Problem Set 1

Here we are – the first problem set of the quarter! This problem set is designed to give you practice writing proofs on a variety of different topics. We hope that this problem set gives you a sense of what proof-based mathematics is like and helps solidify your understanding of set theory.

Before you start this problem set, please do the following:

- Sign up for Piazza so that you have an easy way to ask us questions.
- Review the office hours timetable to find good times to drop on by to ask questions.
- Review Handout #07, "Set Theory Definitions," for a refresher on key terms, definitions, and theorems about set theory that might come up in this problem set.
- Review Handout #08, "Guide to Proofs," which has advice about how to write and structure your proofs.
- Review Handout #09, "Mathematical Vocabulary," which covers mathematical phrases you may need to use in your proofs and how to use them correctly.
- Review Handout #10, "Guide to Indirect Proofs," which provides some guidance about how to set up proofs by contradiction and contrapositive.
- Review Handout #11, "Ten Techniques to Get Unstuck," for advice about how to make progress on these sorts of problems when you're not sure what to do.
- Review Handout #12, "Proofwriting Checklist," for a detailed set of criteria you should apply to your proofs before submitting them. ***We will be running this same checklist on your proofs when grading, so please be sure to look over it before submitting!***
- Review the online "Guide to $\in$ and $\subseteq$" to make sure you understand the distinction between these terms.

As always, please feel free to drop by office hours or post on Piazza if you have any questions. We're happy to help out.

Good luck, and have fun!

Checkpoint Questions Due Monday, January 15th at 2:30PM Pacific time.
Remaining Questions Due Friday, January 19th at 2:30PM Pacific time.

Write your solutions to the following checkpoint problems and submit them through GradeScope by Monday at 2:30PM Pacific time. These problems will be graded on a 0 / 1 / 2 scale. Solutions that reasonably attempt to solve all of the problems, even if the attempts are incorrect, will receive two points. Solutions that reasonably attempt some but not all of the problems will receive one point. Solutions that do not reasonably attempt any of the problems – or solutions that are submitted after the deadline – will receive zero points.

Essentially, if you've made a good, honest effort to solve all of the problems and you submit on time, you should receive two points even if your solutions contain errors.

Please make the best effort you can when solving these problems. We want the feedback we give you on your solutions to be as useful as possible, so the more time and effort you put into them, the better we'll be able to comment on your proof style and technique. We will try to get these problems returned to you with feedback on your proof style by Wednesday. Submission instructions are included in the "Problem Set Policies" handout.

Checkpoint Problem One: Finding Negations

In order to write a proof by contradiction or contrapositive, you'll need to determine the negation of one or more statements. In Friday's lecture, we talked about a few common classes of statements and how to form their negations. Using what you've learned, answer the following multiple-choice questions and ***briefly explain how you arrived at your answer***.

Which of the following is the negation of "everything that has a beginning has an end?"

A) Everything that does not have a beginning has an end.
B) Everything that has a beginning has no end.
C) There is something that has no beginning and has an end.
D) There is something that has a beginning and has no end.

Which of the following is the negation of "there is a successful person who is grateful?"

A) There is an unsuccessful person who is grateful.
B) There is a successful person who is ungrateful.
C) Every successful person is grateful.
D) Every successful person is ungrateful.
E) Every unsuccessful person is grateful.
F) Every unsuccessful person is ungrateful.

Which of the following is the negation of "if $A \subseteq B$, then $A - B = \varnothing$?"

A) If $A \subseteq B$, then $A - B = \varnothing$.
B) If $A \subseteq B$, then $A - B \neq \varnothing$.
C) If $A \not\subseteq B$, then $A - B = \varnothing$.
D) If $A \not\subseteq B$, then $A - B \neq \varnothing$.
E) There are sets A and B where $A \subseteq B$ and $A - B = \varnothing$.
F) There are sets A and B where $A \subseteq B$ and $A - B \neq \varnothing$.
G) There are sets A and B where $A \not\subseteq B$ and $A - B = \varnothing$.
H) There are sets A and B where $A \not\subseteq B$ and $A - B \neq \varnothing$.

Remember that you need to provide a justification for your answers. While it's not required, ideally you should be able to explain both why your answer is correct and why all the other answers are incorrect.

Checkpoint Problem Two: Multiples of Three

In class, we talked a fair amount about odd and even numbers, which arise when dividing numbers in half. This question generalizes the idea of "even" and "odd" to similar terms that arise when dividing by three.

An integer is called a ***multiple of three*** if it can be written as $3k$ for some integer k. An integer is ***congruent to one modulo three*** if it can be written as $3k + 1$ for some integer k, and an integer is ***congruent to two modulo three*** if it can be written as $3k + 2$ for some integer k. For each integer n, exactly one of the following is true (you don't need to prove this):

- n is a multiple of three.
- n is congruent to one modulo three.
- n is congruent to two modulo three.

We'd like you to prove this result:

For every integer n, n is a multiple of three if and only if n^2 is a multiple of three.

To do this, we'll have you prove the following two statements:

For any integer n, if n a multiple of three, then n^2 is a multiple of three.

For any integer n, if n^2 is a multiple of three, then n is a multiple of three.

We've broken this question down into a few parts.

i. Prove the first of these statements with a direct proof.

Not sure how to do that? Take a look at our proof that if n is even, then n^2 is even.

ii. Prove the second of these statements using a proof by contrapositive. Make sure that you state the contrapositive of the statement explicitly before you attempt to prove it.

As a hint, think about using a proof by cases.

iii. Prove, by contradiction, that $\sqrt{3}$ is irrational.

You may want to read over the proof that the square root of two is irrational and use it as a starting point.

The remainder of these problems are due by Friday at 2:30PM.

Problem One: Set Theory Warmup

This question is designed to help you get used to the notation and mathematical conventions surrounding sets. Consider the following sets:

$$A = \{ 1, 2, 3, 4 \}$$
$$B = \{ 2, 2, 2, 1, 4, 3 \}$$
$$C = \{ 1, \{2\}, \{\{3, 4\}\} \}$$
$$D = \{ 1, 3 \}$$
$$E = \mathbb{N}$$
$$F = \{ \mathbb{N} \}$$

Answer each of the following questions and briefly justify your answers. No proofs are necessary.

i. Which pairs of the above sets, if any, are equal to one another?

ii. Is $D \in A$? Is $D \subseteq A$?

iii. What is $A \cap C$? How about $A \cup C$? How about $A \Delta C$?

iv. What is $A - C$? How about $\{A - C\}$? Are those sets equal?

v. What is $|B|$? What is $|E|$? What is $|F|$?

vi. What is $E - A$? Express your answer in set-builder notation.

vii. Is $0 \in E$? Is $0 \in F$?

Problem Two: The Power Set Revisited

In our first lecture, we saw an operation called the power set that, given a set S, produces a set $\wp(S)$ consisting of all the subsets of the set S. Why *didn't* we introduce an operation that, given a set S, produces a set consisting of all the *elements* of S?

Problem Three: Much Ado About Nothing

It can take a bit of practice to get used to the empty set. This problem will ask you to think about a few different sets related to $\varnothing$.

Go to the CS103 website and download the starter files for Problem Set One. Unpack the files somewhere convenient and open up the bundled project. Answer each part of this question by editing the relevant resource files (they're in the `res/` directory). There's information in the top of each of the files about how to represent sets; most importantly, note that to write out the empty set, you should write `{}` rather than using the empty-set symbol. For example, the set $\{\varnothing\}$ would be written as `{{}}`.

i. Edit the file `PartI.object` so that it contains a set equal to $\varnothing \cup \{\varnothing\}$.

ii. Edit the file `PartII.object` so that it contains a set equal to $\varnothing \cap \{\varnothing\}$.

iii. Edit the file `PartIII.object` so that it contains a set equal to $\{\varnothing\} \cup \{\{\varnothing\}\}$.

iv. Edit the file `PartIV.object` so that it contains a set equal to $\{\varnothing\} \cap \{\{\varnothing\}\}$.

v. Edit the file `PartV.object` so that it contains a set equal to $\wp(\wp(\varnothing))$.

vi. Edit the file `PartVI.object` so that it contains a set equal to $\wp(\wp(\wp(\varnothing)))$.

The starter code contains a driver program you can use to see the contents of your files and confirm they're syntactically correct. Submit your answers through GradeScope under "Coding Problems for Problem Set One" by uploading these six files. You're welcome to submit as many times as you'd like.

Problem Four: Set Theory in C++

The C++ standard library contains a type called `std::set` that represents a set of elements, all of which must be of the same type. For example, the type `std::set<int>` represents a set of integers, the type `std::set<std::string>` represents a set of strings, and `std::set<std::set<int>>` is a type representing a set of sets of integers.

There are all sorts of operations you can perform on `std::set`s. For example, here's how you iterate over all the elements of a set:

```
std::set<T> mySet;
for (T elem: mySet) {
    /* … do something with the current element elem … */
}
```

Here's how you check whether a particular value is an element of a set:

```
if (mySet.count(value)) {
    /* … value ∈ mySet … */
} else {
    /* … value ∉ mySet … */
}
```

And, finally, here's how you can get the cardinality of a set:

```
size_t size = mySet.size();
```

Here, the `size_t` type is a type representing a natural number, since sets can't have negative size. (The folks who implemented the C++ standard libraries had a strong discrete math background.)

One of the major differences between the sets that we've been talking about in CS103 and the `std::set` type is that in discrete mathematics, sets can contain anything – numbers, philosophical concepts, recipes, other sets, etc. – but in C++ all objects in a set must have the same type. For the purposes of this problem, we've created a custom C++ type called `Object`. Variables of type `Object` can represent just about anything, so a `std::set<Object>` represents something pretty similar to the sets we've been studying so far.

Some `Object`s are actually just `std::set`s in disguise. If you have an `Object`, you can test whether it's actually a set by using this provided helper function:

```
bool isSet(Object o);
```

This takes in an `Object`, then returns true if that `Object` is actually a set and false otherwise. If you have an `Object` that really is a set, you can convert it to a set by using this helper function:

```
std::set<Object> asSet(Object o);
```

This function takes in an `Object` that you know happens to be a set, then returns the `std::set<Object>` that it actually is.

For example, suppose you have an `Object` that you know is really the set {1, 2, 3, 4}. You could iterate over it using this code:

```
Object reallyASet = /* … */;
for (Object x: asSet(reallyASet)) {
    /* … do something with x … */
}
```

In this problem, we'd like you to demonstrate your understanding of sets and set theory by coding up a number of functions in C++ that operate on sets. In doing so, we hope that you'll solidify your grasp of the distinctions between related concepts in set theory, such as the the ∈ and ⊆ relations and power sets.

(Continued on the next page)

Open the file `SetTheory.cpp` from the starter files. There, you'll find a bunch of stubs of functions that you'll need to implement. The provided starter code contains a test harness you can use to try out your functions. You won't need to modify any of the other C++ files bundled with the starter code.

As with Problem Three, you'll submit the code that you write through GradeScope separately from the rest of the problems on this problem set. The GradeScope autograder will get back to you with feedback about how you're doing on this problem, and you're welcome to submit as many times as you'd like.

i. Implement a function

```
bool isElementOf(Object S, Object T);
```

that takes as input two `Object`s `S` and `T`, then returns whether $S \in T$.

`S` and `T` might not be sets; you'll need to use the `isSet` and `asSet` functions appropriately.

ii. Implement a function

```
bool isSubsetOf(Object S, Object T);
```

that takes as input an object `S` and an object `T`, then returns whether $S \subseteq T$.

`S` and `T` might not be sets; use the `isSet` predicate to check whether the appropriate arguments are sets and `asSet` to get a view of them as sets.

iii. Implement a function

```
bool areDisjointSets(Object S, Object T);
```

that takes as input two objects `S` and `T`, then returns whether `S` and `T` are sets where $S \cap T = \varnothing$. (Two sets with this property are called ***disjoint***.) The input parameters `S` and `T` may or may not be sets, and if they aren't, your function should return false.

iv. Implement a function

```
bool isSingletonOf(Object S, Object T);
```

that takes as input two objects `S` and `T`, then returns whether $S = \{T\}$. Again, `S` and `T` may or may not be sets.

v. Implement a function

```
bool isElementOfPowerSet(Object S, Object T);
```

that takes as input two objects `S` and `T`, then returns whether `S` and `T` are sets and $S \in \wp(T)$. Again, `S` and `T` may or may not be sets.

As a hint, you shouldn't need to write code that computes $\wp(T)$ explicitly. See if you can find a different way to do this.

vi. Implement a function

```
bool isSubsetOfPowerSet(Object S, Object T);
```

that takes as input two objects `S` and `T`, then returns whether `S` and `T` are sets and $S \subseteq \wp(T)$. Again, `S` and `T` may or may not be sets.

vii. Implement a function

```
bool isSubsetOfDoublePowerSet(Object S, Object T);
```

that takes as input two objects `S` and `T`, then returns whether `S` and `T` are sets and $S \subseteq \wp(\wp(T))$. Again, `S` and `T` may or may not be sets.

To submit your work, upload your edited `SetTheory.cpp` file to GradeScope. You'll get immediate feedback on your score from our autograder. (Don't forget to include the files from Problem Three!)

Problem Five: Describing the World in Set Theory

The notation of set theory (e.g. ∪, ∩, ℘, ⊆, ∈, etc.) is a great tool for describing the real world. Answer each of the following questions by writing an expression using set theory notation, but ***without*** using plain English, ***without*** using set-builder notation, ***without*** introducing any new variables, and ***without*** using propositional or first-order logic (which we'll cover next week).

i. Let's have C be the set of US citizens, S the set of people who live in a US state, M be the set of all people eighteen and older, and V be the set of people who are allowed to vote in US presidential elections. Write an expression that says that every US citizen age eighteen and older who lives in a US state can vote in a US presidential election.

*Once you've written up your answer to this problem, take a minute to **type-check** it. As an example, suppose that you have the following answer:*

$$(C \in M) \cap (V \in M)$$

This expression can't possibly be right, and here's one way to see this. The expression $C \in M$ is of type boolean – either $C \in M$ is true or it isn't – and the same is true of $V \in M$. However, the intersection operator ∩ can only be applied to sets. The expression therefore contains a type error: it tries to apply an operator that only works on sets to boolean values.

ii. Suppose you're on an exciting first date. Let Y represent your hobbies and D represent your date's hobbies. Write an expression that says that you have a hobby that your date doesn't have.

You can type-check this answer in a different way. For example, suppose you came up with this expression:

$$Y \cup D$$

Here, Y and D are sets, so it's perfectly fine to write $Y \cup D$, which evaluates to an object of type set. But notice that the statement here asks you to write an expression that says "you have a hobby that your date doesn't have," and that statement is essentially of type boolean (you either do or do not have a hobby your date doesn't have). Therefore. $Y \cup D$ can't possibly be an expression with the right meaning, since the type of the expression (set) doesn't match the type of the statement (boolean).

iii. Tom Stoppard's play *Rosencrantz and Guildenstern are Dead* contains this quote in which the leader of a theater troupe discusses what sorts of plays his group is willing to put on:

> "We're more of the love, blood, and rhetoric school. Well, we can do you blood and love without the rhetoric, and we can do you blood and rhetoric without the love, and we can do you all three concurrent or consecutive. But we can't give you love and rhetoric without the blood. Blood is compulsory."

Let B be the set of all plays involving blood, L be the set of all plays involving love, and R be the set of all plays involving rhetoric. Write an expression for all plays involving at least one of blood, love, and rhetoric which also happen to include blood.

iv. In the Talking Heads song *Crosseyed and Painless*, David Byrne speaks the following lines:

"Facts are simple and facts are straight.
Facts are lazy and facts are late."

Let F be the set of all facts. Let A, B, C, and D represent the set of all things that are simple, straight, lazy, and late, respectively. Write an expression that conveys David Byrne's lyrics in the language of set theory.

v. Let's say that a ***committee*** is a group of people, which we can think of as being represented by the set of people on that committee. Let's have S represent the set of all students at Stanford and let F represent the set of all faculty at Stanford. Write an expression representing the set of all committees you can make from Stanford students and faculty that contain at least one student and at least one faculty member. You can assume no one is both a student and a faculty member.

Something to think about: how would you say "all committees made purely of students?"

Problem Six: Modular Arithmetic

Different numbers can yield the same remainder when divided by some number. For example, the numbers 1, 12, 23, and 34 all leave a remainder of one when divided by eleven. To formalize this relationship between numbers, we'll introduce a relation $\equiv_k$ that, intuitively, indicates that two numbers leave the same remainder when divided by k. For example, we'd say that $1 \equiv_{11} 12$, since both 1 and 12 leave a remainder of 1 when divided by 11, and that $8 \equiv_3 14$, since both 8 and 14 leave a remainder of 2 when divided by 3. To be more rigorous, we'll formally define $\equiv_k$. For any integer k, define $a \equiv_k b$ as follows:

We say that $a \equiv_k b$ if there exists an integer q such that $a - b = kq$

For example, $7 \equiv_3 4$, because $7 - 4 = 3 = 3 \cdot 1$, and $13 \equiv_4 5$ because $13 - 5 = 8 = 4 \cdot 2$. If $x \equiv_k y$, we say that ***x is congruent to y modulo k***, hence the terminology in the checkpoint problem. In this problem, you will prove several properties of modular congruence.

i. Prove that for any integer x and any integer k that $x \equiv_k x$.

Be careful not to assume what you need to prove. Don't start your proof by assuming there's a choice of q where $x - x = kq$ and then solving for q. If you assume there's an integer q where $x - x = kq$, you're already assuming that $x \equiv_k x$! Look at the proofs we did in lecture with odd and even numbers as an example of how to prove that there is a number with a specific property without making any unfounded assumptions.

ii. Prove that for any integers x and y and any integer k that if $x \equiv_k y$, then $y \equiv_k x$.

Keep an eye out for your variable scoping in the above proof. Make sure you introduce the variables x, y, and k before you use them. Are they chosen arbitrarily? Do they represent specific values?

iii. Prove that for any integers x, y, and z and any integer k that if $x \equiv_k y$ and $y \equiv_k z$, then $x \equiv_k z$.

The three properties you have just proven show that modular congruence is an ***equivalence relation***. Equivalence relations show up everywhere in computer science, and we'll talk about them in week three.

Problem Seven: Two Is Irrational?

In lecture, we proved that $\sqrt{2}$ is irrational, and in the checkpoint problem you proved that $\sqrt{3}$ is irrational. Below is a purported proof that $\sqrt{4}$ is irrational:

Theorem: $\sqrt{4}$ is irrational.

Proof: Assume for the sake of contradiction that $\sqrt{4}$ is rational. Then there must exist integers p and q where $q \neq 0$, where $p / q = \sqrt{4}$, and where p and q have no common factors other than 1 and -1.

Starting with $p / q = \sqrt{4}$ and squaring both sides tells us that $p^2 / q^2 = 4$. We can then cross-multiply by q^2 to see that $p^2 = 4q^2$. Since q^2 is an integer and $p^2 = 4q^2$, we see that p^2 is a multiple of four, and therefore that p is a multiple of four. This tells us $p = 4n$ for some integer n.

Since we know that $4q^2 = p^2$ and $p = 4n$, we can do some algebraic substitutions to see that $4q^2 = (4n)^2 = 16n^2$, so $q^2 = 4n^2$. Since n^2 is an integer and $q^2 = 4n^2$, we see that q^2 is a multiple of four, so q is a multiple of four as well. But since both p and q are multiples of four, we see that p and q share a common divisor other than ± 1, contradicting our initial assumption. We have reached a contradiction, so our assumption must have been incorrect. Thus $\sqrt{4}$ is irrational. ■

This proof has to be wrong, because $\sqrt{4} = 2 = {}^2/_1$, so it is indeed rational! What error does this proof make that lets it conclude $\sqrt{4}$ is irrational? Be specific.

The best way to find a flaw in a proof is to find a specific, incorrect claim made in the proof and to explain, concretely, why that claim is incorrect. Also note that your job isn't to try to "fix" the proof by explaining how you'd correct the error. We just want you to convince us you see what's wrong.

Problem Eight: Properties of Sets

Here are some claims about properties of sets. Some of them are true and some of them are false. For each true statement, write a proof that the statement is true. For each false statement, write a ***disproof*** of the statement (take a look at the Proofwriting Checklist for information about how to write a disproof.) You can use any proof techniques you'd like.

i. Prove or disprove: for all sets A, B, and C, if $A \in B$ and $B \in C$, then $A \in C$.

This is your first example of a "prove or disprove" problem. Part of the challenge of approaching a problem like this one is that you'll need to figure out whether or not the statement is even true in the first place, since if it's true you'll want to prove it and if it's false you'll want to disprove it.

Here are two strategies for approaching problems like these. First, try out a lot of examples! You'll want to get a feel for what the symbolic expression above "feels" like in practice. Second, get a sheet of scratch paper and write out both the statement and its negation. One of those statements is true, and your task is to figure out which one it is. Once you have those two statements, think about what you would need to do to prove each of them. In each case, what would you be assuming? What would you need to prove? If you can answer those questions, you can explore both options and seeing which one ends up panning out.

ii. Prove or disprove: for all sets A, B, and C, if $A \subseteq B$ and $A \subseteq C$, then $A \subseteq B \cap C$.

iii. Prove or disprove: for all sets A, B, and C, if $A \subsetneq B$ and $A \subsetneq C$, then $A \subsetneq B \cap C$. (The notation $A \subsetneq B$ says that A is a ***strict subset*** of B, meaning that $A \subseteq B$ and $A \neq B$.)

iv. Prove or disprove: there exists a set A where $\wp(A) = \{A\}$.

v. Prove or disprove: for all sets A and B, if $\wp(A) = \wp(B)$, then $A = B$.

