Proof and Equality

Christopher R. Genovese

Department of Statistics & Data Science

Thu 11 Sep 2025
Session #6

Plan

Recap: Propositions as Types

Plan

Recap: Propositions as Types

Computable Proofs

Plan

Plan

Recap: Propositions as Types

Computable Proofs

Even-Odd Revisited

Binary Search Revisited

Announcements

- Office Hours: me Tuesday 4pm, Thea Thursday 2pm, and by appointment. I encourage you to ask questions.
- **Reading**:
  - Finish previous readings
- **Homework**:
  - **migit** assignment Tasks #1–#6 due Tue 16 Sep. Available on github problem bank.
  - **migit** assignment Tasks #1–#10 due Tue 23 Sep. Available on github problem bank.

Goals for Today

Last time, we looked at testing and considered the fold pattern.

Today's goals:

1. Continue our exploration of proof in code with binary search and even-odd.

2. Big idea: an algorithm that is more than what it appears.

3. Big idea: expressing proof and verifying algorithms through code
   The development is based on the paper Dinges and Hinze (2025).

Plan

Recap: Propositions as Types

Computable Proofs

Even-Odd Revisited

Binary Search Revisited

Proposition as Types

We take a *proposition* as an assertion of a type and *proof* of the proposition as a value of that type.

A type that has at least one value is **inhabited**; one without a value is **uninhabited**. For example, `Void` is uninhabited.

Proposition as Types

We take a *proposition* as an assertion of a type and *proof* of the proposition as a value of that type.

A type that has at least one value is **inhabited**; one without a value is **uninhabited**. For example, `Void` is uninhabited.

We can formalize this:

```
-- For Refuted t to be inhabited, type t must be uninhabited
Refuted : Type -> Type
Refuted t = t -> Void

trait Uninhabited : Type -> Type where
    uninhabited : Refuted t

-- An absurd assumption can discharge a proof obligation
absurd : Uninhabited t => t -> a
```

Proposition as Types (cont'd)

If a and b are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

Proposition as Types (cont'd)

If a and b are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

- The proof of $a \wedge b$ involves the construction of a value of type `(a, b)`.

  To construct a value of this tuple type, we need to construct a value `x : a` and a value `y : b` and package them in a tuple `(x, y)`.

Proposition as Types (cont'd)

If a and b are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

- The proof of $a \vee b$ involves the construction of a value of type **`Either a b`**.

  To construct a value of type `Either a b`, we need to construct *either*

  - a value `x : a` and package it as `Left x`, or
  - a value `y : b` and package it as `Right b`.

Proposition as Types (cont'd)

If *a* and *b* are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

- The proof of $a \implies b$ involves constructing a *function* of type `a -> b`.

  Such a function, when given a value `x : a`, will produce a value of type `b`.

  Note that if `a` is uninhabited there is exactly one function `a -> b`. This fits the logical definition of implication.

Proposition as Types (cont'd)

If a and b are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

- The proof of $!\,a$ involves constructing a value of type `Refuted a`, i.e., a function `a -> Void`.

  Why?

Proposition as Types (cont'd)

If a and b are propositions represented by types `a` and `b`, "Propositions as types" tells us that:

- The proof of $!\,a$ involves constructing a value of type `Refuted a`, i.e., a function `a -> Void`.

  Why?

  We are showing that we cannot construct a value of type `a`.
  Recall from above:

  ```
  -- For Refuted t to be inhabited, type t must be uninhabited
  Refuted : Type -> Type
  Refuted t = t -> Void
  ```

Proposition as Types (cont'd)

"Propositions as types" is a mapping between logic and computation. This allows us to express the entire logical edifice with types and code.

Note, however, that our proofs here are all ***constructive***. Our proofs involve actually building the representative values.

The law of the excluded middle ($p \vee\, !\, p$) is not used – or often allowed – in this framework.

