The Search for Proof

Christopher R. Genovese

Department of Statistics & Data Science

28 Aug 2025
Session #2

Plan

Binary Search

Plan

Binary Search

Recap: The Even-Odd Problem

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

Plan

Plan

Announcements

- Please fill out **office hours poll**. (Treat dates as generic days of week.) https://www.when2meet.com/?31878095-c7DkK
- Office Hours will be posted shortly, likely Th 2pm.
- Email subject: [750]
- Github invitations, assignment repos, clean-start, and new-homework
- **Please bring your laptop to every class**
- **Reading**:
  - Finish Thinking Languages Part 1 if not already
  - System Setup `https://36-750.github.io/course-info/system-setup/`
  - Interlude F Chapter 18 Section 1, material on monoids
- **Homework**:
  - **swag** assignment due Tue 9 Sep. Available on Canvas and github problem bank.

Questions?

Goals

Today and next class we will tell a story.

- Starting from the Even-Odd Problem, we will derive a general (and "positive") concept of search.
- We will see how we can express logic and proofs through types.
- We will see how changing our partial order on the naturals gives us different algorithms.
- From this, we will eventually generalize the famous problem of binary search and see a range of applications beyond searching in *ordered tables*.

We'll get through part of this story, with some time to digest the punch line next time.

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

Even-Odd Revisited

Binary Search Revisited

Binary Search

Binary search is a classical problem for efficiently searching for data in an "ordered table". It is a notoriously difficult algorithm to get right.

There are two traditional formulations:

(Knuth) Given a sorted sequence of n "keys" $k_1 \prec k_2 \prec \cdots \prec k_n$ and an arbitrary key k, we would like to find an index $i \in [1\,..\,n]$ for which $k \equiv k_i$.

The basic approach is to pick an arbitrary index $1 \leq m \leq n$ and compare k to k_m, yielding three possible outcomes:

1. $k < k_m$, so we eliminate keys $k_m \prec \cdots \prec k_n$ from consideration,
2. $k \equiv k_m$, so we have succeeded with $i = m$, or
3. $k > k_m$, so we eliminate keys $k_1 \prec \cdots \prec k_m$ from consideration,

If we iterate this procedure and pick $m = \lfloor (1 + n)/2 \rfloor$, the problem will be solved in logarithmic time.

Binary Search

Binary search is a classical problem for efficiently searching for data in an "ordered table". It is a notoriously difficult algorithm to get right.

There are two traditional formulations:

(Bird and Wadler) Given $\ell, r \in \mathbb{N}$ and a predicate P on $\mathbb{N}$, we want to find an $i \in [\ell \mathinner{..} r]$ such that $P\,i$ holds. Assume that P is monotone: $a \leq b$ and $P\,a$ implies $P\,b$.

The basic approach is similar: pick an arbitrary $m \in [\ell \mathinner{..} r]$ and check $P\,m$, yielding two outcomes:

1. $P\,m$ holds, so we eliminate $1 + m, \ldots, r$ from consideration, or
2. $P\,m$ does not hold, so we eliminate $1, \ldots, m$ from consideration.

Iterating and picking $m = \lfloor (\ell + r)/2 \rfloor$ again gives a logarithmic runtime.

Binary Search

Binary search is a classical problem for efficiently searching for data in an “ordered table”. It is a notoriously difficult algorithm to get right.

There are two traditional formulations:

(Bird and Wadler) Given $\ell, r \in \mathbb{N}$ and a predicate P on $\mathbb{N}$, we want to find an $i \in [\ell \mathinner{.\,.} r]$ such that $P\,i$ holds. Assume that P is monotone: $a \leq b$ and $P\,a$ implies $P\,b$.

The basic approach is similar: pick an arbitrary $m \in [\ell \mathinner{.\,.} r]$ and check $P\,m$, yielding two outcomes:

1. $P\,m$ holds, so we eliminate $1 + m, \ldots, r$ from consideration, or
2. $P\,m$ does not hold, so we eliminate $1, \ldots, m$ from consideration.

Iterating and picking $m = \lfloor(\ell + r)/2\rfloor$ again gives a logarithmic runtime.

Write an (rough) implementation of binary search. You can use either formulation but I will offer a third option as well. What is the type signature of the main function? What helper functions do you need?