Look back at Wednesday's lecture. What's a good general way to prove that two sets are equal?

Before you turn in these proofs, be sure to read over the Proofwriting Checklist and to go one item at a time through each of your proofs. Here are a few specific things to look for:

- *Make sure that the structures of your proofs match the definitions of the relevant terms. For example, to prove that a set S is a subset of a set T, follow the pattern from lecture: pick an arbitrary $x \in S$, then prove that $x \in T$ by making specific claims about x.*
- *However, avoid restating definitions in the abstract. For example, rather than writing*

  "We know that $S \subseteq T$ if every element of S is an element of T.
  Therefore, since we know that $A \subseteq B$ and $x \in A$, we see that $x \in B$."

  instead remove that first sentence and just write something like this:

  "Since $x \in A$ and $A \subseteq B$, we see that $x \in B$."

  Whoever is reading your proof knows all the relevant definitions. They're more interested in seeing ***how those definitions interact with one another*** *than* ***what those definitions are****.*
- *Make sure you clearly indicate what each variable means and whether it's chosen arbitrarily or chosen to have a specific value. For example, in your answers, if you refer to variables like A, B, or C, you should clearly indicate whether they're chosen arbitrarily or refer to specific values.*
- *If you're talking about an arbitrary set A, it's often tempting to try to list off the elements of A by writing something like $A = \{ x_1, x_2, \ldots, x_n \}$. The problem with this approach is that by writing $A = \{ x_1, x_2, \ldots, x_n \}$, you're implicitly saying that the set A is finite, since you're claiming it only has n elements in it. This is a problem if A is an infinite set. In fact, if A is infinite, because of Cantor's theorem you can't necessarily even write $A = \{ x_1, x_2, x_3, \ldots \}$, since you might run out of natural numbers with which to name the elements of A without having listed all of them!*

Problem Nine: Piano Tuning

At a first glance, irrational numbers can seem like a purely mathematical idea without any practical applications, but, surprisingly, irrational numbers have real-world implications.

Prove that $\sqrt[12]{2}$, the twelfth root of two, is irrational. Interestingly, this result means that it's impossible to tune a piano such that every half step, perfect fifth, perfect fourth, and octave are all correct. If you're curious why this is, check out this great Minute Physics video about the different ratios that arise in music and how the twelfth root of two relates.

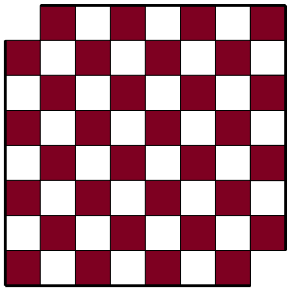
As a hint, do *not* attempt to prove this result by starting with the proof that $\sqrt{2}$ is irrational and making appropriate modifications – that will get really messy, really fast. Instead, see if you can prove the following intermediary result, and build your proof around it:

If $\sqrt[12]{2}$ is rational, then $\sqrt{2}$ is rational.

Some notes on this problem:

- *For the purposes of CS103, we've defined a rational number as a number r that can be written as $^p/_q$ for integers p and q where $q \neq 0$. For example, if you wanted to show that 1.64 is a rational number, you could just remark that it can be written as $^{164}/_{100}$ without any further elaboration, even though 164 and 100 both share 4 as a common factor. While you can always write rational numbers as a ratio of numbers with no common factors, that isn't officially part of the definition.*
- *If you want to use any properties of the rational numbers that we did not prove in class (for example, that the sum of two rational numbers is rational), you should prove those results first.*
- *You may want to read the Mathematical Prerequisites handout for a refresher on higher roots.*

Problem Ten: Tiling a Chessboard



Suppose you have a standard 8 × 8 chessboard with two opposite corners removed, as shown here. In the course notes (pages 60 - 61), there's a proof that it's impossible to tile this chessboard using 2 × 1 dominoes. This question considers what happens if you try to tile the chessboard using ***right triominoes***, L-shaped tiles that look like this:

i. Prove that it's impossible to tile an 8×8 chessboard missing two opposite corners with right triominoes.

ii. ***Prove or disprove:*** there is a natural number $n \geq 3$ where it's possible to tile an $n \times n$ chessboard missing two corners with right triominoes.

Part (ii) of this problem is another example of a prove-or-disprove type problem, and you've had plenty of practice with that from Problem Eight. So approach it the same way – grab a sheet of scratch paper, write out both the statement and its negation, work out what it is that you'd do if you wanted to prove each of those statements is true, and try a lot of examples. Explore both options and see what you find! As with part (i), drawing pictures would be a great strategy here. If you can successfully tile a board of a given size, great! You're done. If you keep running into trouble, perhaps you can spot a pattern about why that is and use that as the basis of a disproof.

Problem Eleven: Yablo's Paradox

A ***logical paradox*** is a statement that results in a contradiction regardless of whether it's true or false. One of the simplest paradoxes is the ***Liar's paradox***, which is the following:

This statement is false.

If the above statement is true, then by its own admission, it must be false – a contradiction! On the other hand, if the above statement is false, then the statement "This statement is false" is false, and therefore the statement "This statement is false" is true – a contradiction! Since this statement results in a contradiction regardless of whether it's true or false, it's a paradox.

Paradoxes often arise as a result of self-reference. In the Liar's Paradox, the paradox arises because the statement directly refers to itself. However, it's not the only paradox that can arise from self-reference. This problem explores a paradox called ***Yablo's paradox***.

Consider the following collection of infinitely many statements numbered S_0, S_1, S_2, …, where there is a statement S_n for each natural number n. These statements are ordered in a list as follows:

(S_0): All statements in this list after this one are false.
(S_1): All statements in this list after this one are false.
(S_2): All statements in this list after this one are false.
…

More generally, for each $n \in \mathbb{N}$, the statement (S_n) is

(S_n): All statements in this list after this one are false.

Surprisingly, the interplay between these statements makes every statement in the list a paradox.

i. Prove that every statement in this list is a paradox.

Some hints on this problem:

- *We've asked you to prove a universal statement (every element in this list is a paradox). What is the general template for proving a universal statement?*
- *Split your proof into two parts. First, show you get a contradiction if any of the statements in the list are true. Then, show you get a contradiction if any of the statements in the list are false.*
- *You should implicitly assume, as we've been doing in class, that every statement is either true or false. You don't need to worry about statements that are neither true nor false or statements that are simultaneously true and false.*
- *How do you negate a universally-quantified statement?*

Now, consider the following modification to this list. Instead of infinitely many statements, suppose that there are "only" 10,000,000,000 statements. Specifically, suppose we have these statements:

(T_0): All statements in this list after this one are false.
(T_1): All statements in this list after this one are false.
(T_2): All statements in this list after this one are false.
…
($T_{9,999,999,999}$): All statements in this list after this one are false.

There's still a lot of statements here, but not infinitely many of them. Interestingly, these statements are all perfectly consistent with one another and do not result in any paradoxes.

ii. For each statement in the above list, determine whether it's true or false and explain why your choices are consistent with one another.

Going forward, don't worry about paradoxical statements in CS103. We won't talk about any more statements like these. ☺

Optional Fun Problem: The Mouse and the Cheese (1 Point Extra Credit)[*]

On each problem set, we'll provide an optional fun problem for extra credit. When we compute final grades at the end of the quarter, we compute the grading curve without any extra credit factored in, then recompute grades a second time to factor in extra credit. This way, you're not at any disadvantage if you decide not to work through these problems. If you do complete the extra credit problems, you may get a slight boost to your overall grade. As a matter of course policy, we don't provide any hints on the extra credit problems – after all, they're supposed to be challenge problems! However, we're happy to chat about them after the problem sets come due.

Suppose that you have a 3" × 3" × 3" cube of cheese subdivided into twenty-seven 1" × 1" × 1" smaller cubes of cheese. A mouse wants to eat the entire cube of cheese and does so as follows: she first picks any small cube to eat first, then moves to an adjacent small cube of cheese (i.e. a cube that shares a face with the cube that was just eaten) to eat next, then repeats this process.

Prove that the mouse can't eat the centermost cube of cheese last.

[*] Adapted from Problem 4E of *A Course in Combinatorics, Second Edition* by Lint and Wilson.

Problem Set 2

This second problem set explores mathematical logic. We've chosen the questions here to help you get a more nuanced understanding for what first-order logic statements mean (and, importantly, what they don't mean) and to give you a chance to practice your proofwriting. By the time you've completed this problem set, we hope that you have a much better grasp of mathematical logic and how it can help improve your proofwriting structure.

Because of the Dr. Martin Luther King, Jr. holiday, there are a few problems on this problem set that reference concepts we will cover in Monday's lecture. Those questions are clearly marked as such, and anything that doesn't explicitly warn about this can be completed purely using the material up to and including the lecture when this problem set is released. If you'd like to get an early jump on the remaining problems, visit the course website and check out the Guide to Negations and Guide to First-Order Translations, which will provide an overview of the relevant skills.

Before attempting this problem set, we recommend that you do the following:

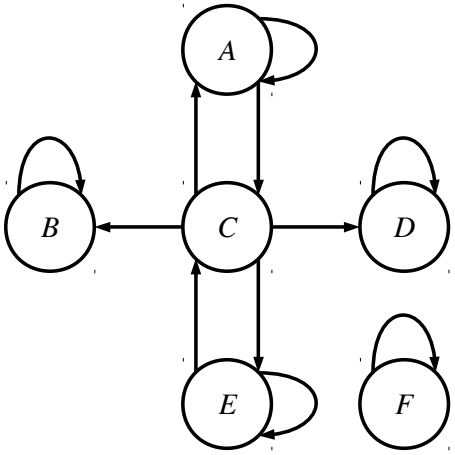
- Familiarize yourself with the online Truth Table Tool and play around with it a bit to get a feel for the propositional connectives.
- Read the online "Guide to Negations" and "Guide to First-Order Translations" (either on Monday, or when this problem set goes out if you want to get a jump on things).
- Read Handout #14, "First-Order Translation Checklist," to get a better sense for common errors in first-order logic translations and how to avoid them. ***We will be running these checklists on your translations, so please be sure to double-check your work before submitting!***

Checkpoint Questions Due Monday, January 22nd at 2:30PM.
Remaining Questions Due Friday, January 26th at 2:30PM.

This week's checkpoint problem comes in the shape of a Google Form, which you can find online using this link. Unlike the other assignments for this quarter, you will not submit this assignment on Grade-Scope, and you'll get feedback about incorrect answers as you go.

Checkpoint Problem: Interpersonal Dynamics (2 Points if Submitted)

The diagram to the right represents a set of people named *A*, *B*, *C*, *D*, *E*, and *F*. If there's an arrow from a person *x* to a person *y*, then person *x* loves person *y*. We'll denote this by writing *Loves*(*x*, *y*). For example, in this picture, we have *Loves*(*C*, *D*) and *Loves*(*E*, *E*), but not *Loves*(*D*, *A*).



There are no "implied" arrows anywhere in this diagram. For example, even though *A* loves *C* and *C* loves *E*, the statement *Loves*(*A*, *E*) is false because there's no direct arrow from *A* to *E*. Similarly, even though *C* loves *D*, the statement *Loves*(*D*, *C*) is false because there's no arrow from *D* to *C*.

At the linked Google Form, you'll find a series of first-order logic formulas about this particular group of people. For each of those formulas, determine whether it's true or false about this group. No justification is necessary.

The remainder of these problems should be completed and submitted through GradeScope by Friday at 2:30PM.

Problem One: Implies and False

Although propositional logic has many different connectives, it turns out that any formula in propositional logic can be rewritten as an equivalent propositional formula that uses only the ¬, ∧, and ⊤ connectives. (You don't need to prove this). In this problem, you will prove a different result: every formula in propositional logic can be rewritten as an equivalent logical formula purely using the → and ⊥ connectives.

i. Find a formula that's logically equivalent to $\neg p$ that uses only the variable p and the → and ⊥ connectives. No justification is necessary.

ii. Find a formula that's logically equivalent to ⊤ that uses only the → and ⊥ connectives. No justification is necessary.

iii. Find a formula that's logically equivalent to $p \wedge q$ that uses only the variables p and q and the → and ⊥ connectives. No justification is necessary.

As a hint, what happens if you negate an implication?

Since you can express ¬, ∧, and ⊤ using just → and ⊥, every possible formula in propositional logic can be expressed using purely the → and ⊥ connectives. Nifty!

Problem Two: Ternary Conditionals

Many programming languages support a *ternary conditional operator*. For example, in C, C++, and Java, the expression x `?` y `:` z means "evaluate the boolean expression x. If it's true, the entire expression evaluates to y. If it's false, the entire expression evaluates to z."

In the context of propositional logic, we can introduce a new ternary connective `?:` such that p `?` q `:` r means "if p is true, the connective evaluates to the truth value of q, and otherwise it evaluates to the truth value of r."

i. Based on this description, write a truth table for the `?:` connective.

ii. Find a propositional formula equivalent to p `?` q `:` r that does not use the `?:` connective. Justify your answer by providing a truth table for your new formula.

It turns out that it's possible to rewrite any formula in propositional logic using only `?:`, ⊤, and ⊥. The rest of this question will ask you to show this.

iii. Find a formula equivalent to $\neg p$ that does not use any connectives besides `?:`, ⊤, and ⊥. No justification is necessary.

iv. Find a formula equivalent to $p \rightarrow q$ that does not use any connectives besides `?:`, ⊤, and ⊥. No justification is necessary.

Since all remaining connectives can be written purely in terms of ¬ and →, any propositional formula using the seven standard connectives can be rewritten using only the three connectives `?:`, ⊤, and ⊥.

The fact that all propositional formulas can be written purely in terms of `?:`, ⊤, and ⊥ forms the basis for the *binary decision diagram*, a data structure for compactly encoding propositional formulas. Binary decision diagrams have applications in program optimization, graph algorithms, and computational complexity theory. Take CS166 or CS243 for more info!

Problem Three: Executable Logic

There's a great quote from Douglas Adams about programming a computer:

> "[I]f you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your mind. [… T]hat's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've learned something about it yourself."

To help you get a better feeling for what first-order logic formulas mean – and, importantly, what they *don't* mean – we'd like you to write a series of short programs that determine whether particular first-order logic formulas are true about specific worlds. Visit the CS103 website and download the starter project for Problem Set Two. Open `ExecutableLogic.cpp`, where you'll find six function stubs. The first-order formulas in this problem deal with sets of people who may or may not be happy and who may or may not love one another. Each person is represented as an object of type `Person`, and we've provided the following predicates functions to you:

```
bool happy(Person p)
bool loves(Person p1, Person p2)
```

These functions are written in lower-case, since that's the established C++ convention. The starter files contain a program that visualizes different groups of people and shows how each of your functions evaluates. We strongly recommend using this to check your work as you go.

i. Consider the following first-order logic formula, where P is a set of people:

$$\exists x \in P.\ Happy(x)$$

Write C++ code for a function

```
bool isFormulaTrueFor_partI(std::set<Person> P)
```

that accepts as input a set of people P and returns whether the above formula is true for that particular set of people.

ii. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ Happy(x)$$

iii. Repeat the above exercise with this first-order logic formula:

$$\exists x \in P.\ (Happy(x) \wedge Loves(x, x))$$

iv. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ (Happy(x) \rightarrow Loves(x, x))$$

v. Repeat the above exercise with this first-order logic formula:

$$\forall x \in P.\ (Happy(x) \rightarrow \exists y \in P.\ (Happy(y) \wedge \neg Loves(x, y)))$$

It's a lot easier to write code for this one if you use a helper function.

vi. Repeat the above exercise with this first-order logic formula:

$$\exists x \in P.\ (Happy(x) \leftrightarrow \forall y \in P.\ (Loves(x, y)))$$

As in Problem Set One, you're welcome to submit your answers to this question as many times as you'd like. To submit your work for this problem, upload the file `ExecutableLogic.cpp`.

Problem Four: First-Order Negations

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

For each of the first-order logic formulas below, find a first-order logic formula that is the negation of the original statement. Your final formula must not have any negations in it except for direct negations of predicates. For example, the negation of the formula $\forall x.\ (P(x) \to \exists y.\ (Q(x) \wedge R(y)))$ could be found by pushing the negation in from the outside inward as follows:

$$\neg(\forall x.\ (P(x) \to \exists y.\ (Q(x) \wedge R(y))))$$
$$\exists x.\ \neg(P(x) \to \exists y.\ (Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \neg\exists y.\ (Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \forall y.\ \neg(Q(x) \wedge R(y)))$$
$$\exists x.\ (P(x) \wedge \forall y.\ (Q(x) \to \neg R(y)))$$

Show every step of the process of pushing the negation into the formula (along the lines of what is done above), and *please preserve the indentation from the original formula* as you go. You don't need to formally prove that your negations are correct.

We strongly recommend reading over the Guide to Negations before starting this problem.

i. $\exists p.\ (\mathit{Problem}(p) \wedge$
   $\forall g.\ (\mathit{Program}(g) \to \neg\mathit{Solves}(g, p))$
   $)$

ii. $\forall x \in \mathbb{R}.\ \forall y \in \mathbb{R}.\ (x < y \to$
   $\exists q \in \mathbb{Q}.\ (x < q \wedge q < y)$
   $)$

iii. $(\forall x.\ \forall y.\ \forall z.\ (R(x, y) \wedge R(y, z) \to R(x, z))) \quad \to \quad (\forall x.\ \forall y.\ \forall z.\ (R(y, x) \wedge R(z, y) \to R(z, x)))$

iv. $\forall x.\ \exists S.\ (\mathit{Set}(S) \wedge$
   $\forall z.\ (z \in S \leftrightarrow z = x)$
   $)$

v. $\forall k.\ (\mathit{SixClique}(k) \to$
   $\exists t.\ (\mathit{Triangle}(t, k) \wedge$
   $(\forall e.\ (\mathit{Edge}(e, t) \to \mathit{Red}(e)) \vee \forall e.\ (\mathit{Edge}(e, t) \to \mathit{Blue}(e)))$
   $)$
   $)$

Problem Five: Vacuous Truths

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

A statement is called ***vacuously true*** if either

- it's an implication of the form $P \to Q$ where P is false,
- it's a universally-quantified implication $\forall x.\ (P(x) \to Q(x))$ where $P(x)$ is never true, or
- it's a universally-quantified implication $\forall x \in S.\ Q(x)$ where S is the empty set.

These statements are true *by definition*. In the first case, the truth table for the $\to$ connective says the implication is true when the antecedent is false, and in the second and third cases we just define these sorts of statements to be true.

You might be wondering why exactly this is the case.

i. Negate the propositional formula $P \to Q$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why when $P \to Q$ is vacuously true, its negation is false.

ii. Negate the first-order formula $\forall x.\ (P(x) \to Q(x))$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why if the original formula is vacuously true, then its negation is false.

iii. Negate the first-order formula $\forall x \in S.\ Q(x)$ and push the negations as deep as possible. Now, look at the resulting formula. Explain why if the original formula is vacuously true, then its negation is false.

Your answers to parts (i), (ii), and (iii) of this problem give another justification for vacuous truths. The three classes of statements given above have false negations in the indicated cases, and therefore to keep everything consistent we choose to define them as true.

iv. Now, look back over the code you wrote for parts (ii), (iv), and (v) of the Executable Logic problem. In the course of writing those functions, you should not have needed to handle vacuous truths by adding in extra code to specifically check whether the group in question is empty. (If you did, see if you can go back and remove it!) Briefly explain why it's not necessary to handle these cases explicitly and why your code will correctly handle empty inputs without singling this case out.

Your answer to part (iv) of this problem gives a different intuition for why statements would be vacuously true – vacuous truth naturally follows from how we might think about checking whether a universally-quantified formula would be true.

Problem Six: This, But Not That

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to Negations, which covers all the skills you'll need.)

Below is a series of pairs of statements about a group of people. For each pair, come up with a single group of people where the first statement is true about that group of people and the second statement is false about that group of people. To submit your answers, edit the files `ThisButNotThatI.people`, `ThisButNotThatII.people`, etc. in the `res/` directory of the starter files for this assignment with a description of those groups. There's a description in each of those files of how to specify a group of people.

The starter file `ThisButNotThat.cpp` we've provided you contain stubs of ten functions representing the ten statements here (the five "this" statements and the five "that" statements). You may ***optionally*** implement those functions in the course of solving this problem so that you can test your worlds, but you are not required to do so. You'll just be graded on the groups you submit.

	Make this statement true…	***… and this statement false.***
i.	$\forall y \in P.\ \exists x \in P.\ Loves(x, y)$	$\exists x \in P.\ \forall y \in P.\ Loves(x, y)$
ii.	$\forall x \in P.\ (Happy(x) \lor Loves(x, x))$	$(\forall x \in P.\ Happy(x)) \lor (\forall x \in P.\ Loves(x, x))$
iii.	$(\exists x \in P.\ Happy(x)) \land (\exists x \in P.\ Loves(x, x))$	$\exists x \in P.\ (Happy(x) \land Loves(x, x))$
iv.	$(\forall x \in P.\ Happy(x)) \to (\forall y \in P.\ Loves(y, y))$	$\forall x \in P.\ \forall y \in P.\ (Happy(x) \to Loves(y, y))$
v.	$\exists x \in P.\ (Loves(x, x) \to \forall y \in P.\ (Loves(y, y))$)	$(\forall x \in P.\ Loves(x, x)) \lor (\forall x \in P.\ \neg Loves(x, x))$

As a hint, if you want to make a statement false, make its negation true.

