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, we ***strongly*** recommend reading over the following handouts:

- Handout 17, the Guide to Proofs on Discrete Structures, which explores how to write proofs when definitions are rigorously specified in first-order logic. This handout contains both general guiding principles to follow and some sample proof templates that you're welcome to use here.
- Handout 18, the Discrete Structures Proofwriting Checklist, which contains some specific items to look for when proofreading your work.

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 Friday, or the proofs about injectivity and surjectivity from this upcoming Monday, are great examples of the style we're looking for.

As with the last problem set, a few of these questions require concepts that we won't have covered when this problem set goes out. Those problems are clearly marked and we'll cover the relevant topics on Monday.

Good luck, and have fun!

Checkpoint due Monday, January 29th at 2:30PM.

Remaining problems due Friday, February 2nd at 2:30PM.

*These **two** checkpoint problems on this problem set are due on Monday at 2:30PM.*

Checkpoint Problem One: Binary Relations IRL

The first part of this problem revolves around a mathematical construct called ***homogenous coordinates*** that shows up in computer graphics. If you take CS148, you'll get to see how they're used to quickly determine where to display three-dimensional objects on screen.

Let $\mathbb{R}^2$ denote the set of all ordered pairs of real numbers. (There's a justification for where the notation $\mathbb{R}^2$ comes from in Problem One on this problem set). For example $(137, 42) \in \mathbb{R}^2$, $(\pi, e) \in \mathbb{R}^2$, and $(-2.71, 103) \in \mathbb{R}^2$. Two ordered pairs are equal if and only if each of their components are equal. That is, we have $(a, b) = (c, d)$ if and only if $a = c$ and $b = d$.

Consider the relation E defined over $\mathbb{R}^2$ as follows:

$$(x_1, y_1)\ E\ (x_2, y_2) \quad \text{if} \quad \text{there is a nonzero real number } k \text{ such that } (kx_1, ky_1) = (x_2, y_2).$$

For example, $(3, 4)\ E\ (6, 8)$ because $(2 \cdot 3, 2 \cdot 4) = (6, 8)$.

i. Prove that E is an equivalence relation.

Remember that the "if" in the definition of the relation E means "is defined as" and isn't an implication.

The second part of this problem explores an entirely different example. Although no one really "wins" the Olympics, it's tempting to find some way of ranking countries by the numbers of medals that they've won. Let's have $\mathbb{N}^3$ represent the set of all ordered triples of natural numbers (you'll see where this notation comes from on Problem Set Five). For example, we have $(137, 42, 0) \in \mathbb{N}^3$ and $(0, 1, 2) \in \mathbb{N}^3$.

We can represent how many medals a country has won as a triple (g, s, b) of natural numbers where g, s, and b represent the number of gold, silver, and bronze medals that the country has won, respectively. We can then define a new binary relation R over $\mathbb{N}^3$ as follows:

$$(g_1, s_1, b_1)\ R\ (g_2, s_2, b_2) \quad \text{if} \quad 5g_1 + 3s_1 + b_1 < 5g_2 + 3s_2 + b_2.$$

This relation essentially gives a country five points for each gold medal, three for each silver, and one for each bronze, then ranks countries by their weighted medal scores.

ii. Prove that R is a strict order. For the purposes of this problem – and every other problem, for that matter – you can assume that the $<$ relation over natural numbers is a strict order.

Once you've written a draft of your proofs of these results, take a few minutes to read over them and apply both the standard Proofwriting Checklist (the one you've used on the first two problem sets) and the new Discrete Structures Proofwriting Checklist. Here are a few specific things to keep an eye on:

- *The key terms in binary relations (reflexivity, symmetry, transitivity, irreflexivity, and asymmetry) are defined in first-order logic and proofs of those properties depend on those first-order definitions. However, as a reminder, you should **not** include first-order logic notation (quantifiers, connectives, etc.) anywhere in your proofs. Look at the proofs from the Guide to Binary Relations and last week's lectures for some examples of what we're expecting.*
- *Make sure that you've set all of your proofs up properly. For example, what should a proof that a relation is symmetric look like? What should you assume, and what you should prove? Does your proof match this pattern?*
- *You don't need to – and in fact, shouldn't – repeat the definitions of the E or R relations in your proofs. You can assume that the reader knows how they're defined.*

Checkpoint Problem Two: Redefining Equivalence Relations?

