seq' :: 'a ⇒ nat ⇒ 'a where
  seq' x i ≡ (f ^^ i) x

abbreviation seq ≡ seq' x0
```

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

```
lemma seq'_not_inj_on_card_UNIV:
  shows ¬inj_on (seq' x) {0 .. card (UNIV::'a set)}
by (simp add: inj_on_iff_eq_card)
   (metis UNIV_I card_mono finite lessI not_less subsetI)

definition properties :: nat ⇒ nat ⇒ bool where
  properties lambda mu ≡
     0 < lambda
   ∧ inj_on seq {0 ..< mu + lambda}
   ∧ (∀ i≥mu. ∀ j. seq (i + j * lambda) = seq i)

lemma properties_existence:
  obtains lambda mu
  where properties lambda mu
proof −
  obtain l where l: inj_on seq {0..l} ∧ ¬inj_on seq {0..Suc l}
    using ex_least_nat_less[where P=λub. ¬inj_on seq {0..ub} and n=card (UNIV :: 'a set)]
          seq'_not_inj_on_card_UNIV
    by fastforce
  moreover
  from l obtain mu where mu: mu ≤ l ∧ seq (Suc l) = seq mu
    by (fastforce simp: atLeastAtMostSuc_conv)
  moreover
  define lambda where lambda = l − mu + 1
  have seq (i + j * lambda) = seq i if mu ≤ i for i j
  using that proof (induct j)
    case (Suc j)
    from l mu have F: seq (l + j + 1) = seq (mu + j) for j
      by (fastforce elim: seq_inj)
    from mu Suc F[where j=i + j * lambda − mu] show ?case
      by (simp add: lambda_def field_simps)
  qed simp
  ultimately have properties lambda mu
    by (auto simp: properties_def lambda_def atLeastLessThanSuc_atLeastAtMost)
  then show thesis ..
qed

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
using P by (simp add: properties_def)

lemma properties_loop:
  assumes mu ≤ i
  shows seq (i + j * lambda) = seq i
using P assms by (simp add: properties_def)
```

```
lemma properties_mod_lambda:
  assumes mu ≤ i
  shows seq i = seq (mu + (i − mu) mod lambda)
using properties_loop[where i=mu + (i − mu) mod lambda and j=(i − mu) div lambda] assms
by simp

lemma properties_distinct:
  assumes j ∈ {0 <..< lambda}
  shows seq (i + j) ≠ seq i
proof(cases mu ≤ i)
  case True
  from assms have A: (i + j) mod lambda ≠ i mod lambda for i
    by (auto simp add: mod_eq_dvd_iff_nat)
  from ‹mu ≤ i›
  have seq (i + j) = seq (mu + (i + j − mu) mod lambda)
       seq i = seq (mu + (i − mu) mod lambda)
    by (auto intro: properties_mod_lambda)
  with P ‹mu ≤ i› assms A[where i=i−mu] show ?thesis
    by (clarsimp simp: properties_def inj_on_eq_iff)
next
  case False with P assms show ?thesis
    by (clarsimp simp: properties_def inj_on_eq_iff)
qed

lemma properties_distinct_contrapos:
  assumes seq (i + j) = seq i
  shows j ∉ {0 <..< lambda}
using assms by (rule contrapos_pp) (simp add: properties_distinct)

lemma properties_loops_ge_mu:
  assumes seq (i + j) = seq i
  assumes 0 < j
  shows mu ≤ i
proof(rule classical)
  assume X: ¬?thesis show ?thesis
  proof(cases mu ≤ i + j)
    case True with P X assms show ?thesis
      by (fastforce simp: properties_def inj_on_eq_iff
                    dest: properties_mod_lambda)
  next
    case False with P assms show ?thesis
      by (fastforce simp add: properties_def inj_on_eq_iff)
  qed
qed