To submit your answer, upload the five `.people` files you edited in the course of solving this problem to GradeScope, and don't forget to also include the `ExecutableLogic.cpp` file from earlier in this problem set!

Problem Seven: Translating into Logic

(We will cover the material necessary to solve this problem on Monday. You can also read over the Guide to First-Order Translations, which covers all the skills you'll need.)

In each of the following, write a statement in first-order logic that expresses the indicated sentence. Your statement may use any first-order construct (equality, connectives, quantifiers, etc.), but you ***must*** only use the predicates, functions, and constants provided. You do not need to provide the simplest formula possible, though we'd appreciate it if you made an effort to do so. ☺ We ***highly*** recommend reading the Guide to First-Order Logic Translations before starting this problem.

i. Given the predicate

Natural(x), which states that x is an natural number

and the functions

$x + y$, which represents the sum of x and y, and
$x \cdot y$, which represents the product of x and y

write a statement in first-order logic that says "for any $n \in \mathbb{N}$, n is even if and only if n^2 is even."

Try translating this statement assuming you have a predicate Even(x). Then, rewrite your solution without using Even(x). Numbers aren't a part of FOL, so you can't use the number 2 in your solution.

ii. Given the predicates

Person(p), which states that p is a person;
Kitten(k), which states that k is a kitten; and
HasPet(o, p), which states that o has p as a pet,

write an FOL statement that says "someone has exactly two pet kittens and no other pets."

Make sure your formula requires that the person have exactly two pet kittens; look at the lecture example of uniqueness as a starting point. Good questions to ask – is your formula false if everyone has exactly one pet kitten? Is it false if everyone has exactly three pet kittens?

iii. The ***axiom of pairing*** is the following statement: given any two distinct objects x and y, there's a set containing x and y and nothing else. Given the predicates

$x \in y$, which states that x is an element of y, and
Set(S), which states that S is a set,

write a statement in first-order logic that expresses the axiom of pairing.

iv. Given the predicates

$x \in y$, which states that x is an element of y, and
Set(S), which states that S is a set,

write a statement in first-order logic that says "every set has a power set."

As a warm-up, solve this problem assuming you have a predicate $X \subseteq Y$ that says that X is a subset of Y. Once you have that working, see if you can solve the full version of this problem.

v. Given the predicates

Lady(x), which states that x is a lady;
Glitters(x), which states that x glitters;
SureIsGold(x, y), which states that x is sure that y is gold;
Buying(x, y), which states that x buys y; and
StairwayToHeaven(x), which states that x is a Stairway to Heaven;

write a statement in first-order logic that says "there's a lady who's sure all that glitters is gold, and she's buying a Stairway to Heaven."

Problem Eight: Hereditary Sets

Let's begin with a fun little definition:

A set S is called a ***hereditary set*** if all its elements are hereditary sets.

This definition might seem strange because it's self-referential – it defines hereditary sets in terms of other hereditary sets! However, it turns out that this is a perfectly reasonable definition to work with, and in this problem you'll explore some properties of these types of sets.

i. Given the self-referential nature of the definition of hereditary sets, it's not even clear that hereditary sets even exist at all! As a starting point, prove that there is at least one hereditary set.

ii. Prove that if S is a hereditary set, then $\wp(S)$ is also a hereditary set.

After you've written up a draft of your proofs, take a minute to read over them and apply the criteria from the Proofwriting Checklist. Here are a few specific things to watch out for:

- *If you want to prove in part (ii) that a set T is a hereditary set, you need to prove the statement "every element of T is a hereditary set." That's a universally-quantified statement. If you're proving it via a direct proof, you'll probably need to pick some arbitrary element $x \in T$, then prove that x is a hereditary set by making specific claims about the variable x. Read over your proof and make sure that (1) you've introduced a new variable to refer to some arbitrarily-chosen element of T and that (2) you're making specific claims about the variable x, rather than talking in general about how elements of T behave. You may need to introduce multiple variables in the course of your proofs.*
- *A common mistake we see people make when they're just getting started is to restate definitions in the abstract in the middle of a proof. For example, we commonly see people say something like "since $A \subseteq B$, we know that every element of A is an element of B." When you're writing a proof, you can assume that whoever is reading your proof is familiar with the definitions of relevant terms, so statements like the one here that just restate a definition aren't necessary. Instead of restating definitions, try to apply those definitions. A better sentence would be something to the effect of "Since $x \in A$ and $A \subseteq B$, we see that $x \in B$," which uses the definition to conclude something about a specific variable rather than just restating the definition.*
- *Although we've just introduced first-order logic as a tool for formalizing definitions and reasoning about mathematical structures, the convention is to **not** use first-order logic notation (connectives, quantifiers, etc.) in written proofs. In a sense, you can think of first-order logic as the stage crew in the theater piece that is a proof – it works behind the scenes to make everything come together, but it's not supposed to be in front of the audience. Make sure that you're still writing in complete sentences, that you're not using symbols like $\forall$ or $\rightarrow$ in place of words like "for any" or "therefore," etc.*

Problem Nine: Symmetric Latin Squares

A ***Latin square*** is an $n \times n$ grid filled with the numbers 1, 2, 3, …, n such that every number appears in every row and every column exactly once. For example, the following are Latin squares:

1	2	3
3	1	2
2	3	1

4	2	1	3
1	3	2	4
3	1	4	2
2	4	3	1

1	3	5	2	4
2	4	1	3	5
3	5	2	4	1
4	1	3	5	2
5	2	4	1	3

A ***symmetric Latin square*** is a Latin square that is symmetric across the main diagonal (the one from the upper-left corner to the lower-right corner). That is, the elements at positions (i, j) and (j, i) are always the same. For example:

1	2	3
2	3	1
3	1	2

4	2	3	1
2	3	1	4
3	1	4	2
1	4	2	3

1	2	3	4	5
2	4	5	3	1
3	5	2	1	4
4	3	1	5	2
5	1	4	2	3

Prove that in any $n \times n$ symmetric Latin square where n is odd, every number 1, 2, 3, …, n must appear at least once on the main diagonal.

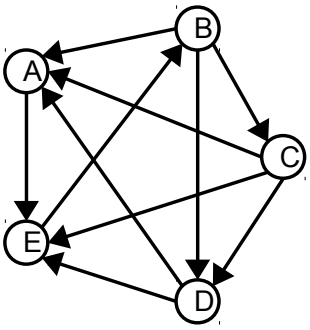
As a hint: split the Latin square into three regions – the main diagonal and the two regions above and below the main diagonal. Then think about how often each element appears in each group.

Once you've written up a draft of your proof for this problem, try out the following exercise as a way of checking your work. If you look at the sample Latin squares shown above, you can see that the result given above is not true in the case where the Latin square isn't symmetric, and it's also not true in the case where the square has even size. As a result, if your proof does not specifically use the fact that the Latin square is symmetric and does not specifically use the fact that the Latin square has odd size, it has to contain an error somewhere. Otherwise, you could change the setup to the problem and end up with a proof of an incorrect result. (Do you see why this is?) So go back over your proof and ask yourself – where, specifically, am I making reference to the fact that the Latin square is symmetric? Where, specifically, am I making reference to the fact that the Latin square has odd size? And why would my proof break down if I eliminated either of those references?

Going forward, this approach to checking your proofs – perturbing the starting assumptions and seeing where your logic breaks down – is an excellent way to smoke out any underlying logic errors.

Problem Ten: Tournament Winners

Here's one more problem to help you practice your proofwriting. It's a classic CS103 problem, and we hope that you enjoy it!



A ***tournament*** is a contest among n players. Each player plays a game against each other player, and either wins or loses the game (let's assume that there are no draws). We can visually represent a tournament by drawing a circle for each player and drawing arrows between pairs of players to indicate who won each game. For example, in the tournament to the left, player A beat player E, but lost to players B, C, and D.

A ***tournament winner*** is a player in a tournament who, for each other player, either won her game against that player, or won a game against a player who in turn won his game against that player (or both). For example, in the graph on the left, players B, C, and E are tournament winners. However, player D is ***not*** a tournament winner, because he neither beat player C, nor beat anyone who in turn beat player C. Although player D won against player E, who in turn won against player B, who then won against player C, under our definition player D is ***not*** a tournament winner. *(Make sure you understand why!)*

i. Let T be an arbitrary tournament and p be any player in that tournament. Prove the following statement: if p won more games than anyone else in T or is tied for winning the greatest number of games, then p is a tournament winner in T.

*This problem is a lot easier to solve if you draw the right picture. Something to think about: what happens if a player p **isn't** a winner in a tournament? What would that mean about player p? And finally, be careful not to make broad claims about tournaments or tournament structures without first proving them!*

A corollary of the result you proved in part (i) is that every tournament with at least one player must have at least one tournament winner – you can always pick someone who won the most games or is tied for winning the most games. However, note that you can win a tournament without necessarily winning the most games – for example, player E in the above example tournament is a winner, though player E only won a single game!

Whenever you prove a new mathematical result (in this case, every tournament with at least one player has at least one winner), it's useful to ask how "resilient" that result is. In other words, if we relax our definition of a tournament a little bit, can we still necessarily guarantee that we can always find a winner? The answer is no, and in fact, even slightly weakening the definition of a tournament invalidates this result.

Let's introduce one more definition. A ***pseudotournament*** is like a tournament, except that exactly one pair of people don't play a game against one another.

ii. Prove that for any $n \geq 2$, there's a pseudotournament P with n players and no tournament winners.

Think about the structure of what you're being asked to prove here. There's both a universal and an existential component to this statement. How do you prove an existential statement?

Optional Fun Problem: Insufficient Connectives (1 Point Extra Credit)

On some of the earlier problems on this problem set, you that every propositional logic formula could be written in terms of just the $\neg$, $\wedge$, and $\top$. You also saw that you could use just $\rightarrow$ and $\bot$, or alternatively that you could just use `?:`, $\top$, and $\bot$. Interestingly, you *cannot* express every possible propositional logic formula using just the $\leftrightarrow$ and $\bot$ connectives. Prove why not.

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:

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

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 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 Cov($<$)? To provide your answer, fill in the blank below, then briefly justify your answer:

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

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!

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 $\text{Cov}(R)$ 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 $\text{Cov}(R)$. (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.

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:

$$pFq \quad \text{if} \quad p.x < q.x \ \wedge \ p.y < q.y$$

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:

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

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

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:

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

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

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:

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

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.

$$p \sim_F q \quad \text{if} \quad \_\_\_\_\_$$
$$p \sim_G q \quad \text{if} \quad \_\_\_\_\_$$
$$p \sim_H q \quad \text{if} \quad \_\_\_\_\_$$

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 `Pixel`s. 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.

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

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:

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

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:*

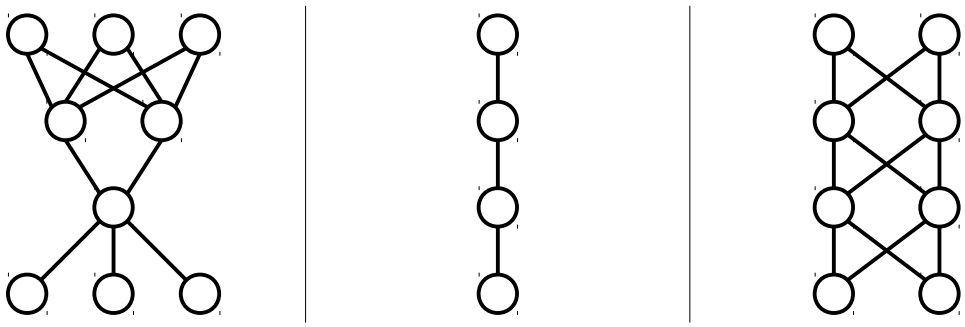
$$\textit{If } aRb \textit{ and } b \sim_R c\textit{, then } aRc$$

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.

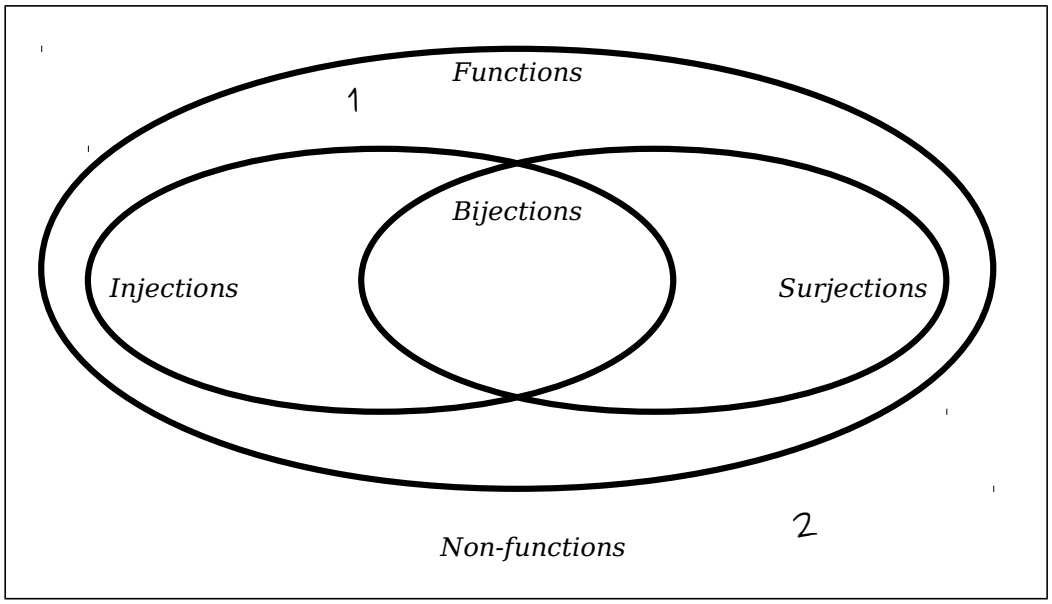
Here are some examples of what these might look like:



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.

Problem Set 4

This fourth problem set explores set cardinality and graph theory. It serves as tour of the infinite (through set theory) and the finite (through graphs and their properties) and will give you a better sense for how discrete mathematical structures connect across these domains. Plus, you'll get to see some pretty pictures and learn about why all this matters in the first place. ☺

We know you have the midterm coming up on Monday, so we've tried to make this checkpoint problem a lot shorter than the one for PS3. We hope it solidifies the relevant concepts without diverting your attention from the exam.

Good luck, and have fun!

Checkpoint due Monday, February 5th at 2:30PM

Remaining problems due Friday, February 9th at 2:30PM

Checkpoint Problem: A Really Simple Bijection? (2 Points if Submitted)

Consider the function $f : \mathbb{N} \to \mathbb{Z}$ defined as $f(n) = n$.

i. Prove that f is not a bijection.

Write out the formal definition of a bijection. If you want to show that f is not a bijection, what do you need to prove?

Below is a purported proof that f is a bijection:

> ***Theorem:*** Let $f : \mathbb{N} \to \mathbb{Z}$ be defined as $f(n) = n$. Then f is a bijection.
>
> ***Proof:*** In lecture, we proved that $|\mathbb{N}| = |\mathbb{Z}|$. Since the sets $\mathbb{N}$ and $\mathbb{Z}$ have the same cardinality, we know that every function between them must be a bijection. In particular, this means that f must be a bijection, as required. ■

This proof has to be incorrect, since, as you proved in part (i), f isn't a bijection.

ii. What's wrong with this proof? Justify your answer.

Make sure you're rock-solid on your answer to part (ii) of this problem – what we've shown above is a very common mistake we see people make when writing proofs about cardinality!

Problem One: Set Cardinalities

If A and B are sets, the ***Cartesian product*** of A and B, denoted $\boldsymbol{A \times B}$, is the set

$$\{ (x, y) \mid x \in A \wedge y \in B \}.$$

Intuitively, $A \times B$ is the set of all ordered pairs you can make by taking one element from A and one element from B, in that order. For example, the set $\{1, 2\} \times \{u, v, w\}$ is

$$\{ (1, u), (1, v), (1, w), (2, u), (2, v), (2, w) \}.$$

For the purposes of this problem, let's let ★ and ☺ denote two arbitrary objects where ★ ≠ ☺. Over the course of this problem, we're going to ask you to prove that $|\mathbb{N} \times \{★, ☺\}| = |\mathbb{N}|$.

i. Draw a picture showing a way to pair off the elements of $\mathbb{N} \times \{★, ☺\}$ with the elements of $\mathbb{N}$ so that no elements of either set are uncovered or paired with multiple elements.

You might want to draw some pictures of the set $\mathbb{N} \times \{★, ☺\}$ so that you can get a better visual intuition.

ii. Based on the picture you came up with in part (i), define a bijection $f : \mathbb{N} \times \{★, ☺\} \to \mathbb{N}$. The inputs to this function will be elements of $\mathbb{N} \times \{★, ☺\}$, so you can define your function by writing

$$f(n, x) = \_\_\_\_\_$$

where $n \in \mathbb{N}$ and $x \in \{★, ☺\}$.

In defining this function, you cannot assume ★ or ☺ are numbers, since they're arbitrary values out of your control. See if you can find a way to define this function that doesn't treat ★ and ☺ algebraically.

iii. Prove that the function you came up with in part (ii) is a bijection.

The result you've proved here essentially shows that $2\aleph_0 = \aleph_0$. Isn't infinity weird?

Problem Two: Understanding Diagonalization

Proofs by diagonalization are tricky and rely on nuanced arguments. In this problem, we'll ask you to review the formal proof of Cantor's theorem to help you better understand how it works. (Please read the Guide to Cantor's Theorem before attempting this problem.)

i. Consider the function $f : \mathbb{N} \to \wp(\mathbb{N})$ defined as $f(n) = \varnothing$. Trace through our formal proof of Cantor's theorem with this choice of f in mind. In the middle of the argument, the proof defines some set D in terms of f. Given that $f(n) = \varnothing$, what is that set D? Provide your answer without using set-builder notation. Is it clear why $f(n) \neq D$ for any $n \in \mathbb{N}$?

Make sure you can determine what the set D is both by using the visual intuition behind Cantor's theorem and by symbolically manipulating the formal definition of D given in the proof.

ii. Let f be the function from part (i). Find a set $S \subseteq \mathbb{N}$ such that $S \neq D$, but $f(n) \neq S$ for any $n \in \mathbb{N}$. Justify your answer. This shows that while the diagonalization proof will always find *some* set D that isn't covered by f, it won't find *every* set with this property.

iii. Repeat part (i) of this problem using the function $f : \mathbb{N} \to \wp(\mathbb{N})$ defined as

$$f(n) = \{ m \in \mathbb{N} \mid m \geq n \}$$

Now what do you get for the set D? Is it clear why $f(n) \neq D$ for any $n \in \mathbb{N}$?

iv. Repeat part (ii) of this problem using the function f from part (iii).

Problem Three: Simplifying Cantor's Theorem?

In our proof of Cantor's theorem, we proved that $|S| \neq |\wp(S)|$ by using a diagonal argument. Below is a purported proof that $|S| \neq |\wp(S)|$ that doesn't use a diagonal argument:

> ***Theorem:*** If S is a set, then $|S| \neq |\wp(S)|$.
>
> ***Proof:*** Let S be any set and consider the function $f : S \to \wp(S)$ defined as $f(x) = \{x\}$. To see that this is a valid function from S to $\wp(S)$, note that for any $x \in S$, we have $\{x\} \subseteq S$. Therefore, $\{x\} \in \wp(S)$ for any $x \in S$, so f is a legal function from S to $\wp(S)$.
>
> Let's now prove that f is injective. Consider any $x_1, x_2 \in S$ where $f(x_1) = f(x_2)$. We'll prove that $x_1 = x_2$. Because $f(x_1) = f(x_2)$, we have $\{x_1\} = \{x_2\}$. Since two sets are equal if and only if their elements are the same, this means that $x_1 = x_2$, as required.
>
> However, f is not surjective. Notice that $\varnothing \in \wp(S)$, since $\varnothing \subseteq S$ for any set S, but that there is no x such that $f(x) = \varnothing$; this is because $\varnothing$ contains no elements and $f(x)$ always contains one element. Since f is not surjective, it is not a bijection. Thus $|S| \neq |\wp(S)|$. ■

Unfortunately, this argument is incorrect. What's wrong with this proof? Justify your answer by pointing to a specific claim that's made here that's incorrect and explaining why it's incorrect.

Problem Four: Paradoxical Sets

What happens if we take *absolutely everything* and throw it into a set? If we do, we would get a set called the ***universal set***, which we denote $\mathscr{U}$:

$$\mathscr{U} = \{\ x \mid x \text{ exists}\ \}$$

Absolutely everything would belong to this set:

$$1 \in \mathscr{U} \qquad \mathbb{N} \in \mathscr{U} \qquad \text{CS103} \in \mathscr{U} \qquad \text{whimsy} \in \mathscr{U}$$

In fact, we'd even have $\mathscr{U} \in \mathscr{U}$, which is strange but not immediately a problem. Unfortunately, the set $\mathscr{U}$ doesn't actually exist, as its existence would break mathematics.

i. Prove that if A and B are arbitrary sets where $A \subseteq B$, then $|A| \leq |B|$.