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 it's possible for a relation to be symmetric and transitive but not reflexive. What's wrong with this proof? Justify your answer. Be as specific as possible.

*It is **critical** that you understand the answer to this problem **completely** and **unambiguously** before you start working on any other proofs on this problem set. It would be a shame if you misidentified the issue above and then went on to make this exact mistake on all the problems in this problem set.*

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 Strict Orders

In Friday'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.

Once you've written up your proof, take a minute to critique your proof by applying the Proofwriting Checklist and the Discrete Structures Proofwriting Checklist. Also, think about the following:

- *The statement you need to prove here is a biconditional. How do you prove a biconditional statement? What should you be assuming in each part of the proof? What do you need to prove?*
- *At some point, you'll need to prove that R is asymmetric under some set of assumptions. What does a proof of asymmetry look like? What should you be assuming? What should you be proving?*
- *The proof you will be doing here will involve reasoning about arbitrarily-chosen binary relations where you have no idea what the relation is or what set it's over and only know some properties that it must have (say, that it's irreflexive). In cases like these, be very careful not to make claims about how the relation works that don't immediately follow from your assumptions. After all, your relation could be something like the $<$ relation over $\mathbb{N}$, or the $\subsetneq$ relation over $\wp(\mathbb{R})$, or the "runs strictly faster than" relation over people.*

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. Edit the file `PartD.relation` in the `res/` directory to define a binary relation that is irreflexive and asymmetric, but not transitive. This shows that a relation that's asymmetric and irreflexive isn't necessarily a strict order.

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!

Problem Three: Covering Relations and Hasse Diagrams

Let R be a strict order over a set A. The ***covering relation for R***, denoted **Cov(*R*)**, is a binary relation over the set A that's defined as follows:

$$x \operatorname{Cov}(R)\, y \quad \text{if} \quad xRy \land \neg\exists z \in A.\ (xRz \land zRy)$$

That definition is quite a mouthful and, as with many dense mathematical definitions, you're not expected to be able to look at it and immediately understand it. Instead, you'll build up an understanding for what the definition means by playing around with it in a few example cases and seeing if you can spot a pattern. Trust us – the above definition has a really nice intuition once you've gotten the hang of it. (In case you're wondering, the notation $x \operatorname{Cov}(R)\, y$ is read aloud as "y covers x" or "x is covered by y.")

The above definition works for any strict order R over any set A, but it's probably easier to see how it works by looking at how it works for a particular choice of a strict order.

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

$$x \operatorname{Cov}(<)\, y \quad \text{if} \quad \underline{\hspace{6cm}}$$

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!

As a hint for how to approach the above problem, a perfectly reasonable strategy is to start off by picking random pairs of natural numbers x and y and seeing whether x Cov(<) y by looking at the above definition. If you repeat this enough times, you can start formulating a hypothesis of how you think Cov(<) *behaves, and eventually you'll land at the answer. Another option would be to plug in < and $\mathbb{N}$ into the above definition, then to look over what comes back and see if you can make sense of what you've found.*

Now that you've found out what Cov(R) looks like in one particular instance, it's reasonable to go and ask what properties you might expect of a cover relation. Are cover relations equivalence relations? Are they strict orders? It's hard to tell this just by eyeballing the definition given above, but now that you have a concrete example of a cover relation in hand, you can start to think about how to answer these questions.

ii. Prove that the relation 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.

As a hint, if you're trying to show that some relation isn't a strict order, what exactly is it that you need to prove about that relation? Start off by getting out a piece of scratch paper and writing out what you'd need to prove. If you do this properly, you should find that you need to prove that one of three (or two) things is true about the binary relation. You can then investigate whether each of those properties is true about Cov(<) *and use that to determine what you'll need to prove.*

So you now know what Cov(<) looks like, and you know that cover relations aren't always strict orders. But at this point, you only have a single example of a cover relation. To get a better sense for what cover relations look like more generally, it might help to look at some other strict orders and their covers.

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.

As you're working through this problem, make sure you can answer the following question: what does it mean for $\subsetneq$ to be a binary relation over $\wp(\mathbb{N})$?

