# Problem Set 3

This third problem set explores binary relations, functions, and their properties. We've chosen these problems to help you get a sense for how to reason about these structures how to write proofs using formal mathematical definitions, and why all this matters in practice.

Before beginning this problem set, you may want to read over Handout 17, which talks about general patterns for proving statements expressed in first-order logic. Specifically, remember that

- if you're proving something about a definition specified in first-order logic, you should use the first-order logic to inform your proof setup, but
- you ***should not*** use first-order logic notation (quantifiers, connectives, etc.) in your proofs, which should be written in plain English along the lines of what you saw on Problem Set One and Problem Set Two.

We recommend that you take a look at the proofs from this week's lectures to get a sense of what this looks like. The proofs on cyclic relations from Wednesday, or the proofs about injectivity and surjectivity from Friday, are great examples of the style we're looking for.

Good luck, and have fun!

**Checkpoint due Monday, October 16th at 2:30PM.**

**Remaining problems due Friday, October 20th at 2:30PM.**

*This checkpoint problem is due on Monday at 2:30PM and should be submitted on GradeScope.*

## Checkpoint Problem: Redefining Strict Orders (2 Points If Submitted)

In Wednesday's lecture, we defined strict orders as binary relations that are irreflexive, asymmetric, and transitive. Interestingly, it turns out that we could have left asymmetry out of our definition and just gone with irreflexivity and transitivity.

i. Prove that a binary relation $R$ over a set $A$ is a strict order if and only if the relation $R$ is irreflexive and transitive.

Going forward, it turns out that one of the easiest ways to prove that a relation is a strict order is to prove that it's irreflexive and transitive. In fact, that's such good advice that we're going to remind you of it later on in this problem set. ☺

While we could have left irreflexivity out of the definition of a strict order, we could *not* have left out transitivity.

ii. Draw the graph of a binary relation that is irreflexive and asymmetric, but not transitive. Then, explain why the the relation you've picked shows that a binary relation can be irreflexive and asymmetric without being a strict order. (The graph of a relation is a the pictorial way of representing a binary relation $R$ over a set $A$ by drawing the elements of $A$, then drawing arrows to indicate which elements are related to one another.)

As a final note, it turns out that we *also* could have equivalently defined strict orders to be binary relations that are asymmetric and transitive. We're not going to ask you to prove this, but it's a great exercise if you want to give it a try!

*Although the remaining problems on this problem set aren't due until Friday, we strongly recommend starting them early. The coding component and early questions are great for building an intuition and you will likely want to stop by office hours to get some help or advice on them.*

## Problem One: So What Exactly Is a Binary Relation, Anyway?

When we described binary relations in lecture, we gave an *operational definition* of a binary relation by saying *what binary relations do*, but we never actually said *what binary relations are*.

Let's begin with a new definition. Given a set $A$, the ***Cartesian square of A***, denoted $A^2$, is the set of all ordered pairs that can be formed from elements of $A$. Formally speaking, we define $A^2$ as

$$A^2 = \{ (a_1, a_2) \mid a_1, a_2 \in A \}$$

For example, if $A = \{1, 3, 7\}$, then

$$A^2 = \{ (1, 1), (1, 3), (1, 7), (3, 1), (3, 3), (3, 7), (7, 1), (7, 3), (7, 7) \}.$$

We can use the Cartesian square of a set to formally define a binary relation. Formally speaking, a binary relation $R$ over a set $A$ is a set $R \subseteq A^2$. The ordered pairs in $R$ correspond to pairs of elements where the relation holds. For example, the $<$ relation over the set $\mathbb{N}$ would formally be defined as

$$< \; = \; \{ (0, 1), (0, 2), (0, 3), \ldots, (1, 2), (1, 3), (1, 4), \ldots, (2, 3), (2, 4), (2, 5), \ldots \}$$

When we've been talking about relations, we've used the notation $xRy$ to denote the fact that $x$ relates to $y$ by relation $R$. Formally speaking, the notation $xRy$ is just a shorthand for $(x, y) \in R$. This means that if you happen to stumble across a random set of pairs of things, you could interpret it as a binary relation.