This idea gives a type theoretic foundation to all of mathematics that is distinct from traditional set theory.

Plan

Recap: Propositions as Types

Computable Proofs

Even-Odd Revisited

Binary Search Revisited

Proposition as Types: Equality

"Boolean blindness" and its implications.

```
data (===) : a -> b -> Type where
  Refl : x === x              -- x is an implicit argument
```

`Refl` is short for *reflexive*.

Notice that the === can only be constructed in the case where the same object is on both sides of the operator.

At our imaginary repl, we can see this:

```
> the ("Foobar" === "Foobar") Refl
Refl : "Foobar" === "Foobar"
> the (True === True) Refl
Refl : True === True
> the (4 === 7) Refl
Type error
```

Proposition as Types: Equality

For an example of how we use this, we will consider the following Peano representation of `Nat` (or `Natural` if we like) numbers:

```
data Nat : Type where
  Zero : Nat              -- represents 0
  Succ : Nat -> Nat       -- Succ n represents 1 + n
```

We express **theorems as functions** that type check. Basic examples:

```
-- Symmetry
sym : a === b -> b === a
sym Refl = Refl

-- Transitivity
trans : a === b -> b === c -> a === c
trans Refl Refl = Refl

-- Congruence
cong : {f : a -> b} -> a === b -> f a === f b
cong {f} Refl = Refl
```

Proposition as Types: Equality

For an example of how we use this, we will consider the following Peano representation of `Nat` (or `Natural` if we like) numbers:

```
data Nat : Type where
  Zero : Nat            -- represents 0
  Succ : Nat -> Nat     -- Succ n represents 1 + n
```

Theorems we could prove:

```
plusCommutative : (left : Nat) -> (right : Nat) -> left + right === right + left
plusZeroLeftNeutral : (right : Nat) -> 0 + right === right
plusZeroRightNeutral : (left : Nat) -> left + 0 === left
plusSuccRightSucc :  (left : Nat)
                  -> (right : Nat)
                  -> Succ (left + right) === left + Succ right
plusAssociative :  (left : Nat)
                -> (centre : Nat)
                -> (right : Nat)
                -> left + (centre + right) === (left + centre) + right
plusConstantRight :  (left : Nat)
                  -> (right : Nat)
                  -> (c : Nat)
                  -> left === right -> left + c === right + c
```

Example Proof: Commutativity

```
-- Background definitions

data Nat : Type where
    Zero : Nat
    Succ : Nat -> Nat

plus : Nat -> Nat -> Nat
plus Zero m = m
plus (Succ k) m = Succ (plus k m)


  -- These lemmas are inferred implicitly from the definition of plus
  -- so are not really needed. Listed here for concreteness, they
  -- could be used with trans in plusCommutative below.
plusByDef0 : plus Zero m === m
plusByDef0 = Refl

plusByDef1 : plus (Succ k) m === Suc (plus k m)
plusByDef1 = Refl
```

Example Proof: Commutativity

```
  -- Base case: plus Zero Zero === Zero becomes Zero === Zero
  -- Inductive step: plus k Zero === k => Succ (plus k Zero) === Succ k
plusZeroRight : (k : Nat) -> plus k Zero === k
plusZeroRight Zero = Refl
plusZeroRight (Succ k) = cong Succ (plusZeroRight k)
```

Example Proof: Commutativity

```
plusZeroRight : (k : Nat) -> plus k Zero === k
plusZeroRight Zero = Refl
plusZeroRight (Succ k) = cong Succ (plusZeroRight k)

  -- Base case: Succ (plus Zero n) === Succ n implicitly by plusByDef0
  -- Inductive step:
  --   Succ (plus k n) === plus k (Succ n)
  --   => Succ (Succ (plus k n)) === Succ (plus k (Succ n)) by cong
  --   => Succ (plus (Succ k) n) === plus (Succ k) (Succ n) by plusByDef1 implicitly
plusSuccRight : (m : Nat) -> (n : Nat) -> Succ (plus m n) === plus m (Succ n)
plusSuccRight Zero n = Refl
plusSuccRight (Succ k) n = cong Succ (plusSuccRight k n)
```