(Continued on the next page)

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

You may want to draw some pictures.

At this point, you've started with a Hairy Scary Definition that's full of dense first-order logic terms, gotten to see what it looks like in some concrete cases, and learned some of the properties of that definition. Nice job! To round out this section, we'd like you to write a short piece of code that lets you get a visual intuition for what covering relations look like.

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 $\operatorname{Cov}(R)$. (As a reminder, don't forget to fill in the domain of $\operatorname{Cov}(R)$ 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.

The provided example relations include a sample of the less-than relation over $\mathbb{N}$ and a sample of the strict subset relation over $\wp(\mathbb{N})$. We strongly recommend using the code you wrote to visualize what their cover relations look like and to compare what you find against the answers you came up with in parts (i) and (iii). Does what you see in practice match what you predicted in theory?

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 the < relation over integers is a strict order and that it behaves the way you've seen it behave in high-school math classes.

i. 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 F over the set of all pixels:

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

Prove that the relation F defined over the set of all `Pixel`s this way is a strict order.

As a reminder, as you saw on Problem One, to prove that F is a strict order, you just need to show that it's irreflexive and transitive. There's no need to prove asymmetry.

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:

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

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

At this point you have experience both proving that a relation is a strict order (you did this on the checkpoint and in part (i) of this problem) and proving that a relation is not a strict order (you did this in part (ii) of the problem on cover relations). To solve this problem, you'll need to first determine whether G is a strict order, then write up a proof or disproof as appropriate. So consider approaching things this way: write out what it is that you'd need to prove if you want to show that G is a strict order, and write out what it is that you'd need to prove if you wanted to show that G is not a strict order. Then, look at what you find, play around with what this would say about G, and see if you can decide which option is correct.

iii. 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 H over the set of all pixels:

$$pHq \qquad \text{if} \qquad p.x < q.x \;\lor\; (p.x = q.x \land p.y < q.y)$$

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

Problem Five: Strict Weak Orders and C++ Operator Overloading

(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. (In case you're wondering, the notation $x \sim_R y$ would be read aloud as "x is indistinguishable from y.")

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:

$$x \sim_< y \quad \text{if} \quad \underline{\hspace{6cm}}$$

You already have experience solving problems like this one from when you did the cover relations problem. See if the approach or approaches you took when solving that problem can help you out here.

ii. Look at the F, G, and H relations from Problem Four and determine what the $\sim_F$, $\sim_G$, and $\sim_H$ relations are by filling in the following blanks. As before, try to find the simplest possible answers that you can, but no justifications are necessary.

$$p \sim_F q \quad \text{if} \quad \underline{\hspace{6cm}}$$
$$p \sim_G q \quad \text{if} \quad \underline{\hspace{6cm}}$$
$$p \sim_H q \quad \text{if} \quad \underline{\hspace{6cm}}$$

The later parts of this problem will reference the answers that you came up with here, so make sure that you've checked your work before moving on. Not sure how to do that? Pick a few random pairs of pixels. Check whether the F, G, and H relations hold between those pixels in each direction. Based on that, determine whether $\sim_F$, $\sim_G$, and $\sim_H$ should hold between those pixels. Then, see whether the definitions you came up with above agree with those results. If you see a mismatch, it means that you probably didn't pin the definition down correctly.

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.

iii. 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 `Pixels`. You can reference the definitions of $\sim_F$, $\sim_G$, and $\sim_H$ that you found above in the course of writing your proofs without justifying why those definitions are correct.

As with all prove-or-disprove problems, we recommend getting out a sheet of scratch paper and writing out two columns: what you'd need to show if you want to prove that a relation is a strict weak order, and what you'd need to show if you want to disprove that a relation is a strict weak order.

*As a hint, if you've done everything correctly up to this point, you should have found that **exactly one** of F, G, and H is a strict weak order.*

Problem Six: Strict Weak Orders and Equivalence Classes

(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:

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

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.

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

This is one of those problems where you need to be precise with what it is that you're assuming and what it is you need to prove. The two-column strategy from lecture is a good one to use here. What exactly is it that you're assuming about R? What do you need to prove about $\sim_R$? For each claim you need to prove about $\sim_R$, how would you go about proving it?