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 dvd i
proof(cases i = 0)
  case False
  with assms have mu ≤ i by (auto simp: properties_loops_ge_mu)
  with assms have seq (i + i mod lambda) = seq i
    using properties_loop[where i=i + i mod lambda and j=i div lambda] by simp
  from properties_distinct_contrapos[OF this] show ?thesis
    by simp (meson dvd_eq_mod_eq_0 mod_less_divisor not_less properties_lambda_gt_0)
qed simp
```

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* ⇒ *'a state* **where**
  find_nu ≡
    (λ*s*. *s*⦇ *nu* := *1*, *tortoise* := *f*(*x0*), *hare* := *f*(*f*(*x0*)) ⦈) ;;
    while (*hare* ≠ *tortoise*)
        (λ*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 ≡ *mu* + *lambda* − *mu mod lambda*

This constant has the following useful properties:

lemma *nu_witness_properties*:
  mu < *nu_witness*
  nu_witness ≤ *lambda* + *mu*
  lambda dvd nu_witness
  mu = *0* ⟹ *nu_witness* = *lambda*
unfolding *nu_witness_def*
using *properties_lambda_gt_0*
apply (*simp_all add*: *less_diff_conv divide_simps*)
apply (*metis minus_mod_eq_div_mult* [*symmetric*] *dvd_def mod_add_self2 mult.commute*)
done

These demonstrate that *nu_witness* has the key property:

lemma *nu_witness*:
  shows *seq* (*nu_witness* + *nu_witness*) = *seq nu_witness*
using *nu_witness_properties properties_loop*
by (*clarsimp simp*: *dvd_def field_simps*)

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

definition *find_nu_measure* :: (*nat* × *nat*) *set* **where**
  find_nu_measure ≡ *measure* (λν. *nu_witness* − ν)

lemma *find_nu_measure_wellfounded*:
  wf find_nu_measure
by (*simp add*: *find_nu_measure_def*)

lemma *find_nu_measure_decreases*:
  assumes *seq* (ν + ν) ≠ *seq* ν
  assumes ν ≤ *nu_witness*
  shows (*Suc* ν, ν) ∈ *find_nu_measure*
using *nu_witness_properties nu_witness assms*
by (*auto simp*: *find_nu_measure_def le_eq_less_or_eq*)

The remainder of the Hoare proof is straightforward.

lemma *find_nu*:
  {|⟨*True*⟩|} *find_nu* {|*nu* ∈ ⟨{*0*<..*lambda* + *mu*}⟩ ∧ *seq* ∘ (*nu* + *nu*) = *seq* ∘ *nu* ∧ *hare* = *seq* ∘ *nu*|}
apply (*simp add*: *find_nu_def*)
apply (*rule hoare_pre*)
 apply (*rule whileI*[**where** *I*=*nu* ∈ ⟨{*0*<..*nu_witness*}⟩ ∧ (∀ *i*. ⟨*0* < *i*⟩ ∧ ⟨*i*⟩ < *nu* ⟶ ⟨*seq* (*i* + *i*) ≠ *seq i*⟩)
                                ∧ *tortoise* = *seq* ∘ *nu* ∧ *hare* = *seq* ∘ (*nu* + *nu*)
                            and *r*=*inv_image find_nu_measure nu*]
            wp_intro)+