Look at the online Guide to Cantor's Theorem. Formally speaking, if you want to prove that $|A| \leq |B|$, what do you need to prove? Your answer should involve defining some sort of function between A and B and proving that function has some specific property or properties.

ii. Using your result from (i), prove that if $\mathscr{U}$ exists, then $|\wp(\mathscr{U})| \leq |\mathscr{U}|$.

iii. The ***Cantor-Bernstein-Schroeder Theorem*** says that if A and B are sets such that $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$. Using the Cantor-Bernstein-Schroeder Theorem and the formal definitions of the different cardinality relations, prove that if A and B are sets where $|A| \leq |B|$, then $|B| \not\leq |A|$.

In this proof you're essentially showing that the $\leq$ and $<$ relations involving set cardinality work like the $\leq$ and $<$ relations over regular numbers. Since the goal of the proof is to show that these cardinality relations work like regular inequality symbols, this result isn't "obvious" and you'll need to use formal definitions.

iv. Using your results from parts (ii) and (iii) of this problem, prove that $\mathscr{U}$ does not exist.

The result you've proven shows that there is a collection of objects (the collection of everything that exists) that cannot be put into a set. When this was discovered at the start of the twentieth century, it led to a reexamination of logical reasoning itself and a more formal definition of what objects can and cannot be gathered into a set. If you're curious to learn more about what came out of that, take Math 161 (Set Theory) or Phil 159 (Non-Classical Logic).

Problem Five: Independent Sets

An ***independent set*** in a graph $G = (V, E)$ is a set $I \subseteq V$ with the following property:

$$\forall u \in I.\ \forall v \in I.\ \{u, v\} \notin E.$$

This question explores independent sets and their properties.

i. Explain what an independent set is in plain English and without making reference to first-order logic. No justification is necessary.

You may want to draw some pictures of graphs to see what independent sets look like. Don't just come up with a literal translation of the first-order logic formula above; see if you can find a simple explanation of what independent sets are.

ii. Download the starter files for Problem Set Four from the website, then open the file `GraphTheory.cpp` and implement a function

```
bool isIndependentSet(Graph G, std::set<Node> I)
```

that takes as input a graph G and a set I, then returns whether I is an independent set in G. The definition of the `Graph` type is provided in `GraphTheory.h`.

Our provided starter code contains logic that, given a graph G, finds the largest independent set in G by making a lot of repeated calls to `isIndependentSet`. You might want to look over some of the sample graphs to get a feel for what large independent sets look like.

iii. You want to conduct a poll for an election and would like to survey people where no two people in the group know each other so that you can get a good representative sample of the population. Ideally, you'd find a large group of mutual strangers so that your poll has good statistical power. Explain how you might model this problem in terms of building some sort of graph, then finding a large independent set in it. Briefly justify your answer; no formal proof is necessary.

We don't want you to design an algorithm for finding large independent sets. Instead, imagine you have a program that you give as input a graph and that hands back to you an independent set in that graph. What graph would you give that program as input? What would you do with the output?

The size of an independent set is the number of nodes in it. Formally speaking, if I is an independent set, then the size of I is given by $|I|$. The ***independence number*** of a graph G, denoted $\boldsymbol{\alpha(G)}$, is the size of the largest independent set in G. (Note that there can be many different independent sets in a graph G that are all tied for the largest.)

iv. Edit the file `PartA.graph` in the `res/` directory to define a graph G where G has exactly five nodes and $\alpha(G) = 5$.

v. Edit the file `PartB.graph` in the `res/` directory to define a graph G where G has exactly five nodes and $\alpha(G) = 1$.

A graph can contain multiple different independent sets.

vi. Prove or disprove: if $G = (V, E)$ is a graph and I_1 and I_2 are independent sets in G, then $I_1 \cap I_2$ is an independent set in G.

Independent sets are specified using a definition given in first-order logic. Make sure your proof is structured around that definition, the same way that proofs of reflexivity, symmetry, etc. are structured around those first-order definitions.

Problem Six: Vertex Covers

A ***vertex cover*** in a graph $G = (V, E)$ is a set $C \subseteq V$ with the following property:

$$\forall u \in V.\ \forall v \in V.\ (\{u, v\} \in E \to u \in C \vee v \in C).$$

This question explores vertex covers and their properties.

i. Translate the definition of a vertex cover into plain English. No justification is necessary.

As before, you may want to draw pictures. See if you can find an explanation that's more than just a literal translation of the above statement.

ii. Implement a function

```
bool isVertexCover(Graph G, std::set<Node> C)
```

that takes as input a graph G and a set C, then returns whether C is a vertex cover of G.

Our provided starter code contains logic that, given a graph G, finds the smallest vertex cover in G by making a lot of repeated calls to `isVertexCover`. You might want to explore some of the sample graphs to get a feel for what vertex covers look like.

iii. Suppose that you have a map of a city's roads and streets (assume that the roads are set up so that it's possible to walk between any two points in the city). You want to set up information kiosks so that no matter where you are, you never need to walk more than a block to get to a kiosk. You also want to use as few information kiosks as possible to accomplish this. Explain how you might model this problem by building some sort of graph and looking for a small vertex cover in that graph. Briefly justify your answer; no formal proof is necessary.

Along the lines of part (iii) of the previous problem, assume you have a black box for finding small vertex covers and don't try to come up with an algorithm on your own. What graph would you hand into that black box? What would you do with the resulting vertex cover it hands back to you?

The size of a vertex cover is the number of nodes in it. Formally speaking, if C is a vertex cover, then the size of C is given by $|C|$. The ***vertex cover number*** of a graph G, denoted $\boldsymbol{\tau(G)}$, is the size of the smallest vertex cover of G. (Note that there can be many different vertex covers in a graph G that are all tied for the smallest.)

iv. Edit the file `PartC.graph` in the `res/` directory to define a graph G with exactly five nodes where $\tau(G) = 0$.

v. Edit the file `PartD.graph` in the `res/` directory to define a graph G with exactly five nodes where $\tau(G) = 4$.

As with independent sets, graphs can contain multiple different vertex covers.

vi. Prove or disprove: if $G = (V, E)$ is a graph and C_1 and C_2 are vertex covers of G, then $C_1 \cap C_2$ is a vertex cover of G.

Vertex covers have some really cool applications. Check out this Numberphile video for one of them!

Problem Seven: Chromatic and Clique Numbers

Recall from lecture that a ***k-vertex coloring*** of a graph is a way of coloring each node in the graph one of up to k different colors such that no two adjacent nodes are the same color. The ***chromatic number*** of a graph, denoted $\chi(G)$, is the minimum value of k for which a k-vertex coloring exists.

i. Implement a function

```
bool isKVertexColoring(Graph G,
                       std::map<Node, Color> colors,
                       std::size_t k);
```

that takes as input a graph, a mapping from nodes in the graph to colors, and a number k, then returns whether the indicated coloring is a k-vertex coloring. You can assume that the map has one key for each node in the graph and that the only keys in the map are nodes in G. (The type `std::size_t` represents a natural number.)

Our provided starter code contains some logic that, given a graph G, finds a minimum k-vertex-coloring of G by making a lot of calls to your `isKVertexColoring` function. We recommend taking some time to look at a few sample graphs and their minimum colorings – they're quite pretty!

Here's a new definition. A ***clique*** in a graph $G = (V, E)$ is a set $K \subseteq V$ with the following property:

$$\forall u \in K.\ \forall v \in K.\ (u \neq v \rightarrow \{u, v\} \in E).$$

This question explores the connection between cliques and chromatic numbers.

ii. Translate the definition of a clique into plain English. No justification is necessary.

iii. Implement a function

```
bool isClique(Graph G, std::set<Node> K)
```

that takes as input a graph G and a set K, then returns whether K is a clique in G.

Our provided starter code contains some logic that, given a graph G, finds the largest clique in G by making a lot of calls to your `isClique` function. You may want to take a look at some of the provided sample graphs to see what large cliques look like.

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

The size of a clique K, denoted $|K|$, is the number of nodes in K. The ***clique number*** of a graph, denoted $\omega(G)$, is the size of the largest clique in G. (Note that there can be many different cliques in a graph G that are all tied for the largest.)

iv. Prove that if G is a graph, then $\chi(G) \geq \omega(G)$.

We're expecting you to write a formal proof here. It may be easiest to do this by contradiction.

v. Edit the file `PartE.graph` in the `res/` directory to contain a graph G where $\chi(G) \neq \omega(G)$. This shows that, in general, the chromatic and clique numbers of a graph don't have to be equal.

Aim to find the smallest example that you can. Although you aren't required to submit the simplest example possible and we aren't asking for an explanation as to why your answer is correct, you should not feel satisfied with your answer unless you can justify why it's got to be the simplest answer possible.

Problem Eight: Chromatic and Independence Numbers

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

In Problem Seven, you explored the connection between clique numbers and chromatic numbers. This problem explores the connection between independence numbers and chromatic numbers.

Let n be an arbitrary positive natural number. Prove that if G is an arbitrary undirected graph with exactly n^2+1 nodes, then $\chi(G) \geq n+1$ or $\alpha(G) \geq n+1$ (or both).

You should definitely check out what the Guide to Proofs on Discrete Structures says to do if you want to prove a statement of the form $P \vee Q$, since it'll make this proof a lot easier to write!

Problem Nine: Bipartite Graphs

An undirected graph $G = (V, E)$ is called a ***bipartite graph*** if there exist two sets V_1 and V_2 such that

- every node $v \in V$ belongs to exactly one of V_1 and V_2, and
- every edge $e \in E$ has one endpoint in V_1 and the other in V_2.

The sets V_1 and V_2 here are called ***bipartite classes*** of G. To help you get a better intuition for bipartite graphs, suppose that you have a group of people and a list of restaurants. You can illustrate which people like which restaurants by constructing a bipartite graph where V_1 is the set of people, V_2 is the set of restaurants, and there's an edge from a person p to a restaurant r if person p likes restaurant r.

Bipartite graphs have many interesting properties. One of the most fundamental is this one:

An undirected graph is bipartite if and only if it contains no cycles of odd length.

Intuitively, a bipartite graph contains no odd-length cycles because cycles alternate between the two groups V_1 and V_2, so any cycle has to have even length. The trickier step is proving that if G is a graph that contains no cycles of odd length, then G has to be bipartite. For now, assume that G has just one connected component; if G has multiple connected components, we can treat each one as a separate graph for the purposes of determining whether G is bipartite. (You don't need to prove this, but I'd recommend taking a minute to check why this is the case.)

Suppose G is a connected, undirected graph with no cycles of odd length. Choose any node $v \in V$. Let V_1 be the set of all nodes that are connected to v by a path of odd length and V_2 be the set of all nodes connected to v by a path of even length. (Note that these paths do not have to be simple paths.) Formally:

$$V_1 = \{ x \in V \mid \text{there is an odd-length path from } v \text{ to } x \}$$

$$V_2 = \{ x \in V \mid \text{there is an even-length path from } v \text{ to } x \}$$

i. Prove that V_1 and V_2 have no nodes in common.

Remember that there might be multiple different paths of different lengths from v to some other node x, so be careful not to talk about "the" path between v and x. Also note that these don't have to be simple paths.

ii. Using your result from part (i), prove that G is bipartite.

What do you need to show to prove that every node belongs to one of exactly two sets? Make sure you can point out how you are using the fact that G is connected and the fact that G has no cycles of odd length.

Optional Fun Problem 1: Hugs All Around! (1 Point Extra Credit)

There's a party with 137 attendees. Each person is either ***honest***, meaning that they *always* tell the truth, or ***mischievous***, meaning that they *never* tell the truth. After everything winds down, everyone is asked how many ***honest*** people they hugged at the party. Surprisingly, each of the numbers 0, 1, 2, 3..., and 136 was given as an answer exactly once.

How many honest people were at the party? Prove that your answer is correct and that no other answer could be correct.

Optional Fun Problem 2: How Many Functions Are There? (1 Point Extra Credit)

If A and B are sets, we can define the set $\boldsymbol{B^A}$ to be the set of all functions from A to B. Formally speaking:

$$B^A = \{\ f\ \mid\ f : A \to B\ \}$$

Prove that $|\mathbb{N}| < |\mathbb{N}^{\mathbb{N}}|$. This shows that $\aleph_0 < \aleph_0^{\aleph_0}$. Isn't infinity weird?

Problem Set 5

This problem set – the last one purely on discrete mathematics – is designed as a cumulative review of the topics we've covered so far and a proving ground to try out your newfound skills with mathematical induction. The problems here span all sorts of topics – parallel processing, the nature of infinity, tiling problems, and social networks – and we hope that it serves as a fitting coda to our whirlwind tour of discrete math!

We recommend that you ***read Handout #28, "Guide to Induction," before starting this problem set***. It contains a lot of useful advice about how to approach problems inductively, how to structure inductive proofs, and how to not fall into common inductive traps. Additionally, before submitting, be sure to ***read over Handout #29, the "Induction Proofwriting Checklist,"*** for a list of specific things to watch for in your solutions before submitting.

As a note on this problem set – normally, you're welcome to use any proof technique you'd like to prove results in this course. On this problem set, we've specifically requested on some problems that you prove a result inductively. For those problems, you should prove those results using induction or complete induction, even if there is another way to prove the result. (If you'd like to use induction in conjunction with other techniques like proof by contradiction or proof by contrapositive, that's perfectly fine.)

As always, please feel free to drop by office hours, visit Piazza, or send us emails if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, February 16th at 2:30PM.
There is no checkpoint problem.

Problem One: Chains and Antichains

Let A be an arbitrary set and $<_A$ be some strict order over A. A ***chain in*** $<_A$ is a series $x_1, \ldots, x_k$ of elements drawn from A such that

$$x_1 <_A x_2 <_A \ldots <_A x_k.$$

Intuitively, a chain is a series of values in ascending order according to the strict order $<_A$. The ***length*** of a chain is the number of elements in that chain.

i. Consider the $\subsetneq$ relation over the set $\wp(\{a, b, c\})$, where $A \subsetneq B$ means that $A \subseteq B$ but $A \neq B$. What is the length of the longest chain in this strict order? Give an example of a chain with that length. No justification is necessary.

Draw the Hasse diagram and see if you can find a visual intuition for the definition of a chain.

Now, let's cover a new definition. An ***antichain in*** $<_A$ is a set $X \subseteq A$ such any two elements in X are incomparable by the $<_A$ relation. In other words, a set $X \subseteq A$ is an antichain if

$$\forall a \in X. \forall b \in X. (a \not<_A b \quad \wedge \quad b \not<_A a)$$

The ***size*** of an antichain X is the number of elements in X.

ii. Consider the $\subsetneq$ relation over the set $\wp(\{a, b, c\})$. What is the size of the largest antichain in this strict order? Give an example of an antichain with that size. No justification is necessary.

Draw the Hasse diagram and see if you can find a visual intuition for the definition of an antichain.

Given an arbitrary strictly ordered set, you can't say anything *a priori* about the size of the largest chain or antichain in that strict order. However, you can say that at least one of them must be relatively large relative to the strictly ordered set.

Let $<_A$ be an arbitrary strict order over an arbitrary set A containing exactly n^2+1 elements for some natural number $n \geq 1$. We're going to ask you to prove the following result: either A contains a chain of length $n+1$ or an antichain of size $n+1$ (or both). Following the advice from Handout 17, we'll prove this by instead proving that if A does *not* contain a chain of length $n+1$ or greater, then A must contain an antichain of size $n+1$ or greater.

iii. For each element $a \in A$, we'll say that the ***height*** of a is the length of the longest chain whose final element is a. Prove that if A does not contain a chain of length $n+1$ or greater, then there must be at least $n+1$ elements of A at the same height.

Something to think about: what's the smallest possible height of an element of A?

iv. Your result from part (iii) establishes that if A does not contain a chain of length $n+1$ or greater, there must be a collection of $n+1$ elements of A at the same height as one another. Prove that if A does not contain a chain of length $n+1$ or greater, then it contains an antichain of size $n+1$ or greater.

Intuitively speaking, if $<_A$ is a strict order over A that represents some prerequisite structure on a group of tasks, a chain represents a series of tasks that have to be performed one after the other, and an antichain represents a group of tasks that can all be performed in parallel (do you see why?) In the context of parallel computing, the result you've proved states that if a group of tasks doesn't contain long dependency chains, that group must have a good degree of parallelism. Take CS149 for more information!

Problem Two: Recurrence Relations

A *recurrence relation* is a recursive definition of the terms in a sequence. Typically, a recurrence relation specifies the value of the first few terms in a sequence, then defines the remaining terms from the previous terms. For example, the *Fibonacci sequence* can be defined by the following recurrence relation:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_{n+2} = F_n + F_{n+1}$$

The first terms of this sequence are $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, etc.

Some recurrence relations define well-known sequences. For example, consider the following recurrence relation:

$$a_0 = 1$$
$$a_{n+1} = 2a_n$$

The first few terms of this sequence are 1, 2, 4, 8, 16, 32, …, which happen to be powers of two. It turns out that this isn't a coincidence – this recurrence relation perfectly describes the powers of two.

i. Prove by induction that for any $n \in \mathbb{N}$, we have $a_n = 2^n$.

In case you're wondering what you're asked to prove here: the official definition a_n is given by the recurrence relation. If you ever need to determine what the value of a_n is for some value of n, look back at that definition. We want you to prove that, as a consequence of that definition, the value of a_n, the nth term in the series, is always exactly 2^n.

Minor changes to the recursive step in a recurrence relation can lead to enormous changes in what numbers are generated. Consider the following two recurrence relations, which are similar to the a_n sequence defined above but with slight changes to the recursive step:

$$b_0 = 1 \qquad\qquad c_0 = 1$$
$$b_{n+1} = 2b_n - 1 \qquad\qquad c_{n+1} = 2c_n + 1$$

ii. Find non-recursive definitions for b_n and c_n, then prove by induction that your definitions are correct.

This one is hard to do just by eyeballing the recurrences. Try expanding out the first few terms of these sequences and see what you find.

Finding non-recursive definitions for recurrences (often called "solving" the recurrence) is useful in the design and analysis of algorithms. Commonly, when trying to analyze the runtime of an algorithm, you will arrive at a recurrence relation describing the runtime on an input of size n in terms of the runtime on inputs of smaller sizes. Solving the recurrence then lets you precisely determine the runtime. To learn more, take CS161, Math 108, or consider reading through the excellent textbook *Concrete Mathematics* by Graham, Knuth, and Patashnik.

Problem Three: It'll All Even Out

Our very first proof by induction was the proof that for any natural number n, we have that

$$2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1.$$

This result is still true for the case where $n = 0$, since in that case the sum on the left-hand side of the equation is the *empty sum* of zero numbers, which is by definition equal to zero. It's also true for the case where $n = 1$; in that case, the sum on the left-hand side of the equality just has a single term in it (2^0) and the right-hand side has the same value.

Below is a proof by complete induction of an incorrect statement about what happens when you sum up zero or more real numbers:

> ***Theorem:*** The sum of any number of real numbers is even.
>
> ***Proof:*** Let $P(n)$ be the statement "the sum of any n real numbers is even." We will prove by complete induction that $P(n)$ holds for all $n \in \mathbb{N}$, from which the theorem follows.
>
> As a base case, we prove $P(0)$, that the sum of any 0 real numbers is even. The sum of any zero numbers is the empty sum and is by definition equal to 0, which is even. Thus $P(0)$ holds.
>
> For our inductive step, assume for some arbitrary $k \in \mathbb{N}$ that $P(0)$, …, and $P(k)$ are true. We will prove that $P(k+1)$ is true, meaning that the sum of any $k+1$ real numbers is even. To do so, let x_1, x_2, …, x_k, and x_{k+1} be arbitrary real numbers and consider the sum
>
> $$x_1 + x_2 + \ldots + x_k + x_{k+1}.$$
>
> We can group the first k terms and the last term independently to see that
>
> $$x_1 + x_2 + \ldots + x_k + x_{k+1} = (x_1 + x_2 + \ldots + x_k) + (x_{k+1}).$$
>
> Now, consider the sum $x_1 + x_2 + \ldots + x_k$ of the first k terms. This is the sum of k real numbers, so by our inductive hypothesis that $P(k)$ is true we know that this sum must be even. Similarly, consider the sum x_{k+1} consisting of just the single term x_{k+1}. By our inductive hypothesis that $P(1)$ is true, we know that this sum must be even.
>
> Overall, we have shown that $x_1 + x_2 + \ldots + x_k + x_{k+1}$ can be written as the sum of two even numbers (namely, $x_1 + x_2 + \ldots + x_k$ and x_{k+1}), so $x_1 + x_2 + \ldots + x_k + x_{k+1}$ is even. Thus $P(k+1)$ is true, completing the induction. ■

Of course, this result has to be incorrect, since there are many sums of real numbers that don't evaluate to an even number. The sum 2 + 3 + 4, for example, works out to 9, and the sum $\pi + 1$ doesn't even work out to an integer!

What's wrong with this proof? Be as specific as possible. For full credit, you should be able to identify a specific claim made in the proof that is not correct, along with an explanation as to why it's incorrect.

Think about our "induction as a machine" analogy from lecture that explains why you can start with a base case and inductive step and end up with a proof that works for all natural numbers. See what happens if you try that out here.