The Fundamental Theorem of Equivalence Relations, which we mentioned in class, essentially says that every equivalence relation splits the elements of its underlying set apart into non-overlapping groups (the equivalence classes of that relation). Your result from part (i) shows that if you have a strict weak order R over a set A, then the $\sim_R$ relation ends up splitting the elements of A apart into non-overlapping equivalence classes.

A reasonable question to ask, then, is what exactly those equivalence classes would look like. Since you happen to have a few strict weak orders lying around right now, it's reasonable to see what equivalence classes you get back from those orders.

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

ii. Let p be a point whose x and y components $p.x$ and $p.y$ are chosen arbitrarily. In Problem Five you found that one of the binary relations R from Problem Four of this problem is a strict weak order. For that relation, determine what $[p]_{\sim R}$ is and express your answer as simply as possible by filling in the following blank. No proof is necessary.

$$[p]_{\sim R} \quad = \quad \{ \quad \underline{\hspace{6cm}} \quad \}$$

There are a few ways you might go about solving this problem. You could start off by writing out the definition of an equivalence class, plugging in some point p, and then trying to simplify what you have by using the definition of the incomparability relations you came up with earlier on. Or you could pick some pixel p and start thinking about what pixels it would be incomparable with, then see if you can spot a pattern.

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.

iii. Suppose you insert the pixels (137, 42), (42, 137), (137, 103), and (42, 103) into an empty `std::set<Pixel>`, in that order, assuming that `std::set` uses the strict weak order you identified from Problem Four. Determine what the final contents of the `std::set` will end up being. Briefly justify your answer; no formal proof is necessary.

As a hint, use your result from part (ii).

Problem Seven: Strict Weak Orders in Theoryland

(This is a follow-up to Problem Six, 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 Six, 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. It turns out that there's a beautiful interplay between the relation R and the equivalence classes of $\sim_R$.

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.

This is probably the trickiest proof involving binary relations that you've seen up to this point. Here are some things to watch out for when you're working through this problem:

- *We definitely don't recommend immediately jumping into a final proof of this result. You'll want to go through some scratch work first.*
- *Unpack the relevant terms and definitions. If $z \in [y]_{\sim R}$, what can you say about how y and z are related by the relation R?*
- *Be very careful not to make any claims that don't follow from your previous assumptions and the definitions of the relevant terms. If you want to claim something like the following, for example, you'd need to prove it first, since nothing in the definition of R or $\sim_R$ says that this should be true:*

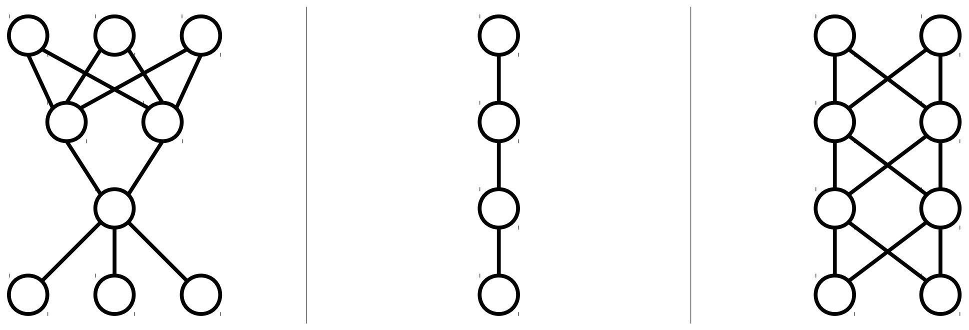
$$\textit{If } aRb \textit{ and } b \sim_R c\textit{, then } aRc$$

And now some context! Your result from above shows 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.