Visit the CS103 website and download the starter project files for Problem Set Three. In `BinaryRelations.h`, there's a definition of a `Relation` type that represents a binary relation expressed as a set of ordered pairs. We'd like you to write some C++ code to analyze and manipulate those relations. You'll do all your coding in `BinaryRelations.cpp`.

i. Implement a function

```
bool isReflexive(Relation R);
```

that takes as input a binary relation $R$ and returns whether $R$ is reflexive.

ii. Implement a function

```
bool isSymmetric(Relation R);
```

that takes as input a binary relation $R$ and returns whether $R$ is symmetric.

iii. Implement a function

```
bool isTransitive(Relation R);
```

that takes as input a binary relation $R$ and returns whether $R$ is transitive.

iv. Implement a function

```
bool isIrreflexive(Relation R);
```

that takes as input a binary relation $R$ and returns whether $R$ is irreflexive.

v. Implement a function

```
bool isAsymmetric(Relation R);
```

that takes as input a binary relation $R$ and returns whether $R$ is asymmetric.

*(Continued on the next page)*

vi. Implement a function

```
bool isEquivalenceRelation(Relation R);
```

that takes as input a binary relation *R* and returns whether *R* is an equivalence relation.

vii. Implement a function

```
bool isStrictOrder(Relation R);
```

that takes as input a binary relation *R* and returns whether *R* is a strict order.

viii. Implement a function

```
std::vector<std::set<int>> equivalenceClassesOf(Relation R);
```

that takes as input a binary relation *R*, which you can assume is an equivalence relation, and returns a list of all equivalence classes of *R*. The return type is a `std::vector` (essentially, a list) of sets of integers; there's information in the starter files about how to work with `std::vector`. You should return exactly one copy of each equivalence class.

If you choose "Show Equivalence Classes" from the dropdown menu and select an equivalence relation, it will use your function to color-code the equivalence classes of that relation.

ix. Edit the file `PartA.relation` in the `res/` directory to define a binary relation that is neither symmetric nor asymmetric. This shows that the terms "symmetric" and "asymmetric" are not negations of one another. There's a description of the expected file format in this file.

x. Edit the file `PartB.relation` in the `res/` directory to define a binary relation that is both symmetric and asymmetric. *(Yes, this is possible!)*

xi. Edit the file `PartC.relation` in the `res/` directory to define a binary relation that is both reflexive and irreflexive. *(Yes, this is possible!)*

In the course of solving these programming problems, please do not edit any of the other starter files. You should submit the `BinaryRelations.cpp` file, along with the `.relation` files you edited. You can submit your answers as many times as you'd like; our autograder will provide feedback on how you're doing.

## Problem Two: Redefining Equivalence Relations?

In lecture, we defined equivalence relations as relations that are reflexive, symmetric, and transitive. Are all three parts of that definition necessary? The answer is yes, and this question explores why this is.

i. Edit the file `PartD.relation` in the `res/` directory of the starter files to define a binary relation that is symmetric, transitive, but not reflexive.

Below is a purported proof that every relation that is both symmetric and transitive is also reflexive.

***Theorem:*** If *R* is a symmetric and transitive binary relation over a set *A*, then *R* is also reflexive.

***Proof:*** Let *R* be an arbitrary binary relation over a set *A* such that *R* is both symmetric and transitive. We need to show that *R* is reflexive. To do so, consider an arbitrary $x, y \in A$ where $xRy$. Since *R* is symmetric and $xRy$, we know that $yRx$. Then, since *R* is transitive, from $xRy$ and $yRx$ we learn that $xRx$ is true. Therefore, *R* is reflexive, as required. ■

This proof has to be wrong, since as you saw in part (i) it's possible for a relation to be symmetric and transitive but not reflexive!

ii. What's wrong with this proof? Justify your answer. Be as specific as possible.

## Problem Three: Hasse Diagrams and Covering Relations

Let $<_A$ be some strict order relation over a set $A$. (We've typically used the letter $R$ as a placeholder for "some general relation," but it's common when working with strict orders to use notation like $<_A$ as a placeholder name.) We can define a new binary relation over the set $A$ called the ***covering relation for*** $<_A$, which we denote as $\text{Cov}(<_A)$, as follows:

$$x \text{ Cov}(<_A) y \quad \text{if} \quad x <_A y \ \wedge \ \neg\exists z \in A.\ (x <_A z \wedge z <_A y)$$

That definition is quite a mouthful, but it has a really nice intuition. In the course of working through this problem, you'll see what this definition means and get a better feel for it.