Problem Four: So What Exactly Is Multiplication, Anyway?

On Problem Set Four, you proved that $2\aleph_0 = \aleph_0$ by finding a bijection between $\mathbb{N} \times \{\bigstar, \smiley\}$ and $\mathbb{N}$. But why exactly did finding that bijection tell you anything about $2\aleph_0$?

Let's suppose you have two cardinal numbers κ_1 and κ_2. (A ***cardinal number*** is a quantity that represents the size of a set; all natural numbers are cardinal numbers, as is $\aleph_0$.) We can define $\kappa_1 \cdot \kappa_2$ as follows: pick any sets A and B where $|A| = \kappa_1$ and $|B| = \kappa_2$, then determine $|A \times B|$. The resulting cardinal number is then defined to be $\kappa_1 \cdot \kappa_2$. In other words, $4 \cdot 3$ is *defined* to be $|A \times B|$ for any sets A and B where $|A| = 4$ and $|B| = 3$. Similarly, $2\aleph_0$ is *defined* to be $|A \times B|$ for any sets A and B where $|A| = 2$ and $|B| = \aleph_0$.

For this definition to work, we have to make sure that the cardinality of the Cartesian product depends purely on the cardinalities of the two sets, not their contents. For example, this definition wouldn't give us a way to compute $4 \cdot 3$ if the cardinality of the Cartesian product of a set of four apples and three oranges was different than the cardinality of the Cartesian product of a set of four figs and three dates. We need to show that for any sets A, B, C, and D, that if $|A| = |C|$ and $|B| = |D|$, then $|A \times B| = |C \times D|$. That way, when determining $\kappa_1 \cdot \kappa_2$, it doesn't matter which sets of cardinality κ_1 and κ_2 we pick. (Contrast thiswith $|A \cup B|$: if you have sets A and B, you can't necessarily predict $|A \cup B|$ from $|A|$ and $|B|$.)

Let A, B, C, and D be arbitrary sets where $|A| = |C|$ and $|B| = |D|$. Our goal is to prove $|A \times B| = |C \times D|$. Since we know $|A| = |C|$, there has to be some bijection $g : A \to C$. Since we know $|B| = |D|$, there has to be some bijection $h : B \to D$. Now, consider the function $f : A \times B \to C \times D$ defined as follows:

$$f(a, b) = (g(a), h(b))$$

That is, the output of f when applied to the pair (a, b) is an ordered pair whose first element is $g(a)$ and whose second element is $h(b)$.

i. Using the function f defined above, prove that $|A \times B| = |C \times D|$. Specifically, prove that f is a bijection between $A \times B$ and $C \times D$.

Two ordered pairs are equal if and only if their corresponding elements are equal. Although the preceding discussion talked about how to multiply cardinal numbers, that discussion only works because of the result that you'll be proving here, so you can't assume that $|A \times B| = |A| \cdot |B|$ in the course of writing this proof. Your proof should purely focus on proving that f is a bijection.

We can define the ***Cartesian power*** of a set as follows. For any set A and any positive natural number n, we define A^n inductively:

$$A^1 = A$$
$$A^{n+1} = A \times A^n \text{ (for } n \geq 1\text{)}$$

Amazingly, we know that $|\mathbb{N}| = |\mathbb{N}^2|$, meaning that there's the same number of pairs of natural numbers as there are natural numbers themselves. Feel free to use this fact in the following problem.

ii. Using your result from (i), the above definition, and the fact that $|\mathbb{N}| = |\mathbb{N}^2|$, prove by induction that $|\mathbb{N}^k| = |\mathbb{N}|$ for all nonzero $k \in \mathbb{N}$. This result means that for any nonzero finite k, there are the same number of k-tuples of natural numbers as natural numbers.

If κ is a cardinal number and $n \geq 1$ is a natural number, then *by definition* the value of κ^n is $|A^n|$, where A is any set of cardinality κ. Your result from part (ii) shows that $\aleph_0{}^n = \aleph_0$ for any positive natural number n.

Problem Five: Induction and Recursion

There's a close connection between mathematical induction and recursion, and many of the proofs by induction that we did in class can be thought of as claims about how specific recursive functions work.

One of the first proofs by induction that we did was to prove that, given a collection of 3^n coins containing a single counterfeit coin that's heavier than the rest, it is always possible to discover which coin is fake using exactly n weighings on a balance. The key idea behind the proof was, essentially, a recursive algorithm that can be used to actually go and find which of the coins is counterfeit!

i. Implement a recursive function

```
Coin counterfeitIn(std::vector<Coin> coins, Balance balance);
```

that takes as input a set of ***exactly 3^n coins*** for some natural number n, one of which is counterfeit and weighs more than the rest, and returns which one that is. You're provided a balance you can use to weigh groups of coins and can make at most n weighings on that balance. Check the header `CounterfeitCoins.h` for a description of the relevant types here.

Test your code locally on your machine before submitting it, since if your solution crashes due to a logic error the autograder won't give you any useful feedback. Our provided starter files provide an interface you can use to test out your function on a number of different inputs and will show you which coins actually get weighed against one another.

Your code from part (i) shows that the *inductive* argument we made in class can be converted into a *recursive* function that actually finds the coin!

Now, here's a fun little variant on the counterfeit coin problem. Imagine that you're given a collection of coins. You're told that there *might* be a counterfeit in it, but then again, there might not be. If there is a counterfeit coin, it's guaranteed to be heavier than the rest. Your job is to determine whether there even is a counterfeit coin at all and, if so, to return which one it is.

ii. Implement a recursive function

```
Coin maybeCounterfeitIn(std::vector<Coin> coins, Balance balance)
```

that takes as input a set of ***exactly $3^n - 1$ coins*** for some natural number n, which *might* contain a counterfeit that weighs more than the rest. The function should either return the counterfeit coin if one exists, or return the special constant `None` if none of the coins are counterfeit. You're provided a balance you can use to weigh groups of coins and can make at most n weighings on that balance.

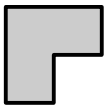
Again, test locally, and test thoroughly – it's easy to miss cases!

iii. Using the recursive intuition that you developed in the course of solving part (ii) of this problem, prove that given any collection of exactly $3^n - 1$ coins, of which at most one is a counterfeit that weighs more than the rest, it is always possible to identify which coin that is using at most n weighings on a balance (or to report that all coins are genuine). Your proof should have a similar structure to the one about counterfeit coins from lecture. While you should should not explicitly reference the code you wrote in part (ii) of this problem, you may want to use the same recursive insight from that problem to guide the structure of your proof.

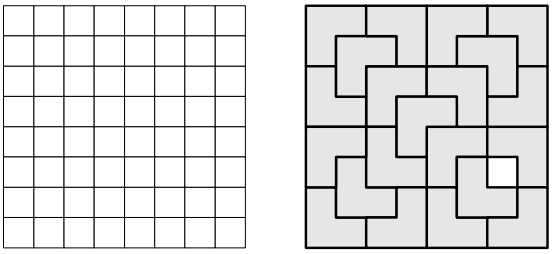
We hope that this exercise gives you a better sense for the interplay between theory (proof by induction) and practice (recursive problem-solving). If you're interested in this sort of thing, we strongly recommend checking out CS161, where you'll alternate between designing clever algorithms and using induction to prove that they work correctly.

Problem Six: Tiling with Triominoes

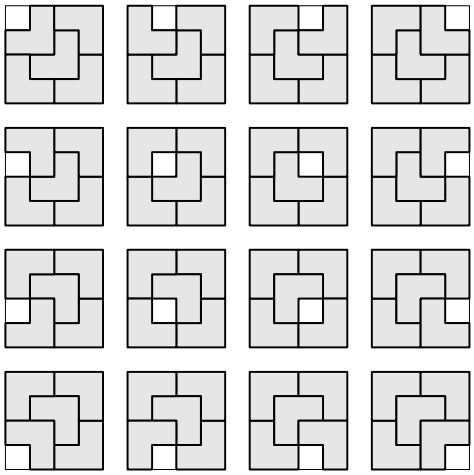
Recall from Problem Set One that a *right triomino* is an L-shaped tile that looks like this:



Suppose you're given a $2^n \times 2^n$ grid of squares and want to tile it with right triominoes by covering the grid with triominoes such that all triominoes are completely on the grid and no triominoes overlap. Here's an attempt to cover an 8×8 grid with triominoes, which doesn't manage to cover all squares:



Amazingly, it turns out that it is always possible to tile any $2^n \times 2^n$ grid that's missing exactly one square with right triominoes. It doesn't matter what n is or which square is removed; there is always a solution to the problem. For example, here are all the ways to tile a 4×4 grid that has a square missing:



This question explores why this is the case.

i. Prove, by induction, that $4^n - 1$ is a multiple of three for any $n \in \mathbb{N}$.

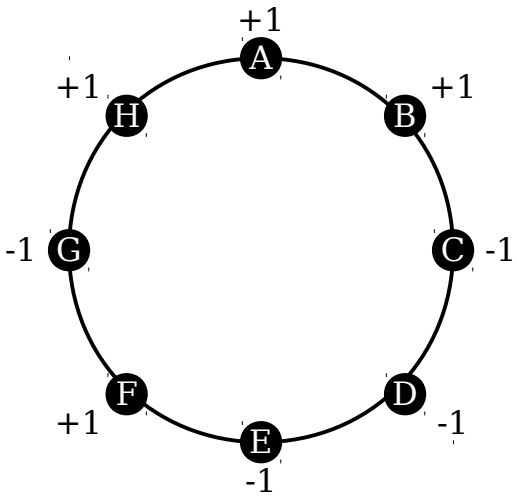
Any $2^n \times 2^n$ grid missing a square has a number of squares has exactly $4^n - 1$ squares, and so its number of squares is a multiple of three. Although you *can* show that a figure *can't* be tiled with triominoes by showing that its number of squares isn't a multiple of three, you *can't* show that a figure *can* be tiled with triominoes purely by showing that its number of squares is a multiple of three. The arrangement matters.

ii. Draw a figure made of squares where the number of squares is a multiple of three, yet the figure cannot be tiled with right triominoes. Briefly justify your answer; no formal proof is necessary.

iii. Prove by induction that for any natural number n, any $2^n \times 2^n$ grid with any one square removed can be tiled by right triominoes.

Before you write this proof, try seeing if you can find a nice recursive pattern you can follow that will let you fully tile any such board. Once you've found it, formalize your idea in your answer. You may want to think about how to start with a larger board and subdivide it into some number of smaller boards.

Problem Seven: The Circle Game



Here's a game you can play. Suppose that you have a circle with $2n$ arbitrarily-chosen points on its circumference. n of these points are labeled +1, and the remaining n are labeled -1. One sample circle with eight points, of which four are labeled +1 and four are labeled -1, is shown to the right.

Here's the rules of the game. First, choose one of the $2n$ points as your starting point. Then, start moving clockwise around the circle. As you go, you'll pass through some number of +1 points and some number of -1 points. You lose the game if at any point on your journey you pass through more -1 points than +1 points. You win the game if you get all the way back around to your starting point without losing.

For example, if you started at point A, the game would go like this:

Start at A: +1.
Pass through B: +2.
Pass through C: +1.
Pass through D: 0.
Pass through E: -1. *(You lose.)*

If you started at point G, the game would go like this:

Start at G: -1 *(You lose.)*

However, if you started at point F, the game would go like this:

Start at F: +1.
Pass through G: 0.
Pass through H: +1.
Pass through A: +2.
Pass through B: +3.
Pass through C: +2.
Pass through D: +1.
Pass through E: +0.
Return to F. *(You win!)*

Interestingly, it turns out that no matter which n points are labeled +1 and which n points are labeled -1, there is always at least one point you can start at to win the game.

Prove, by induction, that the above fact is true for any $n \geq 1$.

This one is all about finding the right setup. Check the Guide to Induction and Inductive Proofwriting Checklist for details.

Problem Eight: Nim

Nim is a family of games played by two players. The game begins with several piles of stones, each of which has zero or more stones in it, that are shared between the two players. Players alternate taking turns removing any nonzero number of stones from any single pile of their choice. If at the start of a player's turn all the piles are empty, then that player loses the game.

Prove, by induction, that if the game is played with just two piles of stones, each of which begins with exactly the same number of stones, then the second player can always win the game if she plays correctly.

Before trying to write up your answer to this question, we recommend playing this game with a partner until you can find a winning strategy. Once you spot the pattern, see if you can find a way to formalize it using induction. Be wary of writing statements of the form "and so on" or "by repeating this;" those aren't rigorous ways to formalize that a process will eventually do something.

Problem Nine: Independent Sets Revisited

Recall from Problem Set Four that an ***independent set*** in a graph $G = (V, E)$ is a set $I \subseteq V$ where no two nodes in I are adjacent to one another. As you saw on that problem set, many problems in graph theory boil down to finding large independent sets in a graph. The question then arises: how big of an independent set can you reasonably expect to find?

i. Let G be a graph where each node has degree less than or equal to d. (As a reminder, the ***degree*** of a node in a graph is the number of nodes that it's adjacent to.) Prove, by ***complete*** induction on n, that if G is a graph with $n \geq 0$ nodes, then G has an independent set of size at least $\frac{n}{d+1}$.

As with the Circle Game problem, one of the big steps in solving this one is making sure you're setting up this problem properly. Will you start with a smaller graph and add another node in, or start with a larger graph and take a note out?

We strongly recommend checking your work by thinking about how you'd convert your inductive proof into a recursive algorithm, then testing that algorithm out on some sample graphs. If you've found the right argument, you should end up with a rather simple algorithm.

There's also no guarantee that the graph G has a node of degree exactly d. The statement "everyone in CS103 has height less than or equal to 1km" is true, even though no one is anywhere close to that tall.

Graphs are often used to model social networks: each person is a node, and friendships are represented by edges. The sorts of graphs you find this way in the real world have all sorts of interesting properties (take CS224W or CS267 for more details!) In particular, social networks tend to have a lot of ***triangles***, collections of three nodes that are all mutually adjacent.

This question explores a class of graphs that are quite different from the graphs that typically arise in social networks: ***triangle-free graphs***. A triangle-free graph is one that contains no triangles. That is, if you pick three distinct nodes in the graph, some two of them will not be adjacent.

ii. Draw a triangle-free graph with nine nodes that has a node of degree eight. Justify your answer.

iii. Prove that if G is a triangle-free graph with n^2 nodes, then G contains an independent set of size at least n.

This last one is tricky, so don't worry if you don't see it immediately. Note that we didn't say to use induction here. You may want to find a way to work in the result from part (i) of this problem. Work backwards: what value of d would you need to have in order to apply that result? Don't forget the result you found in part (ii), though: triangle-free graphs can have very high-degree nodes. What happens then?

Optional Fun Problem: Synchronicity (1 Point Extra Credit)

Let's say that an ***era*** is a historical time period with a definitive start date and definitive end date. For example, the Meiji Era ran from October 23, 1868 to July 30, 1912, and the Cuban Missile Crisis ran from October 16, 1962 to October 28, 1962. For simplicity, we'll assume that these time ranges include the entirety of their start and end dates.

Prove that no matter how you choose any fifty eras from history, you can either (1) find a date that's contained in at least eight of those eras, or (2) find eight eras of which no two have any days in common.

Problem Set 6

This sixth problem set explores the regular languages and their properties. This will be your first foray into computability theory, and I hope you find it fun and exciting!

As always, please feel free to drop by office hours, ask on Piazza, or email the staff list if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, February 23rd at 2:30PM

Problem One: Constructing DFAs

For each of the following languages over the indicated alphabets, construct a DFA that accepts precisely the strings that are in the indicated language. Your DFA does not have to have the fewest number of states possible, though for your own edification we'd recommend trying to construct the smallest DFAs possible.

Please use our online tool to design, test, and submit your answers to this problem. Handwritten or typed solutions will not be accepted. To use the tool, visit the CS103 website and click the "DFA/NFA Editor" link under the "Resources" header. If you're planning on submitting this assignment in a pair, in your GradeScope submission, please let us know the SUNetID (e.g. htiek or cbl, but not 06001234) of the partner who submitted the DFAs so that we can match the problem set to the submitted answers.

Unlike the programming assignments, you will not be able to see the results of the autograder when you submit. As a result, ***be sure to test your solutions thoroughly before you submit!***

i. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, construct a DFA for the language $\{ w \in \Sigma^* \mid w$ contains exactly two cs. $\}$

ii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$, construct a DFA for the language $\{ w \in \Sigma^* \mid w$ contains the same number of instances of the substring ab and the substring ba $\}$. Note that substrings are allowed to overlap, so aba $\in L$ and babab $\in L$.

iii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots, \texttt{z}\}$, construct a DFA for the language $\{ w \in \Sigma^* \mid w$ contains the word "cocoa" as a substring $\}$. *

Test your automaton thoroughly. This one has some tricky edge cases.

iv. Suppose that you are taking a walk with your dog along a straight-line path. Your dog is on a leash that has length two, meaning that the distance between you and your dog can be at most two units. You and your dog start at the same position. Consider the alphabet $\Sigma = \{\texttt{y}, \texttt{d}\}$. A string in Σ^* can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string "yydd" means that you take two steps forward, then your dog takes two steps forward. Let $L = \{ w \in \Sigma^* \mid w$ describes a series of steps that ensures that you and your dog are never more than two units apart $\}$. Construct a DFA for L.

Problem Two: Constructing NFAs

For each of the following languages over the indicated alphabets, construct an NFA that accepts precisely the strings that are in the indicated language. ***Please use our online system to design, test, and submit your automata***; see above for details. As before, ***please test your submissions thoroughly!***

i. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, construct an NFA for $\{ w \in \Sigma^* \mid w$ ends in a, bb, or ccc $\}$.

ii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$, construct an NFA for the language $\{ w \in \Sigma^* \mid$ the last character of w appears nowhere else in w, and $|w| \geq 1 \}$.

Stuck? Try reducing the alphabet to two or three letters and see if you can solve that version.

iii. For the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$, construct an NFA for the language $\{ w \in \Sigma^* \mid w$ contains at least two b's with exactly five characters between them $\}$. For example, baaaaab is in the language, as is aabaabaaabbb and abbbbbabaaaaaaab, but bbbbb is not, nor are bbbab or aaabab.

* DFAs are often used to search large blocks of text for specific substrings, and several string searching algorithms are built on top of specially-constructed DFAs. The *Knuth-Morris-Pratt* and *Aho-Corasick* algorithms use slightly modified DFAs to find substrings extremely efficiently.

Problem Three: $\wp(\Sigma^*)$

Let Σ be an alphabet. Give a short English description of the set $\wp(\Sigma^*)$. Briefly justify your answer.

We think that there is a single "best answer." You should be able to describe the set in at most ten words.

Problem Four: Concatenation, Kleene Stars, and Complements

The regular languages are closed under a number of different operations. This problem explores some properties of those operations.

i. Prove or disprove: if L is a nonempty, finite language and k is a positive natural number, then $|L^k| = |L|^k$. Here, the notation $|L|^k$ represents "the cardinality of L, raised to the kth power," and the notation $|L^k|$ represents "the cardinality of the k-fold concatenation of L with itself."

Call back to formal definitions as you work through this problem. The most common mistakes we see people make on this problem involve making unfounded claims about set cardinalities. If you want to prove that two sets have the same cardinality, set up an explicit bijection between them.

ii. Prove or disprove: there is a language L where $\overline{(L^*)} = (\overline{L})^*$.

A good warm-up problem: what is $\emptyset^*$? Don't answer this question based off of your intuition; look back at the formal definition of the Kleene star.

Problem Five: Monoids and Kleene Stars

The Kleene star operator is one of the more unusual operators we've covered over the course of the quarter. This problem explores one of its fundamental properties.

Let Σ be an arbitrary alphabet. A ***monoid over Σ*** is a set $M \subseteq \Sigma^*$ with the following properties:

$$\varepsilon \in M \qquad\qquad \forall x \in M.\ \forall y \in M.\ xy \in M.$$

Let L be an arbitrary language over Σ and let M be an arbitrary monoid over Σ. Prove that if $L \subseteq M$, then $L^* \subseteq M$.

In the course of writing this proof, please call back to the formal definition of language concatenation and the Kleene star, and use induction as appropriate. Here's a refresher on the definitions:

$$L_1L_2 = \{\ wx \mid w \in L_1 \text{ and } x \in L_2\ \}$$

$$L^0 = \{\varepsilon\} \qquad\qquad L^{n+1} = LL^n$$

$$L^* = \{\ w \mid \exists n \in \mathbb{N}.\ w \in L^n\ \}$$

The result that you've shown here is a building block toward a larger result: the Kleene star of a language L is the smallest monoid containing all the strings in L.

This problem is all about finding the right way to formalize things. Think about, ultimately, what it is that you need to prove. Before you start trying to prove that, break the task down into smaller pieces and make sure you organize everything in a way that makes the logical flow easy to read and rigorously covers all cases. Once you have the setup put together, dive in and fill out each section.

Problem Six: Hard Reset Sequences

A ***hard reset sequence*** for a DFA is a string w with the following property: starting from any state in the DFA, if you read w, you end up in the DFA's start state.