Example Proof: Commutativity

```
plusZeroRight : (k : Nat) -> plus k Zero === k
plusZeroRight Zero = Refl
plusZeroRight (Succ k) = cong Succ (plusZeroRight k)

plusSuccRight : (m : Nat) -> (n : Nat) -> Succ (plus m n) === plus m (Succ n)
plusSuccRight Zero n = Refl
plusSuccRight (Succ k) n = cong Succ (plusSuccRight k n)

plusCommutative : (n : Nat) -> (m : Nat) -> plus n m === plus m n

  -- Base case: plus Zero m === m === plus m Zero
plusCommutative Zero m = sym (plusZeroRight m)

  -- Inductive step:
plusCommutative (Succ k) m =
    let ih = plusCommutative k m          --: plus k m === plus m k
        succ_ih = cong Succ ih            --: Succ (plus k m) === Succ (plus m k)
        right_succ = plusSuccRight m k    --: Succ (plus m k) === plus  m (Succ k)
    in
      -- Implicitly: trans (trans plusByDef1 succ_ih) right_succ
      trans succ_ih right_succ
```

Example Proof: Commutativity

```
plusZeroRight : (k : Nat) -> plus k Zero === k
plusZeroRight Zero = Refl
plusZeroRight (Succ k) = cong Succ (plusZeroRight k)

plusSuccRight : (m : Nat) -> (n : Nat) -> Succ (plus m n) === plus m (Succ n)
plusSuccRight Zero n = Refl
plusSuccRight (Succ k) n = cong Succ (plusSuccRight k n)

plusCommutative : (n : Nat) -> (m : Nat) -> plus n m === plus m n
plusCommutative Zero m = sym (plusZeroRight m)
plusCommutative (Succ k) m =
    let ih = plusCommutative k m        --: plus k m === plus m k
        succ_ih = cong Succ ih          --: Succ (plus k m) === Succ (plus m k)
        right_succ = plusSuccRight m k  --: Succ (plus m k) === plus  m (Succ k)
    in
      trans succ_ih right_succ
```

Aside: Predicates

We usually define a **predicate** as a Boolean function (type `a -> Bool`). For example,

```
isEven : Int -> Bool
isEven n = n `mod` 2 == 0

isEmpty : List a -> Bool
isEmpty [] = True
isEmpty _  = False

leafless : BinaryTree a -> Bool
leafless EmptyTree = True
leafless (Branch _ _ _) = False
```

This is useful, but Booleans are limited. “Boolean blindness”

With rich types, we can generalize the idea of a predicate to a function `a -> Type`.

When the result type is uninhabited, the predicate fails, analogous to a `False` value from an ordinary predicate; otherwise, the predicate holds.

One advantage of this is that the returned type can contain additional information that we can use, e.g., a *proof* of some desirable property.

Proposition as Types: Testing Equality

For two values, we have the Boolean method == for testing equality at runtime. Can we make this more useful? Can it operate at "compile time"?

Consider:

```
naturalEq : (m : Natural) -> (n : Natural) -> Maybe (m === n)
naturalEq Zero Zero = Some Refl
naturalEq Zero (Succ k) = None
naturalEq (Succ j) Zero = None
naturalEq (Succ j) (Succ k) = case naturalEq j k of
                                None -> None
                                Some proof -> Some (cong Succ proof)
```

when this returns we get not just confirmation of equality, but proof as well.

We will see more uses of this idea later.

Proposition as Types: Contracts

Consider a type that witnesses that a value is an element of a vector (type `Vec len a`):

```
data Elem : a -> Vec k a -> Type where
  Here : Elem target (target :: tail)
  There : (later : Elem target tail) -> Elem target (y :: tail)
```

Either `x` is the first element of the vector, or *if you know that x occurs in the tail xs*, you know that `x` occurs in *any* vector with the same tail.