i. Consider the $<$ relation over the set $\mathbb{N}$. What is its covering relation? To provide your answer, fill in the blank below, then briefly justify your answer:

$$x \text{ Cov}(<) y \quad \text{if} \quad \_\_\_\_\_$$

For full credit, you should fill in the blank in the simplest way possible. There's a really short answer you can provide that doesn't require any first-order logic. See if you can find it! *(Hint: Try out some examples and see if you spot a pattern.)*

ii. Prove that the relation $\text{Cov}(<)$ you found in part (i) is ***not*** a strict order. This shows that if you start with a strict order and take its cover, you don't necessarily get back a strict order.

iii. Consider the $\subsetneq$ relation over the set $\wp(\mathbb{N})$. This relation is the strict subset relation, where $S \subsetneq T$ means that $S \subseteq T$ but that $S \neq T$. What is its covering relation? Provide your answer in a similar fashion to how you answered part (i) of this problem, and briefly justify your answer.

You might be wondering why on earth you'd ever want to look at cover relations. It turns out that they have a nice visual intuition.

iv. Let $<_A$ be a strict order over a set $A$. There is a close connection between the covering relation $\text{Cov}(<_A)$ and the Hasse diagram of $<_A$. What is it? Briefly justify your answer, but no proof is required.

Given the explanation you came up with in part (iv), you might wonder why we initially specified the definition in first-order logic. From a computer science perspective, first-order definitions are really useful because they give a clear, clean, unambiguous definition that can often be easily expressed in software.

v. Implement a function

```
Relation coverOf(Relation R);
```

that takes as input a binary relation $R$, which you can assume is a strict order, and returns $\text{Cov}(R)$. (As a reminder, don't forget to fill in the domain of $\text{Cov}(R)$ before you return it!)

Our provided starter files are designed so that you can see what the cover relations of different strict orders look like. Just choose the "Show Covering Relation" option from the middle dropdown menu. If you've selected a strict order, it will show just the arrows from the covering relation. You may want to use this to check your work for the earlier parts of this problem, and more generally just to get a sense of what cover relations look like!

## Problem Four: Strict Orders and C++ Operator Overloading

The C++ programming language lets you define your own custom types. For example, here's a type representing a pixel's position on the screen:

```
struct Pixel {
    int x;
    int y;
};
```

This is a *structure*, a type representing a bunch of different objects all packaged together as one. Here, this structure type groups together two `int`s named `x` and `y`. The name `Pixel` refers to a type, just like `int` or `string`. You can create variables of type `Pixel` just as you can variables of any other type, like this:

```
Pixel p;
```

One you have a variable of type `Pixel`, you can access the constituent elements of the `struct` by using the dot operator. For example:

```
Pixel p;
p.x = 137;
p.y++;
```

As you've seen in Problem Set One and Problem Set Two, in the C++ standard library there's a type called `std::set` which represents a set of values. The `std::set` is (usually) implemented with a data structure called a *binary search tree*. (The details of how binary search trees work are covered in CS106B, and so we won't go into detail about how they work). Of interest here, this means that the `std::set` type requires that elements of the stored type be comparable using the `<` operator. For primitive types like `int` and `double`, that's not a problem. However, if you took the above `Pixel` `struct` and tried to form a `std::set<Pixel>`, you'd get some horrible compiler errors because, by default, it's not possible to compare two `Pixel`s using the `<` operator. On my system, making a `std::set<Pixel>` generates *ninety-six lines* of compiler errors that ultimately track back to the missing less-than operator.

To address this, C++ has a feature called *operator overloading* that us define how to apply the `<` operator to `Pixel`s so that we can make a `std::set<Pixel>` without the compiler yelling at us. In C++, the syntax[1] for overloading the less-than operator on `Pixel`s looks like this:

```
bool operator < (Pixel lhs, Pixel rhs) {
    /* … do some work, then return true or return false … */
}
```

That syntax might look a bit frightening, so let's dissect it. The above is a definition of a C++ function. The name of the function is `operator <` (yes, that's a legal function name). It takes in two `Pixel`s, which correspond to the two operands to the `<` sign (the first argument is the one on the left, and the second is the one on the right), and it returns a `bool` indicating whether the `Pixel` named `lhs` is "less than" the `Pixel` named `rhs`. Aside from the funny name, this function behaves just like every other C++ function, and you can put whatever code in it that you'd like.

