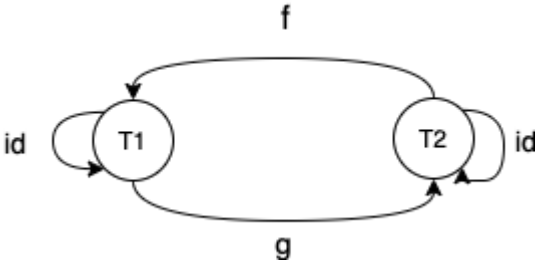
CTfP Ch5 Challenges

> Q1: Show that the terminal object is unique up to unique isomorphism.

Given 2 terminal objects T1/T2, the morphisms between them can be represented via the diagram below. By definition(being terminal), T1 and T2 can only have a single unique morphism from any other object(including itself). Since it belongs in a category the objects also have the identity morphism.



Since the objects are terminal, we know that if there are 2 functions(morphisms) from an object to a terminal object(even if the former is itself), both morphisms must be the same. Thus:

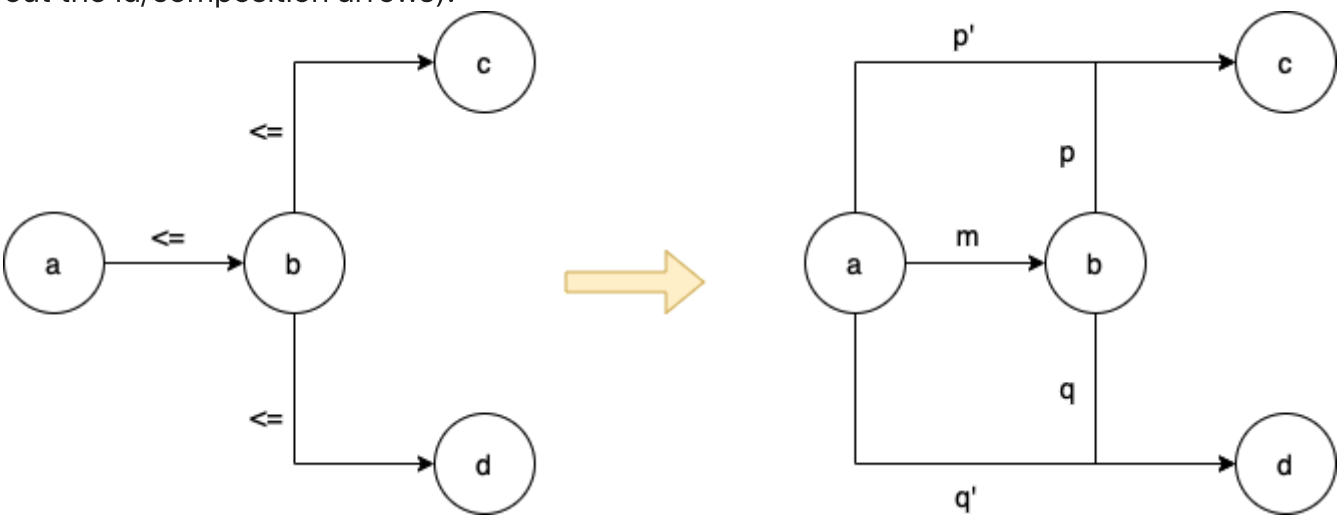
- `f.g` is equivalent to id_{T1}
- `g.f` is equivalent to id_{T2}

This shows that terminals are **unique up to isomorphism**. However, in addition it can also be observed that there exists only a single isoomorphism between any 2 terminal objects. Thus they are also **unique up to unique isomorphism**.

> Q2: What is a product of two objects in a poset?

Referred to this resource for next 2 questions.

The diagram on the left indicates how a the objects in a poset category would look like(without the id/composition arrows).



Now when we add in the morphisms corresponding to composition (eg. `p' ~= (a->b, b->c)`), the resulting figure on the right looks extremely similar to the diagram we used to formulate and reason about the universal construction for products involving pairs from the text.

If we wanted the product of (`c` , `d`): `m` factorizes(or is the common branch) `p/q` out of `p'/q'` , leaving `b` as the highest ranking candidate for the product. This satisfies:

- p' = p.m
- q' = q.m

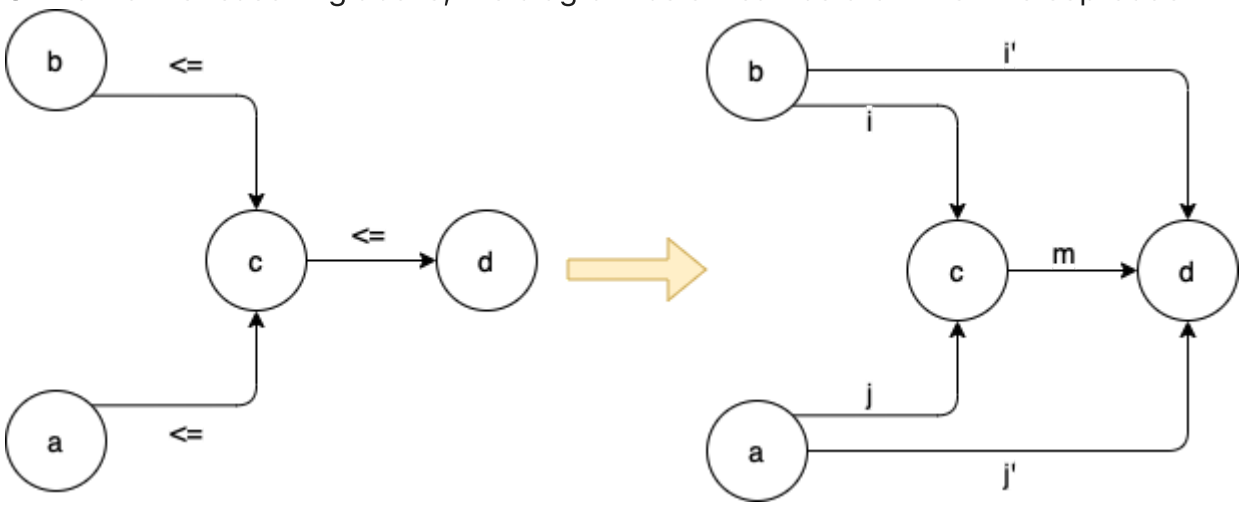
Additionally, `m` should be unique since this is a thin category.

As a counterexample, if we considered `a` , there is no morphism going from `b` backwards to even give as a factorizer(`m`).

So the product of `(c, d)` would be the largest object in the poset smaller than both the objects in consideration.

> Q3: What is the coproduct of two objects in a poset?

Similar to the reasoning above, the diagram below can be drawn for the coproduct:



Once again, this is both a thin category and one that has `m` being the unique morphism factorizing(acting as a common morphism) `i/j` out of `i'/j'` satisfying:

- j' = m . j
- i' = m . i

Finally, this results in the coproduct of `(a, b)` being the smallest number in the set larger than both of them(`c` here).

> Q4: Implement the equivalent of Haskell either as a generic type in your favorite language?

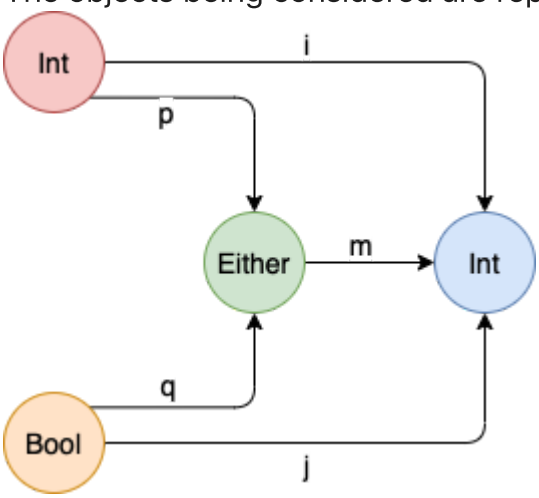
```
public class Either<T,U> {
  private Object theObject;
  private boolean theIsLeft;

  private Either(Object aObject, boolean aIsLeft) {
    theObject = aObject;
    theIsLeft = aIsLeft;
  }

  public static <T,U> Either<T,U> makeLeft(T aLeft) {
```

```
    return new Either<>(aLeft, true);
  }

    public static <T,U> Either<T,U> makeRight(U aRight) {
    return new Either<>(aRight, false);
  }

  @SuppressWarnings("unchecked")
  public T getLeft() {
    return (T) theObject;
  }

  @SuppressWarnings("unchecked")
  public U getRight() {
    return (U) theObject;
  }
}
```

> Q5: Show that Either is a “better” coproduct than int equipped with two injections:

```
int i(int n) { return n; }
int j(bool b) { return b ? 0: 1; }
```

The objects being considered are represented in the diagram below:



Considering the universal construction for coproducts, we can rank Either as higher if there is a way to generate a unique morphism `m` from `Either -> Int` with the `p/q` projections shown in the diagram.