Plan

Back to the Even-Odd Problem

In front of you are $n + 2$ boxes, each of which contains a natural number. All the boxes except the first and last are closed, so you can only see the first and last numbers.

4	?	?	$\cdots$	?	711
0	1	2	$\cdots$	n	$1 + n$

You want to find an *even-odd pair*, two adjacent boxes such that the first contains an even number and the second an odd number. You would like to do this by opening as few boxes as possible.

Back to the Even-Odd Problem

We concluded that if we see an even number in the first box and an odd in the last box, we can *prove* that an even-odd pair exists.

Two preconditions: $n + 2 \geq 2$ boxes and an even in the first box and odd in the last.

The proof is by induction, but we had to take care of how we structured the induction hypothesis (in particular, the order we use).

Base case: $n = 0$, we have an even-odd pair by preconditions.

Induction case: open box 1. If it's even, we apply the induction hypothesis to boxes $1..2 + n$. If it's odd, we have found an even-odd pair.

Moreover, this proof suggested an algorithm for finding it: open boxes one at a time from 1 until we find an odd. This is **linear search**.

Back to the Even-Odd Problem: Invariants

One interesting feature of this algorithm/proposition: we maintain a condition at every step. This is called an **invariant**.

The invariant is that *the leftmost box is Even and the rightmost box is Odd*.

Invariants have three phases in their life cycle:

- Initialization. This is the responsibility of whoever calls/invokes the algorithm.
- Lifetime. This is the responsibility of the algorithm. At each stage, the invariant is maintained. Here, when we open a box, we proceed with a set of boxes that satisfy the invariant.
- Fulfillment. This is the responsibility of the algorithm. Here, the invariant implies the fulfillment of the goal. In our case, when the two end boxes are adjacent.

Back to the Even-Odd Problem: Next Steps

We have three goals next for this problem:

1. Write code that provides proof of its own success.
2. Consider if different order relations can lead to different (and better performing) algorithms.
3. Generalize this to find a novel approach to a common, well-studied problem.

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

Even-Odd Revisited

Binary Search Revisited

Destructuring Types

Following up on the example from last time, start with a simpler example.

```
data RGB = Red | Green | Blue
```

Consider the function

```
hexColor : RGB -> String
hexColor Red = "#ff0000"
hexColor Green = "#00ff00"
hexColor Blue = "#0000ff"
```

The equations defining `show` use **pattern matching**.

The patterns are collectively exhaustive because `Red`, `Green`, and `Blue` are the *only ways* to construct a value of type `RGB`.

Pattern matching is a shorthand for what would be mathematically:

$$\texttt{hexColors}(x) = \begin{cases} \text{"}\#\texttt{ff0000}\text{"} & \text{if } x = \texttt{Red} \\ \text{"}\#\texttt{00ff00}\text{"} & \text{if } x = \texttt{Green} \\ \text{"}\#\texttt{0000ff}\text{"} & \text{if } x = \texttt{Blue} \end{cases}$$

Destructuring Types

Let's see what this looks like in Python and R:

```python
from enum import StrEnum

class RGB(StrEnum):
    RED = 'Red'
    GREEN = 'Green'
    BLUE = 'Blue'

def hex_colors(rgb : RGB) -> str:
    match rgb:
        case RGB.RED:
            return "#ff0000"
        case RGB.GREEN:
            return "#00ff00"
        case RGB.BLUE:
            return "#0000ff"
        case _:
            raise ValueError('hex_colors given an invalid RGB value')
```

Destructuring Types

```r
# Can define RED, GREEN, BLUE in various ways

hexColors <- function(rgb) {
    if ( identical(rgb, RED) ) {
        return( "#ff0000" )
    } else if ( identical(rgb, GREEN) ) {
        return( "#00ff00" )
    } else if ( identical(rgb, BLUE) ) {
        return( "#0000ff" )
    } else {
        stop("hexColors given an invalid RGB value")
    }
}
```

Destructuring Types (cont'd)

Now back to `Maybe`, and note the implicit foralls (made explicit here):

```
data Maybe : Type -> Type where
  None : forall a. Maybe a
  Some : forall a. a -> Maybe a
```

Those are the only two ways to construct data of that type.