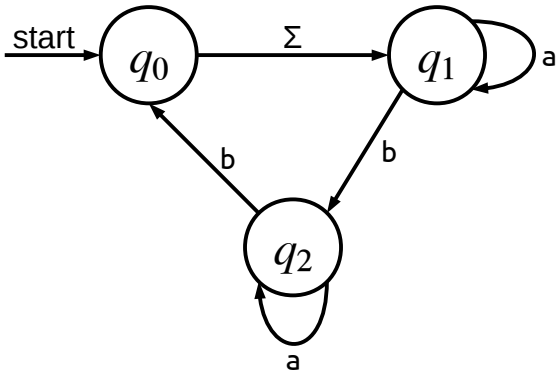
Hard reset sequences have many practical applications. For example, suppose you're remotely controlling a Mars rover whose state you're modeling as a DFA. Imagine there's a hardware glitch that puts the Mars rover into a valid but unknown state. Since you can't physically go to Mars to pick up the rover and fix it, the only way to change the rover's state would be to issue it new commands. To recover from this mishap, you could send the rover a hard reset sequence. Regardless of what state the rover got into, this procedure would guarantee that it would end up in its initial configuration.

Here is an algorithm that, given any DFA, will let you find every hard reset sequence for that DFA:

1. Add a new start state q_s to the automaton with ε-transitions to every state in the DFA.
2. Perform the subset construction on the resulting NFA to produce a new DFA called the ***power automaton***.
3. If the power automaton contains a state corresponding solely to the original DFA's start state, make that state the only accepting state in the power automaton. Otherwise, make every state in the power automaton a rejecting state.

This process produces a new automaton that accepts all the hard reset sequences of the original DFA. It's possible that a DFA won't have any hard reset sequences (for example, if it contains a dead state), in which case the new DFA won't accept anything.

Apply the above algorithm to the following DFA and give us a hard reset sequence for that DFA. For simplicity, please give the subset-constructed DFA as a transition table rather than a state-transition diagram. We've given you space for the table over to the right, and to be nice, we've given you exactly the number of rows you'll need.



	a	b

Sample hard reset sequence: _____

Finding a hard reset sequence for this DFA is a lot easier if you take a few minutes to think about what the power automaton does.

Problem Seven: Complementing NFAs

In lecture, we saw that if you take a DFA for a language L and flip all the accepting and rejecting states, you end up with a DFA for $\overline{L}$.

Draw a simple NFA for a language L where flipping all the accepting and rejecting states does not produce an NFA for $\overline{L}$. Briefly justify your answer; you should need at most a sentence or two.

Problem Eight: DFAs, Formally

When we first talked about graphs, we saw them first as pictures (objects connected by lines), but then formally defined a graph G as an ordered pair (V, E), where V is a set of nodes and E is a set of edges. This rigorous definition tells us what a graph actually is in a mathematical sense, rather than just what it looks like.

We've been talking about DFAs for a while now and seen how to draw them both as a collection of states with transitions (that is, as a state-transition diagram) and as a table with rows for states and columns for characters. But what exactly *is* a DFA, in a mathematical sense?

Formally speaking, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set, the elements of which we call *states*;
- Σ is a finite, nonempty set, the elements of which we call *characters*;
- $\delta : Q \times \Sigma \to Q$ is the ***transition function***, described below;
- $q_0 \in Q$ is the start state;
- $F \subseteq Q$ is the set of accepting states.

The transition function warrants a bit of explanation. When we've drawn DFAs, we've represented the transitions either by arrows labeled with characters or as a table with rows and columns corresponding to states and symbols, respectively. In this formal definition, the transition function δ is what ultimately specifies the transition. Specifically, for any state $q \in Q$ and any symbol $a \in \Sigma$, the transition from state q on symbol a is given by $\delta(q, a)$.

This question explores some properties of this rigorous definition.

i. Is it possible for a DFA to have no states? If so, define a DFA with no states as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.

ii. Is it possible for a DFA to have no *accepting* states? If so, define a DFA with no accepting states as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.

iii. In class, we said that a DFA must obey the rule that for any state and any symbol, there has to be exactly one transition defined on that symbol. What part of the above definition guarantees this?

iv. Is it possible for a DFA to have an unreachable state (that is, a state that is never entered regardless of what string you run the DFA on)? If so, define a DFA with an unreachable state as a 5-tuple, explaining why your 5-tuple meets the above requirements. If not, explain why not.

Defining a DFA requires you to define a transition function δ. You should define that function the same way that we've defined all other functions this quarter: either give a rule, define it as a piecewise function, or draw a picture. If you've having trouble doing so, it might mean that you picked too complex of a DFA and might want to search for something simpler.

Problem Nine: Why the Extra State?

In our proof that the regular languages are closed under the Kleene closure operator (that is, if L is regular, then L^* is regular), we used the following construction:

1. Begin with an NFA N where $\mathscr{L}(N) = L$.
2. Add in a new start state q_{start}.
3. Add an ε-transition from q_{start} to the start state of N.
4. Add ε-transitions from each accepting state of N to q_{start}.
5. Make q_{start} an accepting state.
6. Make every state besides q_{start} a rejecting state.

You might have wondered why we needed to add q_{start} as a new state to the NFA. It might have seemed more natural to do the following:

1. Begin with an NFA N where $\mathscr{L}(N) = L$.
2. Add ε-transitions from each accepting state of N to the start state of N.
3. Make the start state of N an accepting state.
4. Make every other state of N a rejecting state.

Unfortunately, this doesn't work correctly. Find a language L and an NFA N for L such that using the second construction does not create an NFA for L^*. Justify why the language of the new NFA isn't L^*.

The best way to get a handle on this problem is to work through a bunch of small, concrete examples of applying both constructions. Make sure you can articulate why, intuitively, each construction seems like it's a plausible way to form an automaton for the Kleene star of the language of the original automaton. Once you can do that, see if you can identify something potentially problematic about the second approach. Then, based on your understanding of that flaw, play around and see if you can build an NFA where that flaw leads to the construction not working properly.

Optional Fun Problem 1: Why Finite? (1 Point Extra Credit)

A ***deterministic infinite automaton***, or ***DIA***, is a generalization of a DFA in which the automaton has infinitely many different states. Formally speaking, a DIA is given by the same 5-tuple definition as a DFA from Problem Eight, except that Q must be an infinite set. Since DIAs have infinitely many states, they're mostly an object of purely theoretical study.

Prove that if L is an arbitrary language over an alphabet Σ, then there is a DIA that accepts L (that is, the DIA accepts every string in L and rejects every string not in L.) To do so, show how to start with a language L, formally define a 5-tuple corresponding to a DIA for L, then formally prove that that DIA accepts all and only the strings in L.

Optional Fun Problem 2: Edit Distances (1 Point Extra Credit)

The ***edit distance*** between two strings w and x is the minimum number of edits that need to be made to w to convert it into x. Here, an ***edit*** consists of either adding a character somewhere into w, deleting a character somewhere from w, or replacing a character of w with another. For example, `cat` and `dog` have edit distance 3, `table` and `maple` have edit distance 2, and `edit` and `distance` have edit distance 6.

Let Σ = {`w`, `h`, `i`, `m`, `s`, `y`}. Design an NFA for { $w \in \Sigma^*$ | the edit distance of w and `whimsy` is at most three }. Submit your answer online, and let us know in your written assignment who made the submission.

Problem Set 7

What can you do with regular expressions? What are the limits of regular languages? In this problem set, you'll explore the answers to these questions along with their practical consequences.

As always, please feel free to drop by office hours, ask on Piazza, or send us emails if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, March 2nd at 2:30PM

Problem One: Designing Regular Expressions

Below are a list of alphabets and languages over those alphabets. For each language, write a regular expression for that language.

Please use our online tool to design, test, and submit your regular expressions. Typed or handwritten solutions will not be accepted. To use it, visit the CS103 website and click the "Regex Editor" link under the "Resources" header. As before, if you submit in a pair, please make a note in your GradeScope submission of which partner submitted your answers to this question so that we know where to look. Also, as a reminder, please test your submissions thoroughly, since we'll be grading them with an autograder.

i. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ does not contain `ba` as a substring $\}$. Write a regular expression for L.

ii. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ does not contain `bb` as a substring $\}$. Write a regular expression for L.

You may want to draw out some sample strings in this language and look for a pattern.

iii. Suppose you are taking a walk with your dog on a leash of length two. Let $\Sigma = \{\texttt{y}, \texttt{d}\}$ and let $L = \{ w \in \Sigma^* \mid w$ represents a walk with your dog on a leash where you and your dog both end up at the same location $\}$. For example, we have `yyddddyy` $\in L$ because you and your dog are never more than two steps apart and both of you end up four steps ahead of where you started; similarly, `ddydyy` $\in L$. However, `yyyyddd` $\notin L$, since halfway through your walk you're three steps ahead of your dog; `ddyd` $\notin L$, because your dog ends up two steps ahead of you; and `ddyddyyy` $\notin L$, because at one point your dog is three steps ahead of you. Write a regular expression for L.

iv. Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \neq$ `ab` $\}$. Write a regular expression for L.

v. Let $\Sigma = \{\texttt{M}, \texttt{D}, \texttt{C}, \texttt{L}, \texttt{X}, \texttt{V}, \texttt{I}\}$ and let $L = \{ w \in \Sigma^* \mid w$ is number less than 2,000 represented in Roman numerals $\}$. For example, `CMXCIX` $\in L$, since it represents the number 999, as are the strings `L` (50), `VIII` (8), `DCLXVI` (666), `CXXXVII` (137), `CDXII` (412), and `MDCXVIII` (1,618). However, we have `VIIII` $\notin L$ (you'll never have four `I`'s in a row; use `IX` or `IV` instead), that `MM` $\notin L$ (it's a Roman numeral, but it's for 2,000, which is too large), that `VX` $\notin L$ (this isn't a valid Roman numeral), and that `IM` $\notin L$ (the notation of using a smaller digit to subtract from a larger one only lets you use `I` to prefix `V` and `X`, or `X` to prefix `L` and `C`, or `C` to prefix `D` and `M`). The Romans didn't have a way of expressing the number 0, so to make your life easier we'll say that $\varepsilon \in L$ and that the empty string represents 0. (Oh, those silly Romans.) Write a regular expression for L.

(As a note, we're using the "standard form" of Roman numerals. You can see a sample of numbers written out this way via this link.)

Problem Two: Finite and Cofinite Languages

A language L is called ***finite*** if L contains finitely many strings (that is, $|L|$ is a natural number.) A language L is ***cofinite*** if its complement is a finite language; that is, L is cofinite if $|\overline{L}|$ is a natural number.

i. Given a finite language L, explain how to write a regular expression for L. Briefly justify your answer; no formal proof is necessary. This shows that all finite languages are regular.

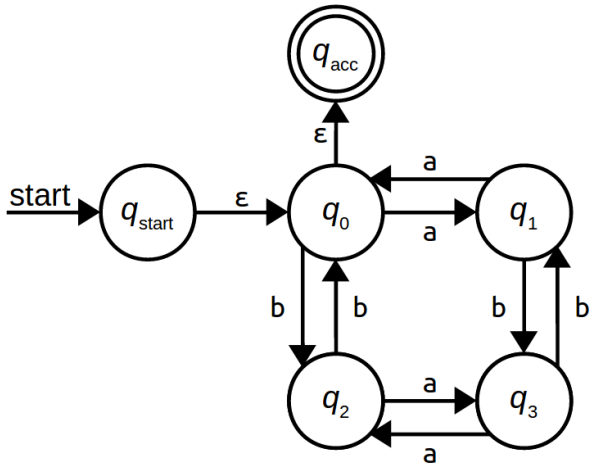
ii. Look up the ***trie*** data structure on Wikipedia. Explain, in your own words, how a trie can be thought of as a finite automaton. This gives another view on why finite languages are regular.

iii. Prove that any cofinite language is regular.

Problem Three: State Elimination

The state elimination algorithm gives a way to transform a finite automaton (DFA or NFA) into a regular expression. It's a really beautiful algorithm once you get the hang of it, so we thought that we'd let you try it out on a particular example.

Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ has an even number of a's and an even number of b's$\}$. Below is a finite automaton for L that we've prepared for the state elimination algorithm by adding in a new start state q_{start} and a new accept state q_{acc}:



We'd like you to use the state elimination algorithm to produce a regular expression for L.

i. Run two steps of the state elimination algorithm on the above automaton. Specifically, first remove state q_1, then remove state q_2. Show your result at this point.

Go slowly. Remember that to eliminate a state q, you should identify all pairs of states q_{in} and q_{out} where there's a transition from q_{in} to q and from q to q_{out}, then add shortcut edges from q_{in} to q_{out} to bypass state q. Remember that q_{in} and q_{out} can be the same state. If you've done everything right at the end of this stage, none of the transitions you have at this point should have Kleene stars in them.

ii. Finish the state elimination algorithm. What regular expression do you get for L?

You should start seeing Kleene stars appearing as you remove the remaining states.

iii. Without making reference to the original automaton given above, give an intuitive explanation for how the regular expression you found in part (ii) works.

Problem Four: Distinguishable Strings

The Myhill-Nerode theorem is one of the trickier and more nuanced theorems we've covered this quarter. This question explores what the theorem means and, importantly, what it *doesn't* mean.

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid |w| \text{ is even} \}$.

i. Show that L is a regular language.

There are lots of ways you can go about doing this. Pick whatever is easiest.

ii. Prove that there is a infinite set $S \subseteq \Sigma^*$ where there are infinitely many pairs of distinct strings $x, y \in S$ such that $x \not\equiv_L y$.

An analogy that might help: do you understand the difference between "a group of six people, any two of which are friends" versus "a group of six people, of which six pairs of those people are friends?"

iii. Prove that there is *no* infinite set $S \subseteq \Sigma^*$ where *all* pairs of distinct strings $x, y \in S$ satisfy $x \not\equiv_L y$.

The distinction between parts (ii) and (iii) is important for the Myhill-Nerode theorem. A language is nonregular not if you can find infinitely many pairs of distinguishable strings, but rather if you can find infinitely many strings that are all *pairwise* distinguishable.

Problem Five: At Least Three Point Five

The Myhill-Nerode theorem is a powerful tool for finding nonregular languages, but it can take some adjusting to get used to.

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and consider the following language:

$$L = \{ w \in \Sigma^* \mid \text{there are at least as many } \mathtt{a}\text{'s as } \mathtt{b}\text{'s in } w \}.$$

Below is a purported proof that L is not a regular language.

> ***Theorem:*** L is not a regular language.
>
> ***Proof:*** Consider the set $S = \{ \mathtt{a}^n \mid n \in \mathbb{N} \}$. This set is infinite because it contains one string for each natural number. We will prove that any two distinct strings in S are distinguishable relative to L. To do so, consider any distinct strings $\mathtt{a}^m, \mathtt{a}^n \in S$, and assume without loss of generality that $m > n$. Then $\mathtt{b}^m\mathtt{a}^m \in L$ because this string contains the same number of $\mathtt{a}$'s and $\mathtt{b}$'s, but $\mathtt{b}^m\mathtt{a}^n \notin L$ because it contains m $\mathtt{b}$'s and n $\mathtt{a}$'s and $m > n$. Therefore, we see that $\mathtt{a}^m \not\equiv_L \mathtt{a}^n$. This means that S is an infinite set of strings that are pairwise distinguishable relative to L. Therefore, by the Myhill-Nerode theorem, L is not regular. ■

Unfortunately, this proof is incorrect.

i. What's wrong with this proof? Be as specific as possible.

The best way to identify a flaw in a proof is to point to a specific claim that's being made that's not true or not properly substantiated and to explain why.

ii. Is L a regular language? If so, show why it's regular. If not, prove that L is not a regular language.

Think about the intuition behind what makes a language regular or not regular. If your intuition tells you that this language is regular, then use that intuition to guide your justification. If your intuition tells you that this language is not regular, use that intuition to guide your disproof. And if you don't have an intuition, take a look back at the lecture slides and feel free to stop by office hours!

Problem Six: Balanced Parentheses

Consider the following language over $\Sigma = \{$ `(`, `)` $\}$:

$$L_1 = \{\ w \in \Sigma^* \mid w \text{ is a string of balanced parentheses } \}$$

For example, we have `()` $\in L_1$, `(())` $\in L_1$, `(()())()` $\in L_1$, $\varepsilon \in L_1$, and `(())((()()))` $\in L_1$, but `)(` $\notin L_1$, `(()` $\notin L_1$, and `((()))` $\notin L_1$. This question explores properties of this language.

i. Prove that L_1 is not a regular language. One consequence of this result – which you don't need to prove – is that most languages that support some sort of nested parentheses, such as most programming languages and HTML, aren't regular and so can't be parsed using regular expressions.

As with all problems involving nonregular languages, proceed with this one in stages. First, ask yourself: if you were reading an input string from left to right, what information would you necessarily have to keep track of? The Myhill-Nerode theorem asks you to find an infinite set of strings that are all pairwise distinguishable, so try creating an infinite set of strings, one for each possible value that this information could take on.

Let's say that the *nesting depth* of a string of balanced parentheses is the maximum number of unmatched open parentheses at any point inside the string. For example, the string `((()))` has nesting depth three, the string `(()())()` has nesting depth two, and the string ε has nesting depth zero.

Consider the language $L_2 = \{\ w \in \Sigma^* \mid w$ is a string of balanced parentheses and w's nesting depth is at most four $\}$. For example, `((()))` $\in L_2$, `(()())` $\in L_2$, and `((((()))))(())` $\in L_2$, but `(((((())))))` $\notin L_2$ because although it's a string of balanced parentheses, the nesting goes five levels deep.

ii. Design a DFA for L_2, showing that L_2 is regular. A consequence of *this* result is that while you can't parse all programs or HTML with regular expressions, you can parse programs with low nesting depth or HTML documents without deeply-nested tags using regexes. ***Please submit this DFA using the DFA editor on the course website*** and tell us on GradeScope who submitted it.

iii. Look back at your proof from part (i) of this problem. Imagine that you were to take that exact proof and blindly replace every instance of "L_1" with "L_2." This would give you a (incorrect) proof that L_2 is nonregular (which we know has to be wrong because L_2 is indeed regular.) Where would the error be in that proof? Be as specific as possible.

Again, you should be able to point at a specific spot in the proof that contains a logic error and explain exactly why the statement in question is not true or not supported by the preceding statements. If you can't do this, it likely means you have an error in your proof from part (i)!

iv. Intuitively, regular languages correspond to problems that can be solved using only finite memory. Using this intuition and without making reference to DFAs, NFAs, or the Myhill-Nerode theorem, explain why L_1 is nonregular while L_2 is regular.

Problem Seven: Tautonyms

A ***tautonym*** is a word that consists of the same string repeated twice. For example, “caracara” and “dikdik” are all tautoynms (the first is a bird, the second the cutest animal you'll ever see), as is “hotshots” (people who aren't very fun to be around). Let $\Sigma = \{ \mathtt{a}, \mathtt{b} \}$ and consider the following language:

$$L = \{ ww \mid w \in \Sigma^* \}$$

This is the language of all tautonyms over Σ. Below is an ***incorrect*** proof that L is not regular:

> ***Proof:*** Let $S = \{ \mathtt{a}^n \mid n \in \mathbb{N} \}$. This set is infinite because it contains one string for each natural number. We claim that any two strings in S are distinguishable relative to L. To see this, consider any two distinct strings $\mathtt{a}^m$ and $\mathtt{a}^n$ in the set S, where $m \neq n$. Then $\mathtt{a}^m\mathtt{a}^m \in L$ but $\mathtt{a}^n\mathtt{a}^m \notin L$, so $\mathtt{a}^m \not\equiv_L \mathtt{a}^n$. This means that S is an infinite set of strings that are pairwise distinguishable relative to L. Therefore, by the Myhill-Nerode theorem, L is not regular. ■

i. What's wrong with this proof? Be as specific as possible.

This one is subtle, and it's not the same error that you identified earlier. As before, point out a specific claim that is made that isn't true or isn't supported by the preceding statements and justify why.

ii. Although the above proof is incorrect, the language L isn't regular. Prove this.

The most common mistake we see people make in part (ii) of this problem is to make the exact same sort of mistake that they identified in part (i) of this problem (oops!) So go slowly here and double-check your proof to make sure it's airtight, especially in light of what you saw in part (i). You may want to take a sentence or two to precisely articulate why each claim that you're making is true, since as you saw above it's easy for an erroneous statement to slip in under the radar if you're not careful!

Problem Eight: State Lower Bounds

The Myhill-Nerode theorem we proved in lecture is actually a special case of a more general theorem about regular languages that can be used to prove lower bounds on the number of states necessary to construct a DFA for a given language.

i. Let L be a language over Σ. Suppose there's a *finite* set S such that any two distinct strings $x, y \in S$ are distinguishable relative to L (that is, $x \not\equiv_L y$ for any two strings $x, y \in S$ where $x \neq y$.) Prove that any DFA for L must have at least $|S|$ states. (You sometimes hear this referred to as *lower-bounding* the size of any DFA for L.)

According to old-school Twitter rules, all tweets need to be 140 characters or less. Let Σ be the alphabet of characters that can legally appear in a tweet and consider the following language:

$$TWEETS = \{ w \in \Sigma^* \mid |w| \leq 140 \}.$$

This is the language of all legal tweets, assuming the empty string is a legal tweet. The good news is that this language is regular. The bad news is that any DFA for it has to be pretty large.

ii. Using your result from part (i), prove that any DFA for *TWEETS* must have at least 142 states.