The folks who designed the C++ programming language happened to be very familiar with discrete mathematics, and so the language standard gives a bunch of rules about how this `operator <` function should behave. If you define a custom `operator <` function for a type you want to use with `std::set`, C++ requires that the `<` relation be a strict order over the underlying type.

*(Continued on the next page)*

1 Typically, you'd pass those arguments by `const` reference for efficiency, but that's a C++ism we'll ignore for now.

This question explores the connection between C++ coding and all this theory we've built up about binary relations. For the purposes of this problem, you can assume that, given any two `int`s $x$ and $y$, that exactly one of the following is true:

$$x < y \qquad x = y \qquad y < x$$

i. Let's imagine that we implement `operator <` for `Pixel`s using the following function:

```
bool operator < (Pixel lhs, Pixel rhs) {
    return lhs.x < rhs.x;
}
```

This corresponds to the following relation $F$ over the set of all pixels:

$$pFq \qquad \text{if} \qquad p.x < q.x$$

Prove or disprove: the relation $F$ defined over the set of all `Pixel`s this way is a strict order.

ii. Alternatively, suppose that we implement `operator <` for `Pixel`s using the following function:

```
bool operator < (Pixel lhs, Pixel rhs) {
    return lhs.x < rhs.x || lhs.y < rhs.y;
}
```

This corresponds to the following relation $G$ over the set of all pixels:

$$pGq \qquad \text{if} \qquad p.x < q.x \ \vee \ p.y < q.y$$

Prove or disprove: the relation $G$ defined over the set of all `Pixel`s this way is a strict order.

iii. Now let's imagine that we implement `operator <` for `Pixel`s using the following function:

```
bool operator < (Pixel lhs, Pixel rhs) {
    return lhs.x < rhs.x && lhs.y < rhs.y;
}
```

This corresponds to the following relation $H$ over the set of all pixels:

$$pHq \qquad \text{if} \qquad p.x < q.x \ \wedge \ p.y < q.y$$

Prove or disprove: the relation $H$ defined over the set of all `Pixel`s this way is a strict order.

iv. And finally, suppose that we implement `operator <` for `Pixel`s using the following function:

```
bool operator < (Pixel lhs, Pixel rhs) {
    return (lhs.x < rhs.x) || (lhs.x == rhs.x && lhs.y < rhs.y);
}
```

This corresponds to the following relation $I$ over the set of all pixels:

$$pIq \qquad \text{if} \qquad p.x < q.x \ \vee \ (p.x = q.x \wedge p.y < q.y)$$

Prove or disprove: the relation $I$ defined over the set of all `Pixel`s this way is a strict order.

When writing up your proofs, please feel free to take advantage of the result from the checkpoint problem! You can prove a relation is a strict order by just showing that it's irreflexive and transitive.

If you're having trouble building an intuition for what these relations look like, it might help to draw a grid of points, single one out of those points, and shade in all the points it relates to.

## Problem Five: Strict Weak Orders and C++ Programming

*(This is a follow-up to Problem Four, so you may want to complete it before attempting this problem.)*

The `std::set` type imposes more restrictions on `operator<` than just requiring it to be a strict order. Specifically, the C++ requires that if you define a `operator<` function for use in `std::set`, that function has to define a special kind of relation called a ***strict weak order*** over the underlying type.

To understand what a strict weak order is, we need to introduce the notion of incomparability. Given a binary relation $R$ over a set $A$, the ***incomparability relation of R***, denoted $\sim_R$, is this binary relation over $A$:

$$x \sim_R y \quad \text{if} \quad x \not R y \text{ and } y \not R x.$$

Notice that there are slashes through those $R$'s, so $x \sim_R y$ means "neither $xRy$ nor $yRx$ are true." This definition might seem pretty abstract, so let's start by trying to make this a bit more concrete.

i. Consider the $<$ relation over the set $\mathbb{N}$. What is the relation $\sim_<$? Provide your answer by filling in the blank below. Briefly justify your answer, and see if you can find the simplest answer you can:

$$x \sim_< y \quad \text{if} \quad \_\_\_\_\_$$

And now, our key definition for this problem. A binary relation $R$ over a set $A$ is called a ***strict weak order*** if it is a strict order and its incomparability relation $\sim_R$ is transitive.

ii. Look at the $F$, $G$, and $H$ relations from Problem Four. For each of those relations, prove or disprove the following claim: that binary relation is a strict *weak* order over `Pixel`s. (It turns out that the $I$ relation is also a strict weak order, but for brevity's sake you don't need to prove this.)