The value `None` has a type of the form `Maybe a` where `a` is inferred from the context. We can use the construct `the type obj` at an imaginary repl to see this

```
> the (Maybe String) None
> None : Maybe String
> the (Maybe Int) None
> None : Maybe Int
> the (Maybe Bool) None
> None : Maybe Bool
```

Destructuring Types (cont'd)

Now back to `Maybe`, and note the implicit foralls (made explicit here):

```
data Maybe : Type -> Type where
  None : forall a. Maybe a
  Some : forall a. a -> Maybe a
```

Those are the only two ways to construct data of that type.

It follows, that any value of type `Maybe String` looks like either `None` or `Some s` for a string value `s`.

```
getString : Maybe String -> String
getString None = "missing value here"
getString (Some s) = s
```

Destructuring Types (cont'd)

A similar and useful example is

```
data Either : Type -> Type -> Type where
  Left  : a -> Either a b
  Right : b -> Either a b
```

with shorthand

```
data Either a b = Left a | Right b
```

This type is analogous to a disjoint union. It is a type that can be a value of two other possible types.

In TL1 we might write

```
countOrLabel : Either Nat String
countOrLabel = Left 42

-- or

countOrLabel : Either Nat String
countOrLabel = Right "life, the universe, and everything"
```

Destructuring Types (cont'd)

A similar and useful example is

```
data Either : Type -> Type -> Type where
  Left  : a -> Either a b
  Right : b -> Either a b
```

with shorthand

```
data Either a b = Left a | Right b
```

This type is analogous to a disjoint union. It is a type that can be a value of two other possible types.

In R or Python, we would just write

```
countOrLabel <- 42

# or

countOrLabel <- "life, the universe, and everything"
```

but by attesting that `countOrLabel` has this type, we have a contract and must handle both cases.

Destructuring Types (cont'd)

We can use an alternative representation of the natural numbers: *the Peano representation*.

This works for both types `Natural` and `Nat`. For instance:

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

A natural number is either zero or one bigger than another natural number. This is the basis of induction!

We have, for instance:

```
1 is Succ Zero
2 is Succ (Succ Zero)
3 is Succ (Succ (Succ Zero))
...
```

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

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.

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

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.

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

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.

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

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.

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

- The proof of $a \Longrightarrow b$ 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

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.

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

- The proof of ! *a* involves showing that we cannot construct a value of type `a`.

  This involves constructing a function of type `a -> Void`.

  Why?

  We can capture this concept with ... a type:

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

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.

If *a* and *b* are propositions represented by types `a` and `b`, "Propositions as types" tells us that: This mapping between logic and types gives us the entire

logical edifice to work with.

Note, however, that our proofs here are all ***constructive***.

The law of the excluded middle ($p \vee\, !\, p$) is not used (or really allowed).

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 sids 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
```

Proposition as Types: Equality

For an example of how we use this, we will imagine the following Peano representation of Natural numbers (works for Nat too if we want):

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

Now, we define

```
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 -> Just (cong Succ proof)

-- This uses a congruence theorem:
cong : {f : a -> b} -> a === b -> f a === f b
```

Why do this? The inadequacy of booleans.

We will see many 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 a function where the proof represents a contract that the implementation can exploit:

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

Decidable and Impossible Propositions

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

-- 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

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

-- Use an absurd assumption to discharge a proof obligation
absurd : Uninhabited t => t -> a
```

Aside: Decidable Equality

`naturalEq` works but still moves the resolution to a runtype check of a `Maybe` value. We can give a stronger form:

```
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

        zeroNEsucc : Refuted (Succ k === Zero)
        zeroNEsucc 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

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

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

```
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.

`(_ ## _)` 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 proof standpoint, we can read that type as

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

What does this proposition `evenOddPairL` mean?

Linear Search as a Proposition: A Proof

```
-- 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)

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 yesterday.

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 (S (S 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?

Plan

Binary Search

Recap: The Even-Odd Problem

Recap: Data Types

Propositions as Types

Even-Odd Revisited

Binary Search Revisited

Next Time

Next time, we will see the end of this story:

- How Even-Odd Pairs generalizes and gives us general linear search.
- How we can derive a provably correct binary search for free.
- How this changes the contract of binary search from a "negative" to a "positive" view.
- How this reconceptualizes binary search away from *ordered tables* to a wide variety of applications.

The End