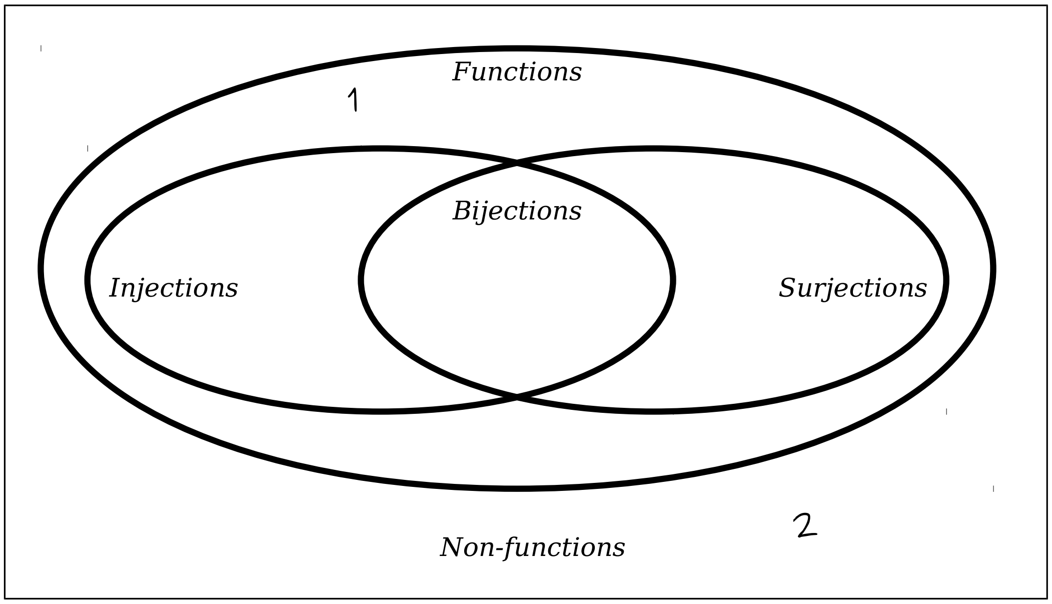
Here are some examples of what these might look like:



Problem Eight: Properties of Functions

(We will cover the material necessary to solve this problem in Monday's lecture.)

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.

To submit your answers, edit the file `FunctionsVennDiagram.h` in the `src/` directory of the starter files for this problem set. For simplicity, we've shown you where functions 1 and 2 go in this Venn diagram.

1. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = 137$
2. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = -137$
3. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = n^2$
4. $f : \mathbb{Z} \to \mathbb{N}$ defined as $f(n) = n^2$
5. $f : \mathbb{N} \to \mathbb{Z}$ defined as $f(n) = n^2$
6. $f : \mathbb{Z} \to \mathbb{Z}$ defined as $f(n) = n^2$
7. $f : \mathbb{R} \to \mathbb{N}$ defined as $f(n) = n^2$
8. $f : \mathbb{N} \to \mathbb{R}$ defined as $f(n) = n^2$
9. $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = \sqrt{n}$. ($\sqrt{n}$ is the ***principal square root*** of n, the nonnegative one.)
10. $f : \mathbb{R} \to \mathbb{R}$ defined as $f(n) = \sqrt{n}$.
11. $f : \mathbb{R} \to \{ x \in \mathbb{R} \mid x \geq 0 \}$ defined as $f(n) = \sqrt{n}$.
12. $f : \{ x \in \mathbb{R} \mid x \geq 0 \} \to \{ x \in \mathbb{R} \mid x \geq 0 \}$ defined as $f(n) = \sqrt{n}$.
13. $f : \{ x \in \mathbb{R} \mid x \geq 0 \} \to \mathbb{R}$ defined as $f(n) = \sqrt{n}$.
14. $f : \mathbb{N} \to \wp(\mathbb{N})$, where f is some injective function.
15. $f : \{0, 1, 2\} \to \{3, 4\}$, where f is some surjective function.
16. $f : \{$*breakfast*, *lunch*, *dinner*$\} \to \{$*shakshuka*, *soondubu*, *maafe*$\}$, where f is some injection.

Problem Nine: Left, Right, and True Inverses

(We will cover the material necessary to solve this problem in Monday's lecture.)

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 $g(x) \neq h(x)$.)

Although you're probably tempted to define functions by writing out some expression like $f(x) = x^2 + 3x - 1$, for the purposes of this problem it's actually a lot easier to define your functions by drawing diagrams of their domains and codomains like we did in lecture. This gives you more precise control over what maps to what.

ii. Prove that if f is a function that has a left inverse, then f is injective.

As a hint on this problem, look back at the proofs we did with injections in lecture. To prove that a function is an injection, what should you assume about that function, and what will you end up proving about it?

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 is a function that 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 next week!

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

(We will cover the material necessary to solve this problem in Monday's lecture.)

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.