You might wonder what's so special about the definition of a strict weak order that the C++ folks decided to enshrine it in the formal language specification. The reason has to do with a specific property of strict weak orders that connects them back to equivalence relations.

iii. Prove that if $R$ is a strict weak order over a set $A$, then $\sim_R$ is an equivalence relation over $A$.

Your result from part (iii) of this problem shows that a strict order $R$ over a set $A$ effectively partitions the elements of $A$ into non-overlapping groups. This follows from the Fundamental Theorem of Equivalence Relations: any equivalence relation defines a partition of the underlying set.

As a reminder, the notation $[p]_{\sim R}$ denotes the equivalence class of $p$ with respect to the $\sim_R$ relation.

iv. Let $p$ be a point whose $x$ and $y$ components $p.x$ and $p.y$ are chosen arbitrarily. For each of the relations from Problem Four that are strict weak orders, determine what $[p]_{\sim R}$ is and express your answer as simply as possible by filling in the following blank. No proof is necessary.

$$[p]_{\sim R} \quad = \quad \{ \_\_\_\_\_ \}$$

The `std::set` type, like the mathematical sets we've studied up to this point, does not allow for duplicate elements. However, the way that it determines what a "duplicate" is is by looking at the $\sim_R$ relation corresponding to `operator <`. Specifically, whenever you insert an element $x$ into an `std::set`, the `std::set` will add it if there are no other elements of $[x]_{\sim R}$ already in the set and will discard it otherwise. In other words, the `std::set` only stores the first element of each equivalence class inserted into it.

v. Suppose you insert the pixels (137, 42), (42, 137), (137, 103), and (42, 103) into an empty `std::set<Pixel>`, in that order. For each of the relations defined in Problem Four that are strict weak orders, determine what the final contents of the `std::set` will end up being if the set uses those strict weak orders to compare elements. Briefly justify your answer; no formal proof is necessary.

## Problem Six: Strict Weak Orders in Theoryland

*(This is a follow-up to Problem Five, so you may want to complete it before attempting this problem.)*

As a refresher from Problem Five, a ***strict weak order*** is a strict order $R$ over a set $A$ where the following relation, called the ***incomparability relation***, is transitive:

$$x \sim_R y \quad \text{if} \quad x \not R y \text{ and } y \not R x.$$

As you proved in Problem Five, if $R$ is a strict weak order, then $\sim_R$ is an equivalence relation. The notation $[a]_{\sim R}$ refers to the equivalence class of $\sim_R$ containing $a$.

There's a beautiful interplay between the relation $R$ and the equivalence classes of $\sim_R$.

i. Let $R$ be a strict weak order over a set $A$ and consider any $x, y \in A$ where $xRy$. Prove that for any $z \in A$ where $z \in [y]_{\sim R}$ that $xRz$.

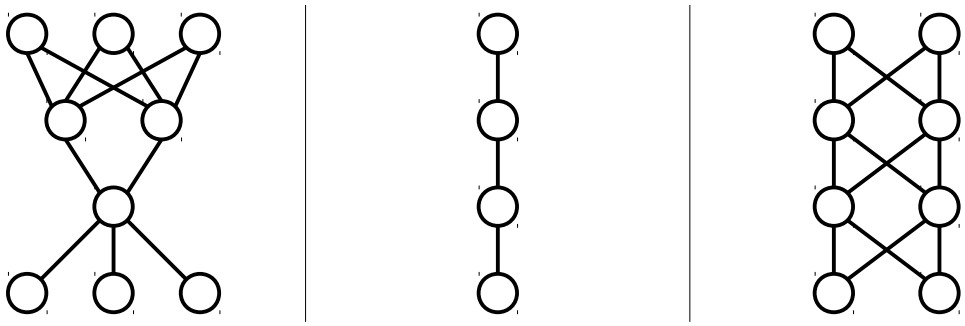
ii. Let $R$ be a strict weak order over a set $A$ and consider any $x, y \in A$ where $xRy$. Prove that for any $w \in A$ where $w \in [x]_{\sim R}$ that $wRy$.