We can use this to write functions where the proof represents a contract that the implementation can exploit. For instance:

```
remove :  (target : a)
       -> (xs : Vec (Succ n) a)
       -> (proof : Elem target xs)
       -> Vec n a
```

Decidable Propositions

```
-- Decidable prop represents a propositioon that
-- can either be proved or disproved
data Decidable : (prop : Type) -> Type where
  Proved : (proof : prop) -> Decidable prop
  Disproved : (contra : Refuted prop) -> Decidable prop
```

Decidable Propositions

```
-- Decidable prop represents a propositioon that
-- can either be proved or disproved
data Decidable : (prop : Type) -> Type where
  Proved : (proof : prop) -> Decidable prop
  Disproved : (contra : Refuted prop) -> Decidable prop
```

The short-hand data declaration makes salient the two possibilities that we pattern match on:

```
data Decidable prop = Proved prop | Disproved (Refuted prop)
```

Decidable Propositions

```
-- Decidable prop represents a propositioon that
-- can either be proved or disproved
data Decidable : (prop : Type) -> Type where
  Proved : (proof : prop) -> Decidable prop
  Disproved : (contra : Refuted prop) -> Decidable prop
```

The short-hand data declaration makes salient the two possibilities that we pattern match on:

```
data Decidable prop = Proved prop | Disproved (Refuted prop)
```

How do these two signatures for `naturalEq` differ and what do those differences mean?

```
naturalEq : (m : Natural) -> (n : Natural) -> Maybe (m === n)

naturalEq : (m : Natural) -> (n : Natural) -> Decidable (m === n)
```

Aside: Decidable Equality

The first form works but moves the resolution to a runtype check of a `Maybe` value. The second form gives us a *proof* of either possibility.

```
naturalEq : (m : Natural) -> (n : Natural) -> Decidable (m === n)
naturalEq Zero Zero = Proved Refl
naturalEq Zero (Succ k) = Disproved zeroNEsucc
naturalEq (Succ j) Zero = Disproved succNEzero
naturalEq (Succ j) (Succ k) =
        case naturalEq j k of
          Proved proof -> Proved (cong Succ proof)
          Disproved contra -> Disproved (noRecurse contra)

  where noRecurse : (contra : Refuted (k === j))
                  -> (Succ k === Succ j)
                  -> Void
        noRecurse contra Refl = contra Refl     -- contradiction

        zeroNEsucc : Refuted (Zero === Succ k)
        zeroNEsucc Refl = impossible            -- impossible is a keyword

        succNEzero : Refuted (Succ k === Zero)
        succNEzero Refl = impossible
```

Aside: Decidable Equality

We can build infrastructure for many types to have this kind of ***decidable equality***.

For instance:

```
trait DecEq t where
  decEq : (x : t) -> (y : t) -> Decidable (x === y)

implements DecEq Natural where
  decEq = naturalEq
```

Plan

Recap: Propositions as Types

Computable Proofs

Even-Odd Revisited

Binary Search Revisited

Linear Search as a Proposition

A few background items:

```
data Even : Nat -> Type where
  ZeroEven : Even Zero
  SuccEven : Even n -> Even (Succ (Succ n))

data Odd : Nat -> Type where
  SuccOdd  : Even n -> Odd  (Succ n)

parity : (n : Nat) -> Either (Odd n) (Even n)
parity 0 = Right ZeroEven
parity 1 = Left (SuccOdd ZeroEven)
parity (2 + n) = parity n
```

Explain these?