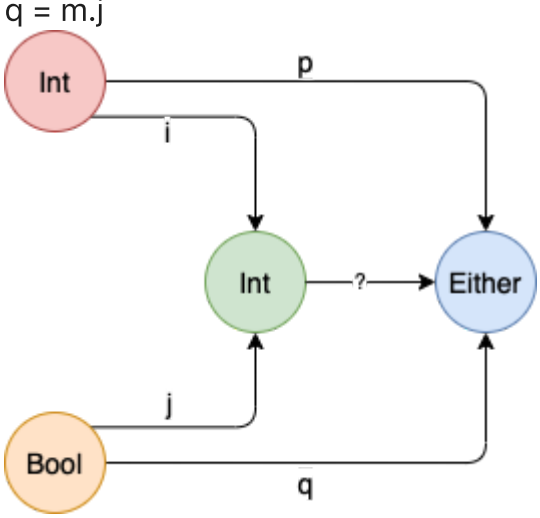
```
-- https://onecompiler.com/haskell/3wq4tvc6q
-- Given
i :: Int -> Int
i = id

j :: Bool -> Int
j True = 0
j False = 1

p :: Int -> Either Int Bool
```

```
p = Left

q :: Bool -> Either Int Bool
q = Right

-- Find the appropriate 'm'
m :: Either Int Bool -> Int
m (Left n) = n
m (Right True) = 0
m (Right False) = 1

{--
Need to prove that m satisfies:
 i = m.p
 j = m.q
--}

main = do
  print $ (i 100) == (m . p) 100
  print $ (j True) == (m . q) True
  print $ (j False) == (m . q) False
```

> Q6: How would you argue that Int with the two injections i and j cannot be “better” than Either?

We would not be able to not construct any morphism `m` from `Int -> Either` with the projections `i/j`, without losing information and thus breaking the requirement:

- p = m.i
- q = m.j




```
-- https://onecompiler.com/haskell/3wq59x7ck
-- m1 works with 'i' but loses information for 'j'
m1 :: Int -> Either Int Bool
m1 x = Left x

-- m2 works with 'j' but loses information for 'i'
m2 :: Int -> Either Int Bool
m2 x
  | x == 0    = Right True
```

```
  | otherwise = Right False

main = do
  print $ (p 100) == (m1 . i) 100
  print $ (q True) /= (m1 . j) True
  print $ (p 100) /= (m2 . i) 100
  print $ (q True) == (m2 . j) True
```

$m_{\_1}$ satisfies the requirements for `p = m.i` but fails that for `q = m.j`. The opposite is true of $m_{\_2}$. There's no way of preserving information to satisfy both requirements with the factorizing morphism `m`.

> Q7: What about these injections?

```
int i(int n) {
  if (n < 0) return n;
  return n + 2;
}
int j(bool b) { return b ? 0: 1; }
```

Note: My initial solution here was incorrect, fixed this after listening to the discussion/gleaning over other solutions.
Once again, you can come up with a unique morphisms `m` going from `Either -> Int` which satisfies the coproduct requirements.

```
m :: Either Int Bool -> Int
m (Left n)
  | n < 0 = n
  | otherwise = n + 2
m (Right True) = 0
m (Right False) = 1
```

On the flip side, it might also be possible to preserve information by the injections `i/j` when trying to come up with a `m` (m_rev below) from `Int -> Either`.

```
m_rev :: Int -> Either Int Bool
m_rev x -
- How to handle an input "x" that maps to "INT_MAX", because the input must be
"INT_MAX" + 2 which will overflow.
  | x == 0    = Right True
  | x == 1    = Right False
  | x < 0     = Left x
  | otherwise = Left (x - 2)
```

Since some languages support effectively infinite values of `int` (like python), I'm still not completely sure about the validity of this argument.

> Q8: Come up with an inferior candidate for a coproduct of int and bool that cannot be better than Either because it allows multiple acceptable morphisms from it to Either.

Taking inspiration from the writeup on product, an obvious candidate which is redundant is the `Triple` object(`data Triple a b c = First a | Second b | Third c`) with the injections:

- i x = First x
- j b = Third b

The code below shows the type declarations and demonstrates 2 possible m's showing that there isnt a unique morphism from `Triple -> Either`

```
-- https://onecompiler.com/haskell/3wq5bc55x
data Triple a b c = First a | Second b | Third c

i :: Int -> Triple Int Int Bool
i = First

j :: Bool -> Triple Int Int Bool
j = Third

p :: Int -> Either Int Bool
p = Left

q :: Bool -> Either Int Bool
q = Right

m1 :: Triple Int Int Bool -> Either Int Bool
m1 (First x) = Left x
m1 (Second x) = Left x
m1 (Third b) = Right b

m2 :: Triple Int Int Bool -> Either Int Bool
m2 (First x) = Left x
m2 (Second x) = Right (x > 0)
m2 (Third b) = Right b

{--
Need to prove that m satisfies:
 p = m.i
 q = m.j
--}

main = do
  -- Multiple m1/m2 satisfy the properties
  print $ (p 100) == (m1 . i) 100
  print $ (q True) == (m1 . j) True
  print $ (q False) == (m1 . j) False

  print $ (p 100) == (m2 . i) 100
  print $ (q True) == (m2 . j) True
  print $ (q False) == (m2 . j) False
```