    using *nu_witness_properties nu_witness*
    apply (*fastforce simp*: *le_eq_less_or_eq elim*: *less_SucE*)
   apply (*simp add*: *find_nu_measure_wellfounded*)
  apply (*simp add*: *find_nu_measure_decreases*)
 apply (*rule wp_intro*)
using *nu_witness_properties*
apply *auto*
done

5.1.1 Side observations

We can also show termination ala Filliâtre (2007).

definition *find_nu_measures* :: (*nat* × *nat*) *set* **where**
  find_nu_measures ≡
    measures [λν. *mu* − ν, λν. *LEAST i*. *seq* (ν + ν + *i*) = *seq* ν]

lemma *find_nu_measures_wellfounded*:
  wf find_nu_measures
by (*simp add*: *find_nu_measures_def*)

lemma *find_nu_measures_existence*:
  assumes ν: *mu* ≤ ν
  shows ∃ *i*. *seq* (ν + ν + *i*) = *seq* ν
proof(*cases seq* (ν + ν) = *seq* ν)
 case *False*
 from *properties_lambda_gt_0* **obtain** *k* **where** *k*: ν ≤ *k* ∗ *lambda*
   by (*metis One_nat_def Suc_leI mult.right_neutral mult_le_mono order_refl*)
 from ν *k* **have** *seq* (ν + ν + (*k* ∗ *lambda* − ν)) = *seq* (*mu* + (ν − *mu*) + *k* ∗ *lambda*) **by** (*simp add*: *field_simps*)
 also from ν *properties_loop* **have** … = *seq* ν **by** *simp*
 finally show *?thesis* **by** *blast*
qed (*simp add*: *exI*[**where** *x*=*0*])

lemma *find_nu_measures_decreases*:
  assumes ν: *seq* (ν + ν) ≠ *seq* ν
  shows (*Suc* ν, ν) ∈ *find_nu_measures*
proof(*cases mu* ≤ ν)
  case *True*
  then have *mu* ≤ *Suc* ν **by** *simp*
  have (*LEAST i*. *seq* (*Suc* ν + *Suc* ν + *i*) = *seq* (*Suc* ν)) < (*LEAST i*. *seq* (ν + ν + *i*) = *seq* ν)
  proof(*rule LeastI2_wellorder_ex*[*OF find_nu_measures_existence*[*OF* ‹*mu* ≤ *Suc* ν›]],
        rule LeastI2_wellorder_ex[*OF find_nu_measures_existence*[*OF* ‹*mu* ≤ ν›]])
    fix *x y*
    assume *x*: *seq* (*Suc* ν + *Suc* ν + *x*) = *seq* (*Suc* ν)
              ∀ *z*. *seq* (*Suc* ν + *Suc* ν + *z*) = *seq* (*Suc* ν) ⟶ *x* ≤ *z*
    assume *y*: *seq* (ν + ν + *y*) = *seq* ν
    from ν ‹*mu* ≤ ν› *y* **have** *0* < *y* **by** (*cases y*) *simp_all*
    with *y* **have** *seq* (*Suc* ν + *Suc* ν + (*y* − *1*)) = *seq* (*Suc* ν) **by** (*auto elim*: *seq_inj*)
    with ‹*0* < *y*› *spec*[*OF x*(*2*), **where** *x*=*y* − *1*] *y* **show** *x* < *y* **by** *simp*
  qed
  with *True* ν **show** *?thesis* **by** (*simp add*: *find_nu_measures_def*)
qed (*auto simp*: *find_nu_measures_def*)

lemma *find_nu_Filliâtre*:
  {|⟨*True*⟩|} *find_nu* {|⟨*0*⟩ < *nu* ∧ *seq* ∘ (*nu* + *nu*) = *seq* ∘ *nu* ∧ *hare* = *seq* ∘ *nu*|}
apply (*simp add*: *find_nu_def*)