Linear Search as a Proposition

A type that represents an order on the natural numbers:

```
-- Less m n means that m < n
data Less : Nat -> Nat -> Type where
  OneL : (n : Nat) -> Less n (1 + n)
  SucL : Less (1 + m) n -> Less m n

LessEq : Nat -> Nat -> Type
LessEq m n = Either (Less m n) (m === n)
```

How would you prove these theorems?

```
lessTrans : Less k m -> Less m n -> Less k n
lessEqTrans : LessEq k m -> LessEq m n -> LessEq k n
```

How would we show that `Less 0 0` is uninhabited?

Linear Search as a Proposition

Our theorem:

```
evenOddPairL :  {ell : Nat}
             -> {r : Nat}
             -> Less ell r
             -> (box : Nat -> Nat)
             -> (Even (box ell), Odd (box r))
             -> (i : Nat ## (Even (box i), Odd (box (1 + i))))
```

Here, think of `Less` as a type assertion that $\ell < r$, and `box` represents the contents of the boxes, which we take to span all `Nat`s. (What would the type of `box` be if we had a smaller, finite number of boxes?)

`(_ ## _)` denotes a *dependent pair*, which pairs a value in the first component with a value in the second component whose *type depends on the value of the first component*. From a logical standpoint, we can read that return type as

$$\exists i \in \mathbb{N} \text{ such that } \mathrm{Even}(\mathrm{box}(i)) \wedge \mathrm{Odd}(\mathrm{box}(1+i))$$

What does this proposition `evenOddPairL` mean?

Linear Search as a Proposition: A Proof

```
data Less : Nat -> Nat -> Type where
  OneL : (n : Nat) -> Less n (1 + n)
  SucL : Less (1 + m) n -> Less m n

evenOddPairL :  {ell : Nat}
             -> {r : Nat}
             -> Less ell r
             -> (box : Nat -> Nat)
             -> (Even (box ell), Odd (box r))
             -> (i : Nat ## (Even (box i), Odd (box (1 + i))))

evenOddPairL (OneL n) box (eproof, oproof) = (n ## (eproof, oproof))
evenOddPairL {m} {n} (SucL mplus1LTn) box (eproof, oproof) =
    case parity (box (1 + m)) of
      Left oproof' -> (m ## (eproof, oproof'))
      Right eproof' -> evenOddPairL mplus1LTn box (eproof', oproof)
```

Let's see how this proof is just what we gave in Week 1.

Beyond Linear Search

If we change the ordering on the natural numbers, we get a new algorithm.

```
  -- floor(n/2)
floorDiv2 : Nat -> Nat
floorDiv2 0 = 0
floorDiv2 1 = 0
floorDiv2 (Succ (Succ k)) = 1 + floorDiv2 k

  -- Midpoint rounded down, floor((m + n)/2)
mid : Nat -> Nat -> Nat
mid m n = floorDiv2 (m + n)

  -- MidLT m n means that m <= mid m n < n
data MidLT : Nat -> Nat -> Type where
  Single : (n : Nat) -> MidLT n (1 + n)
  Split  : MidLT m (mid m n) -> MidLT (mid m n) n -> MidLT m n
```

How does this differ from the relation `Less`? Does `Less m n` imply `MidLT m n`? Vice versa? How does our proof change?

Beyond Linear Search (cont'd)

The revised proof:

```
evenOddPairM (Single n) box (eproof, oproof) = (n ## (eproof, oproof))
evenOddPairM {m} {n} (Split mk kr) box (eproof, oproof) =
    case parity (box (mid m n)) of
      Left oproof' -> evenOddPairM m (mid m n) mk box (eproof, oproof')
      Right eproof' -> evenOddPairM (mid m n) r kr box (eproof', oproof)
```

What does this mean? Let's state it in words.

How does this proof use the properties of `mid`? Does it at all?

Generalization 1

The two orders we have used before, `Less` and `MidLT`, correspond to *search strategies*. Let's generalize this:

```
data IntervalTree : Nat -> Nat -> Type where
  Leaf   : n -> IntervalTree n (1 + n)
  Branch : m -> IntervalTree k m -> IntervalTree m r -> IntervalTree k r
```

We can think of an `IntervalTree m n` as describing a tree that subdivides $[m\,..\,n)$, the half-open increment.