It might be easier to tackle this problem if you consider replacing 140 and 142 with some smaller numbers (say, 2 and 4) to build up an intuition. And work backwards – what will you need to do to invoke part (i)?

iii. Refer back to the formal, 5-tuple definition of a DFA given in Problem Set Six. Define an 142-state DFA for *TWEETS* using the formal 5-tuple definition. Briefly explain how your DFA works. No formal proof is necessary.

Again, this might be a lot easier to do if you first reduce 140 and 142 to 2 and 4, respectively, and see what you come up with. Start by drawing out what the DFA would look like, then think about how you'd formalize your idea as a 5-tuple.

Your results from parts (ii) and (iii) collectively establish that the smallest possible DFA for *TWEETS* has exactly 142 states. This approach to finding the smallest object of some type – using some theorem to prove a lower bound ("we need at least this many states") combined with a specific object of the given type ("we certainly can't do worse than this") is a common strategy in discrete mathematics, algorithm design, and computational complexity theory. If you take classes like CS161, CS254, etc., you'll likely see similar sorts of approaches!

Problem Nine: Closure Properties Revisited

When building up the regular expressions, we explored several closure properties of the regular languages. This problem explores some of their nuances.

The regular languages are closed under complementation: If L is regular, so is $\overline{L}$.

i. Prove or disprove: the *nonregular* languages are closed under complementation.

In other words, if you have a nonregular language L, is $\overline{L}$ necessarily nonregular?

The regular languages are closed under union: If L_1 and L_2 are regular, so is $L_1 \cup L_2$.

ii. Prove or disprove: the *nonregular* languages are closed under union.

We know that the union of any two regular languages is regular. Using induction, we can show that the union of any finite number of regular languages is also regular. As a result, we say that the regular languages are closed under ***finite union***.

An infinite union is the union of infinitely many sets. Formally speaking, imagine that you have a sequence of infinitely many sets S_0, S_1, S_2, …, one for each natural number. Then the ***infinite union*** of those sets is defined as follows:

$$\bigcup_{i=0}^{\infty} S_i = \{ w \mid \exists n \in \mathbb{N}.\ w \in S_n \}.$$

For example, if L is a language, then L^*, which we defined as $\{ w \in \Sigma^* \mid \exists n \in \mathbb{N}.\ w \in L^n \}$, can be thought of as the infinite union

$$\bigcup_{i=0}^{\infty} L^i.$$

Similarly, the set of all rational numbers, $\mathbb{Q}$, can be thought of as

$$\mathbb{Q} = \bigcup_{i=0}^{\infty} \{ \frac{x}{i+1} \mid x \in \mathbb{Z} \}.$$

iii. Prove or disprove: the regular languages are closed under infinite union.

In other words, if you have an infinite list of regular languages L_0, L_1, L_2, …, is their infinite union necessarily a regular language?

Optional Fun Problem: Generalized Fooling Sets (1 Point Extra Credit)

In Problem Eight, you used distinguishability to lower-bound the size of DFAs for a particular language. Unfortunately, distinguishability is not a powerful enough technique to lower-bound the sizes of NFAs. In fact, it's in general quite hard to bound NFA sizes; there's a $1,000,000 prize for anyone who finds a polynomial-time algorithm that, given an arbitrary NFA, converts it to the smallest possible equivalent NFA!

Although it's generally difficult to lower-bound the sizes of NFAs, there are some techniques we can use to find lower bounds on the sizes of NFAs. Let L be a language over Σ. A ***generalized fooling set*** for L is a set $\mathscr{F} \subseteq \Sigma^* \times \Sigma^*$ is a set with the following properties:

- For any $(x, y) \in \mathscr{F}$, we have $xy \in L$.
- For any distinct pairs $(x_1, y_1), (x_2, y_2) \in \mathscr{F}$, we have $x_1y_2 \notin L$ or $x_2y_1 \notin L$ (this is an inclusive OR.)

Prove that if L is a language and there is a generalized fooling set $\mathscr{F}$ for L that contains n pairs of strings, then any NFA for L must have at least n states.

Problem Set 8

In this problem set, you'll transition away from the regular languages to the context-free languages and to the realm of Turing machines. This will be your first foray beyond the limits of what computers can over hope to accomplish, and we hope that you find this as exciting as we do!

As always, please feel free to drop by office hours or ask on Piazza if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, March 9th at 2:30PM.

Problem One: Designing CFGs

For each of the following languages, design a CFG for that language. ***Please use our online tool to design, test, and submit the CFGs in this problem***. To use it, visit the CS103 website and click the "CFG Editor" link under the "Resources" header. You should only have one member from each team submit your grammars; tell us who this person is when you submit the rest of the problems through GradeScope.

i. Given Σ = {`a`, `b`, `c`}, write a CFG for the language { $w \in \Sigma^*$ | w contains `aa` as a substring }. For example, the strings `aa`, `baac`, and `ccaabb` are all in the language, but `aba` is not.

ii. Given Σ = {`a`, `b`}, write a CFG for the language { $w \in \Sigma^*$ | w is *not* a palindrome }, the language of strings that are not the same when read forwards and backwards. For example, `aab` $\in L$ and `baabab` $\in L$, but `aba` $\notin L$, `bb` $\notin L$, and $\varepsilon \notin L$.

Don't try solving this one by starting with the CFG for palindromes and making modifications to it. In general, there's no way to mechanically turn a CFG for a language L into a CFG for the language $\overline{L}$, since the context-free languages aren't closed under complementation. However, the idea of looking at the first and last characters of a given string might be a good idea.

iii. Let Σ be an alphabet containing these symbols:

Ø ℕ { } , ∪

We can form strings from these symbols which represent sets. Here's some examples:

Ø	{Ø, ℕ} ∪ ℕ ∪ Ø	{Ø} ∪ ℕ ∪ {ℕ}	{Ø, Ø, Ø}
{{ℕ, Ø} ∪ {Ø}}	ℕ ∪ {ℕ, Ø}	{}	{ℕ}
{Ø, {Ø, {Ø}}}	{{{{ℕ}}}}	ℕ	{Ø, {}}

Notice that some of these sets, like {Ø, Ø} are syntactically valid but redundant, and others like {} are syntactically valid but not the cleanest way of writing things. Here's some examples of strings that don't represent sets or aren't syntactically valid:

ε	}Ø{	Ø{ℕ}	{{}
ℕ, Ø, {Ø}	{, ℕ}	{ ℕ Ø },	{,}
{Ø	}} ℕ	{ Ø, Ø, Ø, }	{ℕ, , , Ø}

Write a CFG for the language { $w \in \Sigma^*$ | w is a syntactically valid string representing a set }. When using the CFG tool, please use the letters n, u, and o in place of ℕ, ∪, and Ø, respectively.

Fun fact: The starter files for Problem Set One contain a parser that's designed to take as input a string representing a set and to reconstruct what set that is. The logic we wrote to do that parsing was based on a CFG we wrote for sets and set theory. Take CS143 if you're curious how to go from a grammar to a parser!

Test your CFG thoroughly! In Fall 2017, a quarter of the submissions we received weren't able to derive the string {Ø, Ø, Ø}.

Problem Two: The Complexity of Addition

This problem explores the following question:

How hard is it to add two numbers?

Suppose that we want to check whether $x + y = z$, where x, y, and z are all natural numbers. If we want to phrase this as a problem as a question of strings and languages, we will need to find some way to standardize our notation. In this problem, we will be using the ***unary number system***, a number system in which the number n is represented by writing out n 1's. For example, the number 5 would be written as `11111`, the number 7 as `1111111`, and the number 12 as `111111111111`.

Given the alphabet $\Sigma = \{$`1`, `+`, `=`$\}$, we can consider strings encoding $x + y = z$ by writing out x, y, and z in unary. For example:

4 + 3 = 7 would be encoded as `1111+111=1111111`

7 + 1 = 8 would be encoded as `1111111+1=11111111`

0 + 1 = 1 would be encoded as `+1=1`

Consider the alphabet $\Sigma = \{$`1`, `+`, `=`$\}$ and the following language, which we'll call *ADD*:

$$\{\ \texttt{1}^m\texttt{+1}^n\texttt{=1}^{m+n} \mid m, n \in \mathbb{N}\ \}$$

For example, the strings `111+1=1111` and `+1=1` are in the language, but `1+11=11` is not, nor is the string `1+1+1=111`.

i. Prove or disprove: the language *ADD* defined above is regular.

ii. Write a context-free grammar for *ADD*, showing that *ADD* is context-free. (Please submit your CFG online.)

You may find it easier to solve this problem if you first build a CFG for this language where you're allowed to have as many numbers added together as you'd like. Once you have that working, think about how you'd modify it so that you have exactly two numbers added together on the left-hand side of the equation.

Problem Three: The Complexity of Pet Ownership

This problem explores the following question:

How hard is it to walk your dog without a leash?

Let's imagine that you're going for a walk with your dog, but this time don't have a leash. As in Problem Set Six and Problem Set Seven, let $\Sigma = \{$`y`, `d`$\}$, where `y` means that you take a step forward and `d` means that your dog takes a step forward. A string in Σ^* can be thought of as a series of events in which either you or your dog moves forward one unit. For example, the string `yydd` means that you take two steps forward, then your dog takes two steps forward.

Let *DOGWALK* = $\{\ w \in \Sigma^* \mid w$ describes a series of steps where you and your dog arrive at the same point $\}$. For example, the strings `yyyddd`, `ydyd`, and `yyyddddddyyy` are all in *DOGWALK*.

i. Prove or disprove: the language *DOGWALK* defined above is regular.

ii. Write a context-free grammar for *DOGWALK*, showing that *DOGWALK* is context-free. (Please submit your CFG online.)

Be careful, and test your CFG! As you saw in lecture, a lot of ideas that seem plausible here don't work.

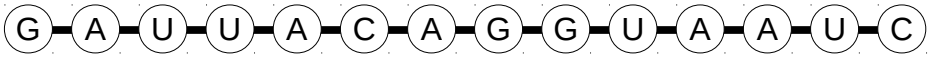
Problem Four: The Complexity of RNA Hairpins

RNA strands consist of strings of *nucleotides*, molecules which encode genetic information. Computational biologists typically represent each RNA strand as a string made from four different letters, `A`, `C`, `G`, and `U`, each of which represents one of the four possible nucleotides.

Each of the the four nucleotides has an affinity for a specific other nucleotide. Specifically:

`A` has an affinity for `U` (and vice-versa) `C` has an affinity for `G` (and vice-versa)

This can cause RNA strands to fold over and bind with themselves. Consider this RNA strand:



If you perfectly fold this RNA strand in half, you get the following:

Notice that each pair of nucleotides – except for the `A` and the `G` on the far right – are attracted to the corresponding nucleotide on the other side of the RNA strand. Because of the natural affinities of the nucleotides in the RNA strand, the RNA strand will be held in this shape. This is an example of an *RNA hairpin*, a structure with important biological roles.

For the purposes of this problem, we'll say that an RNA strand forms a hairpin if

- it has even length (so that it can be cleanly folded in half);
- it has length at least ten (there are at least four pairs holding the hairpin shut); and
- all of its nucleotides, except for the middle two, have an affinity for its corresponding nucleotide when folded over. (The middle two nucleotides in a hairpin might coincidentally have an affinity for one another, but it's not required. For example, `CCCCAUGGGG` forms a hairpin.)

This problem explores the question

How hard is it to determine whether an RNA strand forms a hairpin?

Let Σ = {`A`, `C`, `G`, `U`} and let L_{RNA} = { $w \in \Sigma^*$ | w represents an RNA strand that forms a hairpin }. For example, the strings `UGACCCGUCA`, `GUACAAGUAC`, `UUUUUUUUUAAAAAAAAA`, and `CCAACCUUGG` are all in L_{RNA}, but the strings `AU`, `AAAAACUUUUU`, `GGGC`, and `GUUUUAAAAG` are all not in L_{RNA}.

i. Prove that L_{RNA} is not regular. Since this language imposes a lot of requirements on the strings it contains, if in the course of your proof you want to claim that a particular string is or is not in L_{RNA}, please articulate clearly why the string does or does not meet each of the requirements of strings in L_{RNA}.

There's a good amount of trial and error required here. Test your proof carefully – pick concrete examples of strings from your set and make sure that your argument really does work for them. The fact that the two middle characters don't have to match makes this a little bit trickier than you might initially suspect.

ii. Design a CFG for L_{RNA}, which proves that the language is context-free. Please submit your grammar online.

Problem Five: Right-Linear Grammars

A context-free grammar is called a ***right-linear grammar*** if every production in the grammar has one of the following three forms:

- A → ε
- A → B, where B is a nonterminal.
- A → aB, where a is a terminal and B is a nonterminal.

For example, the following is a right-linear grammar:

A → aB | bB | ε

B → aC | bA | C

C → bA | aA | ε

The right-linear grammars are all context-free grammars, so their languages are all context-free. However, it turns out that this class of grammars precisely describe the regular languages. That is, a language *L* is regular if and only if there is a right-linear grammar *G* such that $L = \mathscr{L}(G)$.

i. Let *G* be a right-linear grammar. Describe how to start with *G* and construct an NFA *N* such that $\mathscr{L}(G) = \mathscr{L}(N)$. Briefly justify why *N* accepts precisely the strings that *G* can generate. To illustrate your construction, show the NFA you'd build from the following right-linear grammar:

A → aB | bC

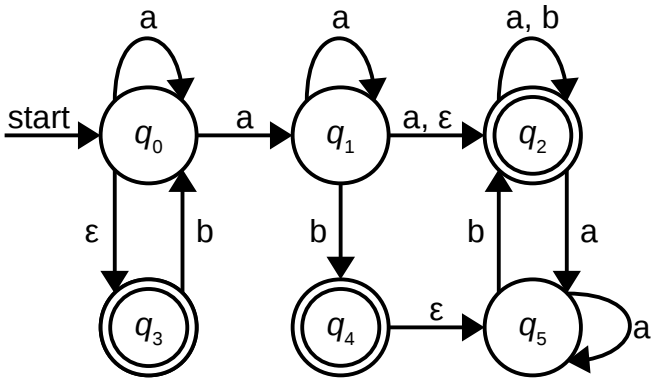
B → aB | ε

C → aD | A | bC

D → aD | bD | ε

Please submit your NFA for this question through our DFA/NFA editor.

ii. Let *N* be an NFA. Describe how to start with *N* and construct a right-linear grammar *G* such that $\mathscr{L}(G) = \mathscr{L}(N)$. Briefly justify why *N* accepts precisely the strings that *G* can generate. To illustrate your construction, show the grammar that you'd build from the following NFA:



Please submit your right-linear grammar through our online CFG tool.

We're expecting you to actually apply your construction to this NFA, not to eyeball the NFA and find a grammar that happens to have the same language. The point of doing this is to make sure that you've thought through all of the necessary cases.

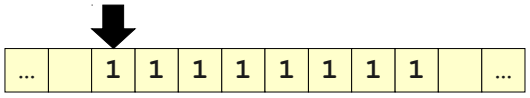
Problem Six: The Collatz Conjecture

The *Collatz conjecture* is a famous conjecture (an unproved claim) that says the following procedure (called the *hailstone sequence*) terminates for all positive natural numbers n:

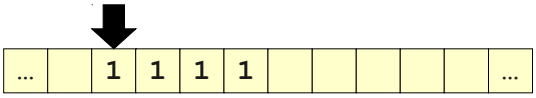
- If $n = 1$, stop.
- If n is even, set $n = n / 2$ and repeat from the top.
- If n is odd, set $n = 3n + 1$ and repeat from the top.

Let $L = \{ 1^n \mid n \geq 1$ and the hailstone sequence terminates for $n \}$ be a language over the singleton alphabet $\Sigma = \{1\}$. It turns out that it's possible to build a TM for this language, which means that $L \in$ **RE**, and in this problem you'll do just that. Parts (i) and (ii) will ask you to design two useful subroutines, and you'll assemble the overall machine in part (iii).

i. Design a TM subroutine that, given a tape holding a string of the form 1^{2n} (where $n \in \mathbb{N}$) surrounded by infinitely many blanks, ends with 1^n written on the tape, surrounded by infinitely many blanks. You can assume the tape head begins reading the first 1 (or points to an arbitrary blank cell in the case where $n = 0$), and your TM should end with the tape head reading the first 1 of the result (or any blank cell if $n = 0$). For example, given the initial configuration



your TM subroutine would end with this configuration:



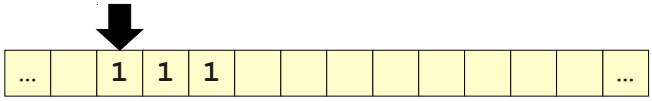
You can assume that there are an even number of 1s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done.

For reference, our solution has fewer than 10 states. If you have significantly more than this and are struggling to get your TM working, you might want to change your approach. It's totally fine if you have a bunch of states, provided that your solution works.

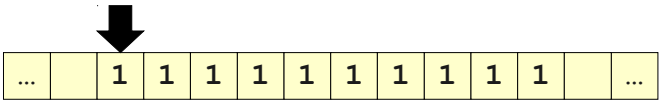
There are a lot of different solutions here. Some use very little extra tape. Some use a lot of extra tape. Some don't need any other tape symbols. Some do. Be creative, try things out, and don't be afraid to back up and try something else if your approach doesn't seem to be panning out.

(Continued on the next page)

ii. Design a TM subroutine that, given a tape holding a string of the form 1^n (for some $n \in \mathbb{N}$), surrounded by infinitely many blanks, ends with 1^{3n+1} written on the tape, surrounded by infinitely many blanks. You can assume that the tape head begins reading the first 1, and your TM should end with the tape head reading the first 1 of the result. For example, given this configuration



your TM subroutine would end with this configuration:



You can assume the number of 1s on the tape at startup is odd and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done. *(For reference, our solution has fewer than 10 states. If you have significantly more than this, you might want to change your approach.)*

iii. Draw the state transition diagram for a Turing machine *M* that recognizes *L*. Our TM tool is configured for this problem so that you can use our reference solutions for parts (i) and (ii) as subroutines in your solution. To do so, follow these directions:

1. Create states named `half`, `half_`, `trip`, and `trip_`.
2. To execute the subroutine that converts 1^{2n} into 1^n, transition into the state named `half`. When that subroutine finishes, the TM will automagically jump into the state labeled `half_`. You do not need to – and should not – define any transitions into `half_` or out of `half`.
3. To execute the subroutine that converts 1^n into 1^{3n+1}, transition into the state named `trip`. When that subroutine finishes, the TM will automagically jump into the state labeled `trip_`. You do not need to – and should not – define any transitions into `trip_` or out of `trip`.

Calling ahead to Monday's lecture: a TM *M* recognizes a language *L* if *M* accepts all of the strings in *L* and either rejects or loops on all strings that are not in *L*. In other words, your TM should accept every string in *L*, and for any string not in *L* it can either loop infinitely or reject the string.

Please use our provided TM editor to design, develop, test, and submit your answer to this question. *(For reference, our solution has fewer than 15 states. If you have significantly more than this, you might want to change your approach.)*

Problem Seven: TMs, Formally

Just as it's possible to formally define a DFA using a 5-tuple, it's possible to formally define a TM as an 8-tuple $(Q, \Sigma, \Gamma, B, q_0, Y, N, \delta)$ where

- Q is a finite set of states, which can be anything;
- Σ is a finite, nonempty set called the ***input alphabet***;
- Γ is a finite, nonempty set called the ***tape alphabet***, where $\Sigma \subseteq \Gamma$;
- $B \in \Gamma - \Sigma$ is the ***blank symbol***;
- q_0 is the start state, where $q_0 \in Q$;
- $Y \subseteq Q$ is the set of ***accepting states***;
- $N \subseteq Q$ is the set of ***rejecting states***, where $Y \cap N = \varnothing$; and
- δ is the ***transition function***, described below.

This question explores some aspects of the definition.

i. Is it possible to have a TM with no states? Justify your answer.

ii. Is it possible to have a TM with no *accepting* states? Justify your answer.

iii. Is it possible to have a TM with no *rejecting* states? Justify your answer.

iv. Why is the restriction $Y \cap N = \varnothing$ there? Justify your answer.

v. Is it possible to have a TM where $\Sigma = \Gamma$? Justify your answer.

Now, let's talk about the transition function. As with DFAs, the transition function of a Turing machine is what formally defines the transitions. If q is a state in a TM that isn't an accepting state or a rejecting state and a is a symbol that can appear on the TM's tape, then

$$\delta(q, a) = (r, b, D)$$

where r is the new state to transition into, b is the symbol to write back to the tape, and D is either L for "move left" or R for "move right." Because TMs immediately stop running after entering an accepting or rejecting state, the δ function should not be defined for any state q that's either accepting or rejecting. Aside from this, δ should be defined for every combination of a (non-accepting, non-rejecting) state q and any symbol a that can appear on the tape.

vi. Based on the above description of δ, what should the domain of δ be? What should it codomain be? Answer this question by filling in the following blanks, and briefly justify your answer.

$$\delta : \_\_\_\_\_ \rightarrow \_\_\_\_\_$$

Problem Eight: Regular and Decidable Languages

In class, we alluded to the fact that **REG** (the class of all regular languages) is a subset of **R** (the class of all decidable languages), but we never actually justified this claim.

Describe a construction that, given a DFA D, produces a decider D' where $\mathscr{L}(D) = \mathscr{L}(D')$. Briefly justify why the TM D' you construct is a decider and why it accepts precisely the strings that D accepts. Illustrate your example by applying it to a small DFA D of your choice.