These results show that given two equivalence classes $[x]_{\sim R}$ and $[y]_{\sim R}$ of the incomparability relation, either every element of $[x]_{\sim R}$ relates to every element of $[y]_{\sim R}$, or every element of $[y]_{\sim R}$ relates to every element of $[x]_{\sim R}$, or $[x]_{\sim R}$ and $[y]_{\sim R}$ are just different names for the same set.

Practically speaking, this makes strict weak orders excellent for setups where you need to keep things in sorted order. If you pick any group of elements from a strict weak order, you sort them so that each element relates to or is indistinguishable from all the elements after it.

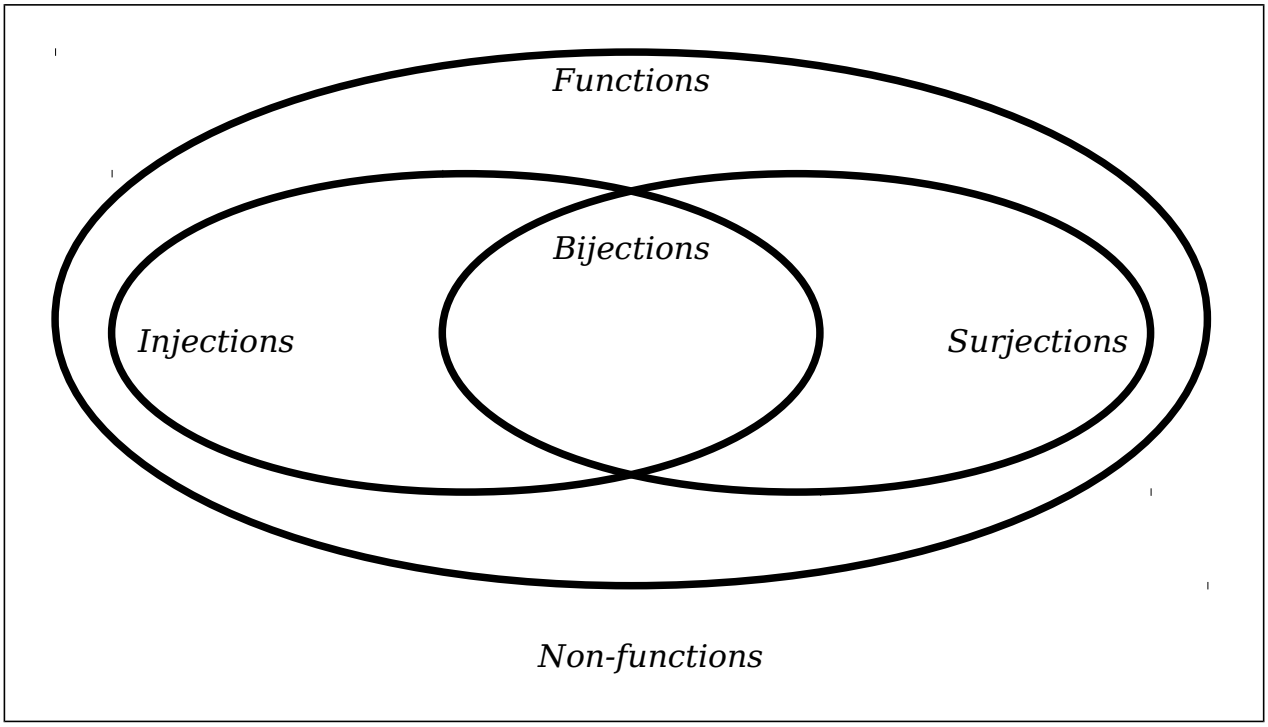
Theoretically speaking, this means that the Hasse diagrams of strict weak orders can be nicely split apart into a bunch of different "layers," where each layer represents an equivalence class and each element of each layer is connected directly to the layer above it. The layers must stack on top of one another and there can't be any two "incomparable" layers thanks to what you proved above.

Here are some examples of what these might look like:



## Problem Seven: Properties of Functions

Consider the following Venn diagram:




Below is a list of purported functions. For each of those purported functions, determine where in this Venn diagram that object goes. No justification is necessary.

1. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = n^2$
2. $f : \mathbb{Z} \to \mathbb{N}$ defined as $f(n) = n^2$
3. $f : \mathbb{N} \to \mathbb{Z}$ defined as $f(n) = n^2$
4. $f : \mathbb{Z} \to \mathbb{Z}$ defined as $f(n) = n^2$
5. $f : \mathbb{R} \to \mathbb{N}$ defined as $f(n) = n^2$
6. $f : \mathbb{N} \to \mathbb{R}$ defined as $f(n) = n^2$
7. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = \sqrt{n}$ . ( $\sqrt{n}$ is the ***principal square root*** of $n$, the nonnegative one.)
8. $f : \mathbb{R} \to \mathbb{R}$ defined as $f(n) = \sqrt{n}$ .
9. $f : \mathbb{R} \to \{ x \in \mathbb{R} \mid x \geq 0 \}$ defined as $f(n) = \sqrt{n}$ .
10. $f : \{ x \in \mathbb{R} \mid x \geq 0 \} \to \{ x \in \mathbb{R} \mid x \geq 0 \}$ defined as $f(n) = \sqrt{n}$ .
11. $f : \{ x \in \mathbb{R} \mid x \geq 0 \} \to \mathbb{R}$ defined as $f(n) = \sqrt{n}$ .
12. $f : \mathbb{N} \to \wp(\mathbb{N})$, where $f$ is some injective function.
13. $f : \{0, 1, 2\} \to \{3, 4\}$, where $f$ is some surjective function.
14. $f : \{\textit{breakfast}, \textit{lunch}, \textit{dinner}\} \to \{\textit{shakshuka}, \textit{soondubu}, \textit{maafe}\}$, where $f$ is some injection.

## Problem Eight: Left, Right, and True Inverses

Let $f : A \to B$ be a function. A function $g : B \to A$ is called a ***left inverse*** of $f$ if the following is true:

$$\forall a \in A.\ g(f(a)) = a.$$

i. Find examples of a function $f$ and two *different* functions $g$ and $h$ such that both $g$ and $h$ are left inverses of $f$. This shows that left inverses don't have to be unique. (Two functions $g$ and $h$ are different if there is some $x$ where $g(x) \neq h(x)$.) *(Hint: Define your functions through pictures.)*

ii. Prove that if $f$ has a left inverse, then $f$ is injective. Your proof must call back to the formal definitions of injectivity and left inverses, but as with the other proofs in this problem set must *not* contain any first-order logic.

Let $f : A \to B$ be a function. A function $g : B \to A$ is called a ***right inverse*** of $f$ if the following is true:

$$\forall b \in B.\ f(g(b)) = b.$$

iii. Find examples of a function $f$ and two different functions $g$ and $h$ such that both $g$ and $h$ are right inverses of $f$. This shows that right inverses don't have to be unique.

iv. Prove that if $f$ has a right inverse, then $f$ is surjective.

If $f : A \to B$ is a function, then a ***true inverse*** (often just called an ***inverse***) of $f$ is a function $g$ that's simultaneously a left and right inverse of $f$. In parts (i) and (iii) of this problem you saw that functions can have several different left inverses or right inverses. However, a function can only have a single true inverse.

v. Prove that if $f : A \to B$ is a function and both $g_1 : B \to A$ and $g_2 : B \to A$ are inverses of $f$, then $g_1(b) = g_2(b)$ for all $b \in B$.

vi. Explain why your proof from part (v) doesn't work if $g_1$ and $g_2$ are just *left* inverses of $f$, not full inverses. Be specific – you should point at a specific claim in your proof of part (v) that is no longer true in this case.

vii. Explain why your proof from part (v) doesn't work if $g_1$ and $g_2$ are just right inverses of $f$, not full inverses. Be specific – you should point at a specific claim in your proof of part (v) that is no longer true in this case.

Left and right inverses have some surprising applications. We'll see one of them, which talks about the limits of data compression, next week!

## Optional Fun Problem: Infinity Minus Two (1 Point Extra Credit)

Let [0, 1] denote the set $\{ x \in \mathbb{R} \mid 0 \leq x \leq 1 \}$ and (0, 1) denote the set $\{ x \in \mathbb{R} \mid 0 < x < 1 \}$. That is, the set [0, 1] is the set of all real numbers between 0 and 1, *inclusive*, and the set (0, 1) is the set of all real numbers between 0 and 1, *exclusive*. These sets differ only in that the set [0, 1] includes 0 and 1 and the set (0, 1) excludes 0 and 1.

Give the definition of bijection $f : [0, 1] \to (0, 1)$ via an explicit rule (i.e. writing out $f(x) =$ _____ or defining $f$ via a piecewise function), then prove that your function is a bijection.