We can define mappings from both of our strategies to this one:

```
sequential : Less ell r -> IntervalTree ell r
binarySubdivision : MidLT ell r -> IntervalTree ell r
```

and in fact, we can map the other way, e.g.,

```
treeLess : IntervalTree ell r -> Less ell r
balanced : Less ell r -> MidLT ell r
```

With `IntervalTree`, we can simplify and generalize our proof for any such strategy.

Generalization 1

```
data IntervalTree : Nat -> Nat -> Type where
  Leaf   : n -> IntervalTree n (1 + n)
  Branch : m -> IntervalTree k m -> IntervalTree m r -> IntervalTree k r

evenOddPairI :  IntervalTree ell r
             -> (box : Nat -> Nat)
             -> (Even (box ell), Odd (box r))
             -> (i : Nat ## (Even (box i), Odd (box (1 + i))))

evenOddPairI (Leaf n) box (eproof, oproof) = (n ## (eproof, oproof))
evenOddPairI (Branch m left right) box (eproof, oproof) =
    case parity (box m) of
      Left oproof' -> evenOddPairI left box (eproof, oproof')
      Right eproof' -> evenOddPairI right box (eproof', oproof)
```

The Leaf case means success; the Branch case means we make a decision on one side or the other.

Note that the `(Even (box ell), Odd (box r))` here expresses the ***functional invariant*** that we discussed before.

Generalization 2

We have derived a solution to the Even-Odd problem that abstracts away from the search strategy. Can we also abstract away from even-ness/odd-ness as the criteria?

Consider two predicates $P, Q \colon \mathbb{N} \longrightarrow \texttt{Type}$ that we can call *perky* and *quirky*.

Taken together, we want every natural number to be either perky or quirky, but we do not insist on exclusivity. Instead, we have an oracle

```
oracle : (n : Nat) -> Either (P n) (Q n)
```

that tells us how to classify any given number.

```
intervalSearch :  IntervalTree ell r
               -> ((n : Nat) -> Either (P n) (Q n))
               -> (P ell, Q r)
               -> (i : Nat ## (P i, Q (i + 1)))

intervalSearch (Leaf n) oracle (perky, quirky) = n ## (perky, quirky)
intervalSearch (Branch m left right) oracle (perky, quirky)
  case oracle m of
      Left perky'   -> intervalSearch right oracle (perky', quirky)
      Right quirky' -> intervalSearch left  oracle (perky, quirky')

evenOddPair strategy box = intervalSearch strategy (\n -> parity (box n))
```

Plan

Recap: Propositions as Types

Computable Proofs

Even-Odd Revisited

Binary Search Revisited

Deriving Binary Search from Even-Odd

Now, we can express binary search directly:

```
binarySearch :  Less ell r
             -> ((n : Nat) -> Either (P n) (Q n))
             -> (P ell, Q r)
             -> (i : Nat ## (P i, Q i))

binarySearch lt = intervalSearch (binarySubdivision (balanced lt))
```

where

```
balanced : Less ell r -> MidLT ell r
binarySubdivision : MidLT ell r -> IntervalTree ell r
```

and for instance

```
binarySubdivision (Single n) = Leaf n
binarySubdivision (Split left right) = Branch {_} (binarySubdivision left)
```

where `{_}` is an *implicit argument* determined from the types of `left` and `right`, `MidLT m {_}` and `MidLT {_} n`.

Notice that this does not require ordered tables. Rather than taking a "negative" view of ruling out regions, we are taking a "positive" view of ruling in the regions that

Examples and Applications: Guessing Game

In the "guess a number" game, I choose an at most three-digit number (say), and you try to guess it. After each guess, I tell you whether the true number is less then, equal to, or greater than your guess.