```
apply (rule hoare_pre)
 apply (rule whileI[where I=⟨0⟩ < nu ∧ tortoise = seq ∘ nu ∧ hare = seq ∘ (nu + nu)
                   and r=inv_image find_nu_measures nu]
             wp_intro)+
    apply clarsimp
   apply (simp add: find_nu_measures_wellfounded)
  apply (simp add: find_nu_measures_decreases)
 apply (rule wp_intro)
apply (simp add: properties_lambda_gt_0)
done
```

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 ⦃⟨True⟩⦄ find_nu ⦃nu = ⟨lambda⟩⦄
by (rule hoare_post_imp[OF find_nu]) (fastforce simp: assms dvd_def dest: lambda_dvd_nu)
```

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 ≤ j
  shows seq (j + i) = seq j
using lambda_dvd_nu[OF si] properties_loop[OF j]
by (clarsimp simp: dvd_def field_simps)

definition (in fx0) find_mu :: 'a state ⇒ 'a state where
  find_mu ≡
    (λs. s⦇ m := 0, tortoise := x0 ⦈) ;;
    while (hare ≠ tortoise)
          (λs. s⦇ tortoise := f (tortoise s), hare := f (hare s), m := m s + 1 ⦈)

lemma find_mu:
  ⦃nu ∈ ⟨{0<..lambda + mu}⟩ ∧ seq ∘ (nu + nu) = seq ∘ nu ∧ hare = seq ∘ nu⦄
     find_mu
   ⦃nu ∈ ⟨{0<..lambda + mu}⟩ ∧ tortoise = ⟨seq mu⟩ ∧ m = ⟨mu⟩⦄
apply (simp add: find_mu_def)
apply (rule hoare_pre)
 apply (rule whileI[where I=nu ∈ ⟨{0<..lambda + mu}⟩ ∧ seq ∘ (nu + nu) = seq ∘ nu ∧ m ≤ ⟨mu⟩
                        ∧ tortoise = seq ∘ m ∧ hare = seq ∘ (m + nu)
                   and r=measure (⟨mu⟩ − m)]
             wp_intro)+
    using properties_loops_ge_mu
    apply (force dest: mu_nu simp: less_eq_Suc_le[symmetric])
   apply simp
  apply (force dest: mu_nu simp: le_eq_less_or_eq)
 apply (rule wp_intro)
apply simp
done
```

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 ⇒ '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⟩|}
apply (simp add: find_lambda_def)
apply (rule hoare_pre)
 apply (rule whileI[where I=nu ∈ ⟨{0<..lambda + mu}⟩ ∧ l ∈ ⟨{0<..lambda}⟩
                          ∧ tortoise = ⟨seq mu⟩ ∧ hare = seq ∘ (⟨mu⟩ + l) ∧ m = ⟨mu⟩
                      and r=measure (⟨lambda⟩ − l)]
            wp_intro)+
    using properties_lambda_gt_0 properties_mod_lambda[where i=mu + lambda] properties_distinct[where i=mu]
    apply (fastforce simp: less_eq_Suc_le[symmetric])
   apply simp
  using properties_mod_lambda[where i=mu + lambda]
  apply (fastforce simp: le_eq_less_or_eq)
 apply (rule wp_intro)
using properties_lambda_gt_0
apply simp
done
```

5.4 Top level

The complete program is simply the steps composed in order.

```
definition (in fx0) tortoise_hare :: 'a state ⇒ '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⟩|}
unfolding tortoise_hare_def
by (rule find_nu find_mu find_lambda wp_intro)+

end

corollary tortoise_hare_correct:
  assumes s': s' = fx0.tortoise_hare f x arbitrary
  shows fx0.properties f x (l s') (m s')
using assms properties.tortoise_hare[where f=f and ?x0.0=x]
by (fastforce intro: fx0.properties_existence[where f=f and ?x0.0=x]
              simp: Basis.properties_def valid_def)
```

Isabelle can generate code from these definitions.

```
schematic_goal tortoise_hare_code[code]:
  fx0.tortoise_hare f x = ?code
unfolding fx0.tortoise_hare_def fx0.find_nu_def fx0.find_mu_def fx0.find_lambda_def fcomp_assoc[symmetric] fcomp_comp
by (rule refl)

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 ⇒ 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
by (simp_all add: lg.simps)

lemma lg_inv:
```

0 < x ⟹ lg (2 ^ x) = x
proof(*induct x*)
case (*Suc x*) **then show** *?case*
by (*cases x*, *simp_all add*: *lg.simps Suc_lessI not_le*)
qed *simp*

lemma *lg_inv2*:
‹*2 ^ lg x = x*› **if** ‹*2 ^ i = x*› **for** *x*
proof −
have ‹*2 ^ lg (2 ^ i) = (2::nat) ^ i*›
by (*induction i*) (*simp_all add*: *lg_safe mult_2*)
with *that* **show** *?thesis*
by *simp*
qed