Although you have a formal 5-tuple definition of a DFA and a formal 8-tuple definition of a TM at your disposal, we're not expecting you to write your solution at that level of detail.

Remember that DFAs and TMs work completely differently with regards to accepting and rejecting states and that the transitions in TMs have a very different structure than the transitions in DFAs!

Problem Nine: Jumbled Jargon

(We will cover the material necessary to solve this problem on Monday.)

We've introduced a number of terms and definitions pertaining to Turing machines, languages, and what it means to solve a problem. Some of the terms we've described are adjectives that can only describe TMs, while others are adjectives that can only describe languages. Using them incorrectly leads to statements that aren't mathematically meaningful.

To reason by analogy, consider the statement "the set $\mathbb{N}$ is even." This statement isn't meaningful, because "even" can only be applied to individual natural numbers, and $\mathbb{N}$ isn't a natural number. Similarly, the statement $1 \in 5$ isn't meaningful, since 5 isn't a set. The statement $\mathbb{Z} \subseteq \mathbb{N}$ is meaningful but not true – it's the mathematical equivalent of a grammatically correct statement that just happens to be false.

Below is a series of statements. For each statement, decide whether that statement is mathematically meaningful or not. If it's not mathematically meaningful, explain why not. If it is mathematically meaningful, determine whether it's true or false and briefly justify your answer.

i. If M is a Turing machine, w is a string, and M accepts w, then A_{TM} accepts $\langle M, w \rangle$.

ii. If M is a Turing machine, w is a string, and M loops on w, then $\langle M, w \rangle \notin \mathscr{L}(U_{TM})$.

iii. U_{TM} is decidable.

iv. $\langle U_{TM} \rangle$ is decidable.

v. $\{\langle U_{TM} \rangle\}$ is decidable.

You have a ton of experience type-checking things by this point in the quarter. Use that intuition here.

Problem Ten: What Does it Mean to Solve a Problem?

Let L be a language over Σ and M be a TM with input alphabet Σ. Below are three properties that may hold for M:

1. M halts on all inputs.
2. For any string $w \in \Sigma^*$, if M accepts w, then $w \in L$.
3. For any string $w \in \Sigma^*$, if M rejects w, then $w \notin L$.

At some level, for a TM to claim to solve a problem, it should have at least some of these properties. Interestingly, though, just having two of these properties doesn't say much.

i. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (2).

ii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (3).

iii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (2) and (3).

iv. Suppose that L is a language over Σ for which there is a TM M that satisfies properties (1), (2), and (3). What can you say about L? Prove it.

The whole point of this problem is to show that you have to be extremely careful about how you define "solving a problem," since if you define it incorrectly then you can "solve" a problem in a way that bears little resemblance to what we'd think of as solving a problem. Keep this in mind as you work through this one.

Problem Eleven: R and RE Languages

The following problems are designed to explore some of the nuances of how Turing machines, languages, decidability, and recognizability all relate to one another. We hope that by working through them, you'll get a much better understanding of key computability concepts.

i. Give a TM M such that $\mathscr{L}(M) \in \mathbf{R}$, but M is not a decider (you can draw a concrete example of TM, or give pseudocode for a program along the lines of what we've done in class). Briefly justify your answer. This shows that just because a TM's language is decidable, it's not necessarily the case that the TM itself must be a decider.

ii. Only *languages* can be decidable or recognizable; there's no such thing as an "undecidable string" or "unrecognizable string." Prove that for every string w, there's an **R** language containing w and an **RE** language containing w.

iii. Here's a weird one. Let Σ be an alphabet containing all characters that can appear in a person's name. Prove that the following language L_{2020} is decidable, subject to the assumption that there is a single US presidential election in 2020 and that it ends with a single winner:

$$L_{2020} = \{ w \in \Sigma^* \mid w \text{ is the name of the winner of the 2020 presidential election } \}$$

Then, explain how it's possible to build a decider for the language L_{2020} given that no one has any idea who is going to win the 2020 election!

Optional Fun Problem: TMs and Regular Languages (1 Point Extra Credit)

Let M be a TM with the following property: there exists a natural number k such that after M is run on any string w, M always halts after at most k steps. Prove that $\mathscr{L}(M)$ is regular.

Problem Set 9

What problems are beyond our capacity to solve? Why are they so hard? And why is anything that we've discussed this quarter at all practically relevant? In this problem set – the last one of the quarter! – you'll explore the absolute limits of computing power.

Before attempting any of the problems on this problem set, we strongly recommend reading over the ***Guide to Self-Reference*** and ***Guide to the Lava Diagram*** that are available on the course website, which provide a ton of extra background that you might find useful here.

As always, please feel free to drop by office hours or ask questions on Piazza if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, March 16th at 2:30PM

Because this problem set is due on the last day of class, no late days may be used and no late submissions will be accepted. Sorry about that! On the plus side, we'll release solutions as soon as the problem set comes due.

Problem One: Isn't Everything Undecidable?

(We recommend reading the Guide to Self-Reference on the course website before attempting this problem.)

In lecture, we proved that A_{TM} and the halting problem are undecidable – that, in some sense, they're beyond the reach of algorithmic problem-solving. The proofs we used involved the nuanced technique of self-reference, which can seem pretty jarring and weird the first time you run into it. The good news is that with practice, you'll get the hang of the technique pretty quickly!

One of the most common questions we get about self-reference proofs is why you can't just use a self-reference argument to prove that *every* language is undecidable. As is often the case in Theoryland, the best way to answer this question is to try looking at some of the ways you might use self-reference to prove that *every* language is undecidable, then see where those arguments break down.

To begin with, consider this proof:

> ***Theorem:*** All languages are undecidable.
>
> ***Proof:*** Suppose for the sake of contradiction that there is a decidable language L. This means there's a decider for L; call it `inL`.
>
> Now, consider the following program, which we'll call P:
>
> ```
> int main() {
>     string input = getInput();
>
>     /* Do the opposite of what's expected. */
>     if (inL(input)) {
>         reject();
>     } else {
>         accept();
>     }
> }
> ```
>
> Now, given any input w, either $w \in L$ or $w \notin L$. If $w \in L$, then the call to `inL(input)` will return true, at which point P rejects w, a contradiction! Otherwise, if $w \notin L$, then the call to `inL(input)` will return false, at which point P accepts w, a contradiction!
>
> In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, no languages are decidable. ■

This proof has to be wrong because we know of many decidable languages.

i. What's wrong with this proof? Be as specific as possible.

Go one sentence at a time and check that each claim is correct. Something is fishy here.

Here's another incorrect proof that all languages are undecidable:

> ***Theorem:*** All languages are undecidable.
>
> ***Proof:*** Suppose for the sake of contradiction that there is a decidable language *L*. This means that there is some decider *D* for the language *L*, which we can represent in software as a method `willAccept`. Then we can build the following self-referential program, which we'll call *P*:
>
> ```
> int main() {
>     string me = mySource();
>     string input = getInput();
>
>     /* See whether we'll accept, then do the opposite. */
>     if (willAccept(me, input)) {
>         reject();
>     } else {
>         accept();
>     }
> }
> ```
>
> Now, given any input *w*, program *P* either accepts *w* or it does not accept *w*. If *P* accepts *w*, then the call to `willAccept(me, input)` will return true, at which point *P* rejects *w*, a contradiction! Otherwise, we know that *P* does not accept *w*, so the call to `willAccept(me, input)` will return false, at which point *P* accepts *w*, a contradiction!
>
> In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, no languages are decidable. ■

It's a nice read, but this proof isn't correct.

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

Many of the examples we've seen of undecidable languages involve checking for properties of Turing machines or computer programs, which might give you the sense that *every* question you might want to ask about TMs or programs is undecidable. That isn't the case, though, and this question explores why.

Consider the following language L:

$$L = \{ \langle P \rangle \mid P \text{ is a syntactically valid C++ program } \}$$

Below is a purported proof that L is undecidable:

> ***Theorem:*** The language L is undecidable.
>
> ***Proof:*** Suppose for the sake of contradiction that L is decidable. That means that there's some decider D for L, which we can represent in software as a function `isSyntacticallyValid` that takes as input a program and then returns whether that program has correct syntax. Given this function, consider the following program P:
>
> ```
> int main() {
>     string me = mySource();
>
>     /* Execute a line based on whether our syntax is right. */
>     if (isSyntacticallyValid(me)) {
>         oops, this line of code isn't valid C++!
>     } else {
>         int num = 137; // Perfectly valid syntax!
>     }
> }
> ```
>
> Now, either this program P is syntactically valid or it is not. If P has valid syntax, then when P is run on any input, it will get its own source code, determine that it is syntactically valid, then execute a syntactically invalid line of code – a contradiction! Otherwise, if P is not syntactically valid, then when P is run on any input, it will get its own source code, determine that it is not syntactically valid, at which point it executes a perfectly valid line of C++ code – a contradiction!
>
> In either case we reach a contradiction, so our assumption must have been incorrect. Therefore, L is undecidable. ■

This proof, unfortunately, is incorrect.

iii. What's wrong with this proof? Be as specific as possible.

Problem Two: Password Checking

(We recommend reading the Guide to Self-Reference on the course website before attempting this problem.)

If you're an undergraduate here, you've probably noticed that the dorm staff have master keys they can use to unlock any of the doors in the residences. That way, if you ever lock yourself out of your room, you can, sheepishly, ask for help back in. (Not that I've ever done that or anything.) Compare this to a password system. When you log onto a website with a password, you have the presumption that your password is the only possible password that will log you in. There shouldn't be a "master key" password that can unlock any account, since that would be a huge security vulnerability. But how could you tell? If you had the source code to the password checking system, could you figure out whether your password was the only password that would grant you access to the system?

Let's frame this question in terms of Turing machines. If we wanted to build a TM password checker, "entering your password" would correspond to starting up the TM on some string, and "gaining access" would mean that the TM accepts your string. Let's suppose that your password is the string `iheartquokkas`. A TM that would work as a valid password checker would be a TM M where $\mathscr{L}(M) = \{\texttt{iheartquokkas}\}$: the TM accepts your string, and it doesn't accept anything else. Given a TM, is there some way you could tell whether the TM was a valid password checker?

Consider the following language L:

$$L = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) = \{\texttt{iheartquokkas}\} \}$$

Your task in this problem is to prove that L is undecidable (that is, $L \notin \mathbf{R}$). This means that there's no algorithm that can mechanically check whether a TM is suitable as a password checker. Rather than dropping you headfirst into this problem, we've split this problem apart into a few smaller pieces.

Let's suppose for the sake of contradiction that $L \in \mathbf{R}$. That means that there is some function

```
bool isPasswordChecker(string program)
```

with the following properties:

- If `program` is the source of a program that accepts just the string `iheartquokkas`, then calling `isPasswordChecker(program)` will return `true`.
- If `program` is not the source of a program that accepts just the string `iheartquokkas`, then calling `isPasswordChecker(program)` will return `false`.

We can try to build a self-referential program that uses the `isPasswordChecker` function to obtain a contradiction. Here's a first try:

```
int main() {
    string me = mySource();
    string input = getInput();

    if (isPasswordChecker(me)) {
        reject();
    } else {
        accept();
    }
}
```

This code is, essentially, a (minimally) modified version of the self-referential program we used to get a contradiction for the language A_{TM}.

(Continued on the next page.)

i. Prove that the above program *P* is not a valid password checker.

What is the definition of a password checker? Based on that, what do you need to prove to show that P is not a password checker?

ii. Suppose that this program is ***not*** a valid password checker. Briefly explain why no contradiction arises in this case – no formal justification is necessary.

A good question to think about in the course of answering part (ii) of this problem: this program is very close to the one from the proof that A_{TM} is not decidable. Why do you get a contradiction in the original proof that A_{TM} is undecidable? Why doesn't that same contradiction work here?

Ultimately, the goal of building a self-referential program here is to have the program cause a contradiction regardless of whether or not it's a password checker. As you've seen in part (ii), this particular program does not cause a contradiction if it isn't a password checker. Consequently, if we want to prove that $L \notin \mathbf{R}$, we need to modify it so that it leads to a contradiction in the case where it is not a password checker.

iii. Modify the above code so that it causes a contradiction regardless of whether it's a password checker. Then, briefly explain why your modified program is correct. (No formal proof is necessary here; you're going to do that in the next step.)

Follow the advice from the Guide to Self-Reference. Write out a specification of what your self-referential program is trying to do. Based on that, craft code for each of the two cases.

iv. Formalize your argument in part (iii) by proving that $L \notin \mathbf{R}$. Use the proof that $A_{TM} \notin \mathbf{R}$ as a template for your proof.

Problem Three: L_D, Cantor's Theorem, and Diagonalization

Here's another perspective of the proof that $L_D \notin \mathbf{RE}$. Suppose we let *TM* be the set of all encodings of Turing machines. That is,

$$TM = \{ \langle M \rangle \mid M \text{ is a TM} \}$$

We can then define a function $\hat{L} : TM \to \wp(TM)$ as follows:

$$\hat{L}(\langle M \rangle) = \mathscr{L}(M) \cap TM$$

This question explores some properties of this function.

i. Briefly describe, in plain English, what $\hat{L}(\langle M \rangle)$ represents.

You shouldn't need more than a sentence.

ii. Trace through the proof of Cantor's theorem from the Guide to Cantor's Theorem, assuming that the choice of the function *f* in the proof is the function $\hat{L}$. What is the set *D* that is produced in the course of the proof? Why?

Problem Four: Double Verification

This problem explores the following beautiful and fundamental theorem about the relationship between the **R** and **RE** languages:

If L is a language, then $L \in \mathbf{R}$ if and only if $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$

This theorem has a beautiful intuition: it says that a language L is decidable ($L \in \mathbf{R}$) precisely if for every string in the language, it's possible to prove it's in the language ($L \in \mathbf{RE}$) and, simultaneously, for every string not in the language, it's possible to prove that the string is not in the language ($\overline{L} \in \mathbf{RE}$). In this problem, we're going to ask you to prove one of the two directions of this theorem.

Let L be a language where $L \in \mathbf{RE}$ and $\overline{L} \in \mathbf{RE}$. This means that there's a verifier V_{yes} for L and a verifier V_{no} for $\overline{L}$. In software, you could imagine that V_{yes} and V_{no} correspond to methods with these signatures:

```
bool imConvincedIsInL(string w, string c)
bool imConvincedIsNotInL(string w, string c)
```

Prove that $L \in \mathbf{R}$ by writing pseudocode for a function

```
bool isInL(string w)
```

that accepts as input a string w, then returns true if $w \in L$ and returns false if $w \notin L$. Then, write a brief proof explaining why your pseudocode meets these requirements. You don't need to write much code here. If you find yourself writing ten or more lines of pseudocode, you're probably missing something.

The theorem you proved in this problem is extremely useful for building an intuition for what languages are decidable. You'll see this in the next problem.

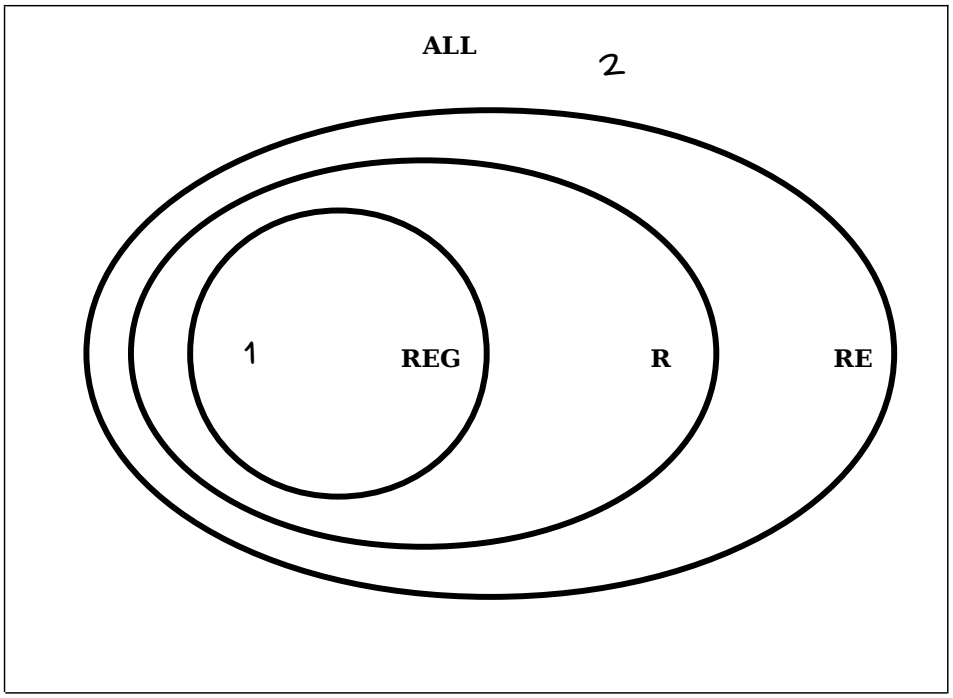
What other constructions have we done on verifiers? How did they work?

Problem Five: The Lava Diagram

Below is a Venn diagram showing the overlap of different classes of languages we've studied so far. We have also provided you a list of twelve numbered languages. For each of those languages, draw where in the Venn diagram that language belongs. As an example, we've indicated where Language 1 and Language 2 should go. No proofs or justifications are necessary – the purpose of this problem is to help you build a better intuition for what makes a language regular, **R**, **RE**, or none of these.

We ***strongly*** recommend reading over the Guide to the Lava Diagram before starting this problem.

To submit your answers, edit the file `LavaDiagram.h` in the `src/` directory of the starter files for this problem set.



1. Σ^*
2. L_D
3. $\{ \mathtt{a}^n \mid n \in \mathbb{N} \}$
4. $\{ \mathtt{a}^n \mid n \in \mathbb{N}$ and is a multiple of 137 $\}$
5. $\{ \mathtt{1}^n\mathtt{+}\mathtt{1}^m \stackrel{?}{=} \mathtt{1}^{n+m} \mid m, n \in \mathbb{N} \}$
6. $\{ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) \neq \varnothing \}$
7. $\{ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = \varnothing \}$
8. $\{ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = L_D \}$
9. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***accepts*** all strings in its input alphabet of length at most $n \}$
10. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***rejects*** all strings in its input alphabet of length at most $n \}$
11. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***loops*** on all strings in its input alphabet of length at most $n \}$
12. $\{ \langle M_1, M_2, M_3, w \rangle \mid M_1, M_2$, and M_3 are TMs, w is a string, and at least two of M_1, M_2, and M_3 accept $w. \}$

Problem Six: The Big Picture

We have covered a *lot* of ground in this course throughout our whirlwind tour of computability and complexity theory. This last question surveys what we have covered so far by asking you to see how everything we have covered relates.

Take a minute to review the hierarchy of languages we explored:

$$\textbf{REG} \subsetneq \textbf{CFL} \subsetneq \textbf{P} \stackrel{?}{=} \textbf{NP} \subsetneq \textbf{R} \subsetneq \textbf{RE} \subsetneq \textbf{ALL}$$

The following questions ask you to provide examples of languages at different spots within this hierarchy.

i. Give an example of a regular language. How might you prove that it is regular? You don't need to actually prove that it's regular – just tell us what proof technique you'd use.

ii. Give an example of a context-free language is not regular. How might you prove that it is context-free? How might you prove that it is not regular? Just tell us what techniques you'd use for these proofs; no formal proof is actually necessary here.

iii. Give an example of a language in **P**.

iv. Give an example of an **NP**-complete language. *(We'll talk about this on Wednesday.)*

v. Give an example of a language in **RE** not contained in **R**. How might you prove that it is **RE**? How might you prove that it is not contained in **R**? Again, we just need to proof techniques you'd use, not actual formal proofs.

vi. Give an example of a language that is not in **RE**. How might you prove it is not contained in **RE**? As before, we're just looking for proof techniques, not actual proofs.

Problem Seven: Class Participation Opt-Out

You have the option to choose to shift the 5% weight of your grade that's normally associated with your class participation onto the final exam. If you do this, you will receive no credit based on participation, and your final exam will be worth 40% of your grade rather than the regular 35%.

If you'd like to opt out of your class participation grade and shift the weight to the final, follow **this link** and fill out the form. ***Do not submit your answer through GradeScope; Keith and Cynthia won't see it***.

If you'd like to keep 5% of your grade for participation, you don't need to do anything. You're all set!

Optional Fun Problem: Quine Relays (1 Point Extra Credit)

Write four different C++ programs with the following properties:

- Running the first program prints the complete source code of the second program.
- Running the second program prints the complete source code of the third program.
- Running the third program prints the complete source code of the fourth program.
- Running the fourth program prints the complete source code of the first program.
- None of the programs perform any kind of file reading.

In other words, we'd like a collection of four different programs, each of which prints the complete source of the next one in the sequence, wrapping back around at the end. You can download starter files for this assignment from the course website and should submit your files through GradeScope.

This is actually a really fun problem to try. Once you figure out the trick, it's not that hard to code it up.

Grand Challenge Problem: P $\stackrel{?}{=}$ NP (Worth an A+, $1,000,000, and a Ph.D)

Prove or disprove: **P = NP**.

Take fifteen minutes and try this. Seriously. And if you can't crack this problem, feel free to submit your best effort, or the silliest answer you can think of.