Choose predicates `P n` and `Q n`, where `P n i = LessEq i n` and `Q n i = Less n i` and `n` represents the true number.

```
numberGuess : (n : Nat) -> Less n 1000 -> (i : Nat ## (P n i, Q n i))
numberGuess n valid = binarySearch (less 0 999) (compare n) (lessEq 0 n, valid)
  where
    less : (a : Nat) -> (b : Nat) -> Less a (1 + a + b)
    less a Zero = OneL a
    less a m = iterate m SucL (OneL (a + m))

    lessEq : (a : Nat) -> (b : Nat) -> LessEq a (a + b)
    lessEq a Zero = Right $ sym $ plusZeroRightNeutral a
    lessEq a (Succ k) = Left $ iterate k SucL (OneL (a + k))

compare : (m : Nat) -> (k : Nat) -> Either (LessEq k m) (Less m k)
compare m Zero = Left $ lessEq 0 m
compare m k = if k <= m
                then Left $ lessEq k (m `monus` k)
                else Right $ less n (k `monus` m)
```

What does the returned value (post-condition) tell you here??

Examples and Applications: Table Lookup

We can extend this to a search in a table. Let table: $\mathbb{N} \longrightarrow \mathbb{N}$ represent a table of values, keyed by the input, e.g., $\langle k \rangle \mapsto k^2$. We want to search for a specified key in the table.

```
tableLookup :  Less ell r
            -> (table : Nat -> Nat)
            -> (key : Nat)
            -> (LessEq (table ell) key, Less key (table r))
            -> (i : Nat ## (LessEq (table i) key, Less key (table (i + 1)))

tableLookup ell_LT_r table key = binarySearch ell_LT_r (\i -> compare key (
```

This search always succeeds, returning an index i such that $t_i \preceq \text{key} \prec t_{i+1}$.

(What do we get, for instance, when table $= \langle k \rangle \mapsto k^2$?)

But note: **we do not need an assumption that table is monotone**!

We do need to prove `(LessEq (table ell) key, Less key (table r))`. This is the invariant that is maintained.

We've replaced monotonicity with a general invariant!

Examples and Applications: Local Maxima

Here's a puzzle from Erickson (2011) as adapted by Dinges and Hinze (2025).

> *You and several opponents are each given a sequence of n boxes, with each box containing a natural number. It costs \$100 to open a box, and your goal is to find a box whose number is no smaller than the numbers in its neighboring boxes. (Imagine virtual boxes containing 0 on either end of the row.)*
>
> *The player who spends the least money to find their boxes wins the game.*

0	4	2	7	6	5	3	9	8	1
1	2	3	4	5	6	7	8	9	A

Can you use binary search to solve this problem? If so, what are P and Q?

Examples and Applications: Local Maxima

Here's a puzzle from Erickson (2011) as adapted by Dinges and Hinze (2025).

> *You and several opponents are each given a sequence of n boxes, with each box containing a natural number. It costs \$100 to open a box, and your goal is to find a box whose number is no smaller than the numbers in its neighboring boxes. (Imagine virtual boxes containing 0 on either end of the row.)*
>
> *The player who spends the least money to find their boxes wins the game.*

0	4	2	7	6	5	3	9	8	1
1	2	3	4	5	6	7	8	9	A

Consider making $P\ i$ hold if the slope at box i $((\text{box}(1+i) - \text{box}(i))/(1+i-i))$ is non-negative and $Q\ i$ hold if the slope at box i is non-positive. (Note that P and Q are not exclusive here.)

We can now use `binarySearch` but instead of using `compare` as an oracle, we use a similar function with return type
`Either (LessEq (box i) (box i + 1)) (LessEq (box i + 1) (box i))`.

This shows both that binary search is much more general than it seemed and that we can get strong guarantees out of the algorithm when we take a "positive" view.

THE END