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 :: nat  — μ
  l :: nat  — λ
  carrots :: nat
  hare :: 'a
  tortoise :: 'a

context properties
begin

definition (in fx0) find_lambda :: 'a state ⇒ 'a state where
  find_lambda ≡
    (λs. s(| carrots := 1, l := 1, tortoise := x0, hare := f x0 |)) ;;
    while (hare ≠ tortoise)
          ( ( if carrots = l then (λs. s(| tortoise := hare s, carrots := 2 ∗ carrots s, l := 0 |))
                             else SKIP ) ;;
            (λs. s(| hare := f (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 ⇒ nat where
  carrots_total c ≡ ∑i<lg c. 2 ^ i

lemma carrots_total_simps:
  carrots_total (Suc 0) = 0
  carrots_total (Suc (Suc 0)) = 1
  2 ^ i = c ⟹ carrots_total (c + c) = c + carrots_total c
by (auto simp: carrots_total_def lg_simps)

definition find_lambda_measures :: ( (nat × nat) × (nat × nat) ) set where
  find_lambda_measures ≡
    measures [λ(l, c). mu − carrots_total c,
              λ(l, c). LEAST i. lambda ≤ c ∗ 2^i,
              λ(l, c). c − l]

lemma find_lambda_measures_wellfounded:
```

```
  wf find_lambda_measures
by (simp add: find_lambda_measures_def)

lemma find_lambda_measures_decreases1:
  assumes c = 2 ^ i
  assumes mu ≤ carrots_total c ⟶ c ≤ lambda
  assumes seq (carrots_total c) ≠ seq (carrots_total c + c)
  shows ( (c', 2 * c), (c, c) ) ∈ find_lambda_measures
proof(cases mu ≤ carrots_total c)
  case False with assms show ?thesis
    by (auto simp: find_lambda_measures_def carrots_total_simps mult_2 field_simps diff_less_mono2)
next
  case True
  { fix x assume x: (0::nat) < x have ∃ n. lambda ≤ x * 2 ^ n
    proof(induct lambda)
      case (Suc i)
      then obtain n where i ≤ x * 2 ^ n by blast
      with x show ?case
        by (clarsimp intro!: exI[where x=Suc n] simp: field_simps mult_2)
              (metis Nat.add_0_right Suc_leI linorder_neqE_nat mult_eq_0_iff add_left_cancel not_le numeral_2_eq_2
old.nat.distinct(2) power_not_zero trans_le_add2)
    qed simp } note ex = this
  have (LEAST j. lambda ≤ 2 ^(i + 1) * 2 ^j) < (LEAST j. lambda ≤ 2 ^ i * 2 ^j)
  proof(rule LeastI2_wellorder_ex[OF ex, rotated], rule LeastI2_wellorder_ex[OF ex, rotated])
    fix x y
    assume lambda ≤ 2 ^ i * 2 ^ y
           lambda ≤ 2 ^(i + 1) * 2 ^ x
           ∀ z. lambda ≤ 2 ^(i + 1) * 2 ^ z ⟶ x ≤ z
    with True assms properties_loop[where i=carrots_total c and j=1]
    show x < y by (cases y, auto simp: less_Suc_eq_le)
  qed simp_all
  with True ‹c = 2 ^ i› show ?thesis
    by (clarsimp simp: find_lambda_measures_def mult_2 carrots_total_simps field_simps power_add)
qed

lemma find_lambda_measures_decreases2:
  assumes ls < c
  shows ( (Suc ls, c), (ls, c) ) ∈ find_lambda_measures
using assms by (simp add: find_lambda_measures_def)

lemma find_lambda:
  ⦃⟨True⟩⦄ find_lambda ⦃l = ⟨lambda⟩⦄
apply (simp add: find_lambda_def)
apply (rule hoare_pre)
apply (rule whileI[where I=⟨0⟩ < l ∧ l ≤ carrots ∧ (⟨mu⟩ ≤ carrots_total ∘ carrots ⟶ l ≤ ⟨lambda⟩) ∧ (∃ i. carrots
= ⟨2^i⟩)
                                ∧ tortoise = seq ∘ carrots_total ∘ carrots ∧ hare = seq ∘ (l + (carrots_total ∘ carrots))
                          and r=inv_image find_lambda_measures (l ⋈ carrots)]
            wp_intro)+
   using properties_lambda_gt_0
   apply (clarsimp simp: field_simps mult_2_right carrots_total_simps)
   apply (intro conjI impI)
      apply (metis mult_2 power_Suc)
     apply (case_tac mu ≤ carrots_total (l s))
      apply (cut_tac i=carrots_total (l s) and j=l s in properties_distinct_contrapos, simp_all add: field_simps)[1]
     apply (cut_tac i=carrots_total (l s) and j=l s in properties_loops_ge_mu, simp_all add: field_simps)[1]
    apply (cut_tac i=carrots_total (2 ^ x) and j=1 in properties_loop, simp)
    apply (fastforce simp: le_eq_less_or_eq field_simps)
   apply (cut_tac i=carrots_total (2 ^ x) and j=l s in properties_loops_ge_mu, simp_all add: field_simps)[1]
   apply (cut_tac i=carrots_total (2 ^ x) and j=l s in properties_distinct_contrapos, simp_all add: field_simps)[1]
  apply (simp add: find_lambda_measures_wellfounded)
 apply (clarsimp simp: add.commute find_lambda_measures_decreases1 find_lambda_measures_decreases2)
apply (rule wp_intro)
using properties_lambda_gt_0
apply (simp add: carrots_total_simps exI[where x=0])
```

done

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 ⇒ 'a state where
  find_mu ≡
    (λs. s⦇ m := 0, tortoise := x0, hare := seq (l s) ⦈) ;;
    while (hare ≠ tortoise)
          (λs. s⦇ tortoise := f (tortoise s), hare := f (hare s), m := m s + 1 ⦈)

lemma find_mu:
  ⦃l = ⟨lambda⟩⦄ find_mu ⦃l = ⟨lambda⟩ ∧ m = ⟨mu⟩⦄
apply (simp add: find_mu_def)
apply (rule hoare_pre)
apply (rule whileI[where I=l = ⟨lambda⟩ ∧ m ≤ ⟨mu⟩ ∧ tortoise = seq ∘ m ∧ hare = seq ∘ (m + l)
                     and r=measure (⟨mu⟩ − m)]
          wp_intro)+
   using properties_lambda_gt_0 properties_loop[where i=mu and j=1]
   apply (fastforce simp: le_less dest: properties_loops_ge_mu)
  apply simp
 using properties_loop[where i=mu and j=1, simplified]
 apply (fastforce simp: le_eq_less_or_eq)
apply (rule wp_intro)
apply simp
done
```

6.3 Top level

```
definition (in fx0) brent :: 'a state ⇒ 'a state where
  brent ≡ find_lambda ;; find_mu

theorem brent:
  ⦃⟨True⟩⦄ brent ⦃l = ⟨lambda⟩ ∧ m = ⟨mu⟩⦄
unfolding brent_def
by (rule find_lambda find_mu wp_intro)+

end

corollary brent_correct:
  assumes s': s' = fx0.brent f x arbitrary
  shows fx0.properties f x (l s') (m s')
using assms properties.brent[where f=f and ?x0.0=x]
by (fastforce intro: fx0.properties_existence[where f=f and ?x0.0=x]
              simp:  Basis.properties_def valid_def)

schematic_goal brent_code[code]:
  fx0.brent f x = ?code
unfolding fx0.brent_def fx0.find_lambda_def fx0.find_mu_def fcomp_assoc[symmetric] fcomp_comp
by (rule refl)

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.